Automated CPU Design by Learning from Input-Output Examples

Shuyao Cheng[1,2,3], Pengwei Jin[1,2,3], Qi Guo[1], Zidong Du[1,4], Rui Zhang[1], Xing Hu[1,4], Yongwei Zhao[1], Yifan Hao[1], Xiangtao Guan[5], Husheng Han[1,2], Zhengyue Zhao[1,2], Ximing Liu[1,2], Xishan Zhang[1,3], Yuejie Chu[1], Weilong Mao[1], Tianshi Chen[3], Yunji Chen[1,2] *

[1]State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences
[3]Cambricon Technologies
[4]Shanghai Innovation Center for Processor Technologies
[5]University of Science and Technology of China

Abstract

Designing a central processing unit (CPU) requires intensive manual work of talented experts to implement the circuit logic from design specifications. Although considerable progress has been made in electronic design automation (EDA) to relieve human efforts, all existing tools require hand-crafted formal program codes (e.g., Verilog, Chisel, or C) as the input. To automate the CPU design without human programming, we are motivated to learn the CPU design from only input-output (IO) examples, which are generated from test cases of design specification. The key challenge is that the learned CPU design should have almost zero tolerance for inaccuracy, which makes well-known approximate algorithms such as neural networks ineffective.

We propose a new AI approach to generate the CPU design in the form of a large-scale Boolean function, from only external IO examples instead of formal program code. This approach employs a novel graph structure called Binary Speculative Diagram (BSD) to approximate the CPU-scale Boolean function accurately. We propose an efficient BSD expansion method based on *Boolean Distance*, a new metric to quantitatively measure the structural similarity between Boolean functions, gradually increasing the design accuracy up to 100%. Our approach generates an industrial-scale RISC-V CPU design within 5 hours, reducing the design cycle by about 1000$\times$ without human involvement. The taped-out chip, `Enlightenment-1`, the world's first CPU designed by AI, successfully runs the Linux operating system and performs comparably against the human-design Intel 80486SX CPU. Our approach even autonomously discovers human knowledge of the von Neumann architecture.

1 Introduction

CPU (central processing unit) design has long been considered one of the most challenging intellectual tasks humans have ever undertaken. In conventional CPU design flow, a team of talented engineers use formal programming languages (e.g., Verilog [IEEE, 1996], Chisel [Bachrach *et al.*, 2012], or C/C++ [Coussy *et al.*, 2009; Gajski *et al.*, 2012]) to implement the circuit logic of a CPU based on design specification. Then billions of dedicated test cases with both inputs and their expected outputs are developed to test the functionality of the circuit logic for verification and debugging, as shown in Fig. 1(a). Though modern commercial electronic design automation (EDA) tools such as logic synthesis [Rudell, 1989] or high-level synthesis (HLS) [McFarland *et al.*, 1990; Lahti *et al.*, 2018] tools are available to alleviate the manual efforts of designing circuit logic, all these tools require hand-crafted formal programming codes as input. Thus, the conventional design flow remains highly complex and needs non-trivial manual effort in hand-crafted formal program coding. A real case is that designing a new Intel CPU involves a team of more than 500 engineers with up to 2 years [Bentley, 2005].

To automate CPU design without tedious human program-

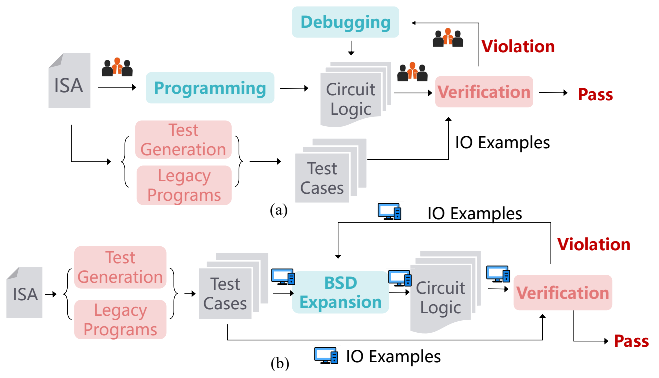


Figure 1: **Comparison of CPU design flow.** (a) The conventional design flow is an iterative process with huge manual efforts in programming, debugging, and verification of the circuit logic. (b) The proposed automated design flow, which learns the circuit logic only from input-output (IO) examples of test cases with the proposed BSD expansion, is a fully-automatic and iterative process that eliminates manual efforts on programming, debugging, and verification of the circuit logic.

*Yunji Chen (cyj@ict.ac.cn) is the corresponding author.

ming, we are motivated to automatically learn the circuit logic only from external input-output (IO) examples, which are easily accessible from a large number of legacy or automatically generated test cases. As shown in Fig. 1(b), in the proposed automated CPU design flow, the IO examples are either directly obtained from legacy programs with particular functionalities, or automatically generated with constrained test generation techniques [Kabylkas *et al.*, 2021; Adir *et al.*, 2004]. Different from the conventional manual design flow in Fig. 1(a), the proposed automated design flow eliminates manual efforts in the entire logic design flow, including programming, debugging and verification. Hence, the problem of automated CPU design can be formulated as generating the circuit logic in form of a large-scale Boolean function satisfying the IO specification.

The main challenge of the formulated problem is to ensure the strict accuracy of the generated large-scale Boolean function, i.e., to find circuit logic that is almost 100% accurate so that it can pass the functional verification. In the industrial practice of designing the Intel Pentium 4 microprocessor, one billion tests (each with $10,000$ instructions) are executed during verification, indicating that verification accuracy should be over 99.99999999999% [Bentley, 2001]. Due to error accumulation effects, larger-scale cases such as modern CPUs are more challenging.[1] Well-known probabilistic learning algorithms such as neural networks (NNs) have not ensured this strict accuracy even on much easier small-scale cases. For example, existing NN and reinforcement learning approaches only generate correct circuit logic of at most 200 logic gates [Rai *et al.*, 2021; Chen and Wang, 2012; Roy *et al.*, 2021]. With the manual description of the circuit structure, existing LLM (Large Language Model) methods can only design circuits with less than 1000 gates [Blocklove *et al.*, 2023; Fu *et al.*, 2023]. Moreover, these small-scale designs are only verified on limited number of test cases, without a sound accuracy guarantee for tape-out industrial implementation.

To address this accuracy challenge, we propose to automatically design CPU-scale circuit logic with a novel graph structure called Binary Speculation Diagram (BSD). BSD is an approximate representation of the well-known Binary Decision Diagram (BDD) [Akers, 1978]. In the BSD, certain sub-diagrams (nodes and their child nodes) of the conventional BDD are replaced with a speculated constant node 0 or 1. For every constant node 0 or 1 in the BSD, it can be further expanded with Boole's Expansion Theorem [Boole, 1854] to two sub-nodes to increase the approximate accuracy. We theoretically prove that along with the BSD expansion, its accuracy increases gradually up to 100%. Therefore, by continuously expanding the BSD scale to meet strict accuracy constraints from a compact approximation, the automated design flow iteratively enables verified implementation for large-scale circuit design.

Despite all possible BSD expansions ensuring design accuracy, it is computationally intractable to straightforwardly expand the BSD due to the exponentially increased size of BSD nodes. To address this issue, we proposed an efficient BSD expansion method to find a BSD with much fewer nodes while still maintaining design accuracy. The expansion method iteratively chooses a cluster of BSD nodes according to the Boolean Distance and expands these nodes with Boole's Expansion Theorem. Boolean Distance is a new metric to quantitatively measure the structural similarity between two Boolean functions, i.e. how many BSD nodes can be reused to represent these two Boolean functions. Considering the cluster of BSD nodes chosen by the Boolean Distance contains reusable graph structures, in each expansion iteration, the Boolean function can be represented with fewer BSD nodes by reusing these structures. Although the Boolean Distance needs exponential complexity to be calculated accurately, it can be efficiently approximated with the Monte Carlo method. We theoretically prove that by sampling more IO examples, the accuracy of the Monte Carlo method can achieve any arbitrary small error bound, therefore maintaining design accuracy.

The proposed automated design flow is validated by designing a 32-bit RISC-V CPU, i.e., `Enlightenment-1`. `Enlightenment-1` was automatically designed within only 5 hours and successfully taped out, running the Linux operating system and SPEC CINT2000 benchmark on it. `Enlightenment-1` performs comparably against the human-designed Intel 80486SX CPU[Intel, 1993], while the design cycle is significantly reduced by about $1000\times$. To our best knowledge, this is the world's first CPU automatically designed by AI, which may reform the semiconductor industry by significantly reducing the design cycles. We also demonstrate that our approach autonomously discovers the general von Neumann architecture [Von Neumann, 1993] from scratch.

This paper makes the following contributions:

- We propose a novel automated CPU design flow, which completely eliminates manual efforts on iterative design and verification in the conventional design flow and thus reduces design cycles significantly.
- We propose a new graph structure, i.e., BSD, to efficiently represent large-scale Boolean functions. BSD is theoretically proven to approximate the Boolean function with desirable accuracy.
- We propose an efficient BSD expansion method to generate the large-scale Boolean functions from sampled IO examples. By iteratively expanding the BSD guided by Boolean Distance, the final BSD with ultra-high accuracy can be generated efficiently.
- We apply the proposed method to automatically design a 32-bit RISC-V CPU within 5 hours. The taped-out CPU successfully runs the Linux OS and performs comparably against the human-designed Intel 80486SX CPU.

2 Problem Statement

Fig. 1(b) shows the proposed automated CPU design flow, which takes the IO examples of vast test cases as the inputs and then automatically generates the circuit logic of the CPU. The generated circuit logic will be checked with validation

[1]Even the Intel 80486 CPU designed in the 1990s has 1.2 million transistors, equivalent to about $300,000$ logic gates [Collen, 2011].

test cases as well. Once it fails to pass specific tests, more IO examples will be automatically sampled from the test cases to update the generated circuit logic. This is a fully automatic and iterative process where the generated circuit logic is verified and debugged till it passes a sufficient number of tests (e.g., one billion tests).

Since the sequential circuit logic can be simply unfolded as combinational circuit logic, the underlying problem of automatically generating the circuit logic becomes: with only *finite* IO examples, infer the circuit logic in form of *Boolean function* that can be generalized to infinite IO examples with extremely high accuracy. The reason to use finite IO examples is that it is impossible to obtain all IO examples for large-scale circuits as their number increases exponentially (2^n for n-bit input circuits). Based on the above analysis, the problem of automated CPU design can be formulated as follows:

Definition 1 (Automated CPU Design)**.** *There is an* Oracle $\phi : \{0,1\}^n \mapsto \{0,1\}^m$*, satisfying IO examples of the CPU. Given at most N input-output probes from the oracle* $\{(\mathbf{x}_1, \phi(\mathbf{x}_1)), (\mathbf{x}_2, \phi(\mathbf{x}_2)), \ldots, (\mathbf{x}_N, \phi(\mathbf{x}_N))\}$*, construct a Boolean function ψ to simulate ϕ, such that $\forall \mathbf{x} \in \{0,1\}^n$,*

$$P(\phi(\mathbf{x}) = \psi(\mathbf{x})) \geq 1 - \epsilon \; (\epsilon \to 0),$$

where the Boolean function ψ is the generated CPU design.

The key challenge of this problem is to obtain the large-scale circuit logic ψ (e.g., $1,000,000$ logic gates) with an extremely small ϵ (e.g., 10^{-13} corresponding to 99.99999999999% accuracy) given only a small number of probes (e.g. $N \approx 10^{40}$, comparing to all possible 10^{540} IOs). Existing machine learning methods fail to address this accuracy challenge, and the accuracy of the circuit designed by all of these methods is now less than 99.99% (i.e., $\epsilon < 10^{-4}$) with a size of at most 200 logic gates, which is multiple orders of magnitude smaller than that of a CPU. Therefore, it is required to design a new approach that can theoretically guarantee any given high accuracy even in an iterative manner.

3 Methodology

3.1 Circuit Representation with BSDs

To address the challenge of the automated CPU design flow, we propose a new graph structure called the Binary Speculation Diagram (BSD). BSD is an approximate representation of the Binary Decision Diagram (BDD), one of the most well-known and efficient data structure for representing large-scale circuits in the form of Boolean functions (details of BDD and circuit representations are in the Appendix). Based on BSD, the proposed automated design flow can obtain the accurate Boolean functions of large-scale circuits only from input-output examples.

The BSD is a rooted, directed acyclic graph (DAG) which consists of internal decision nodes and leaf speculation nodes. The internal decision node indicates a Boolean variable with the assignment of value 0 or 1 to its two child nodes, and the speculation nodes approximate the sub-functions represented by the child nodes with constant 0 or 1. Fig. 2 shows the circuit representation by BDD and BSD. BDD is the 100% accurate BSD representation, as shown in Fig. 2(a). If we

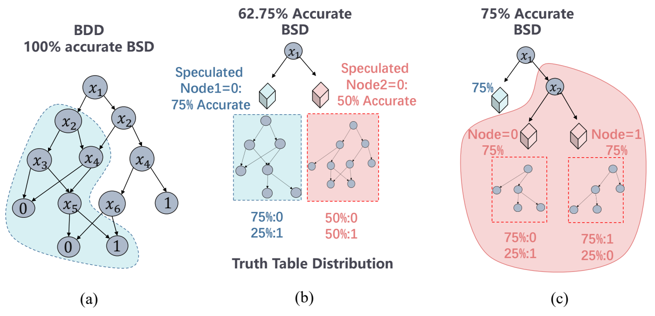


Figure 2: **The circuit logic representation by BDD and BSD.** (a) The BDD representation of a Boolean function with variables x_1 to x_6, also known as the 100% accurate BSD. (b) A 3-node BSD with two speculate nodes, the accuracy is 62.75% as the accuracy of its two child nodes are 75% and 50%. (c) Expand one speculated node, and get a 5-node BSD with three speculated nodes, the accuracy increases to 75%. The circles are the decision nodes, and the cubes are the speculated nodes, which approximately represent a sub-graph in the box below with constant 0/1.

use only one decision node to approximate the corresponding BDD, the result is shown in Fig. 2(b). Each speculated node in the BSD approximately represents a sub-function of the BDD, with constant 0 or 1 according to the IOs. For example, the blue speculated node on the left side in Fig. 2(b) represents the Boolean function F in the blue box below, $F \leftrightarrow (\neg x_2 \wedge x_3 \wedge x_5) \vee (x_2 \wedge x_4 \wedge x_5)$, and thus if it is speculated to constant 0, i.e. $F_{spec} = 0$, the accuracy $Acc = P(F = F_{spec}) = 0.75$. The total accuracy is the expectation of the accuracy of all the speculated nodes. Since the BDD is a BSD with no speculated node, it is a 100% accurate BSD. The BSD becomes more accurate when it has more nodes, as shown in Fig. 2(c), with one more decision node, the accuracy of the BSD increases from 62.75% to 75%.

Since the precise representation of the sub-functions requires multiple nodes, the speculation nodes trade off the representation accuracy with a more compact structure. To decrease the accuracy loss, each speculation node in the BSD can be expanded with the Boole's Expansion Theorem into two child speculation nodes, formulated as follows:

Definition 2 (BSD Expansion)**.** *In the k-th expanding iteration, the sub-function represented by the speculation node $\mathcal{F}_k(\mathtt{x})$ can be represented with one less variable $x_i \in \mathtt{x}$, where $\mathtt{x}$ is the input variable set,*

$$\mathcal{F}_k(\mathtt{x}) = \overline{x_i}\mathcal{F}_{k+1}(\mathtt{x}|x_i = 0) + x_i\mathcal{F}_{k+1}(\mathtt{x}|x_i = 1).$$

Note that in the BSD expansion, similar to the original Boole's Expansion Theorem, the child function F_{k+1} is represented by either an existing node functional equivalent with it or a newly inserted node if no functional equivalent nodes exist in the BSD. By expanding the speculation nodes of the BSD, the BSD representation grows larger and more accurate and finally becomes exactly the same as the original BDD representation. We prove Theorem 1 (Proof in the Appendix): along with the BSD expansion, the accuracy of the BSD is increased gradually up to 100%.

Theorem 1 (The accuracy of BSD increases after node expansion)**.** *After expanding the generated BSD $\mathcal{F}_k(\mathtt{x})$ (shorted*

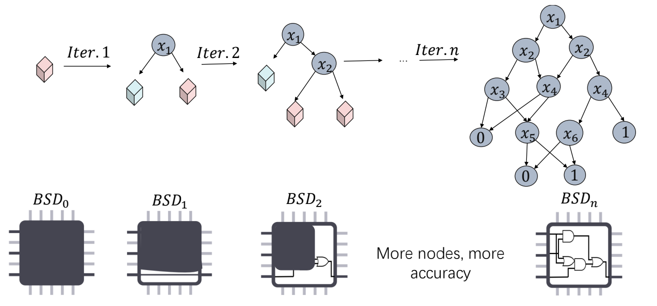


Figure 3: **Learning the circuit logic with BSD expansion.**

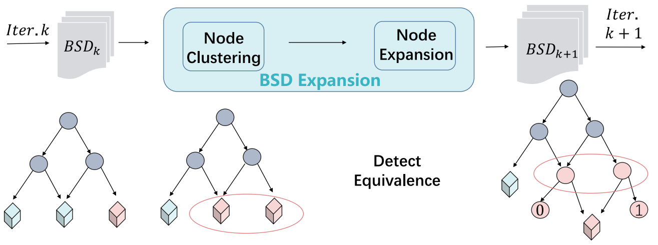


Figure 4: **An iteration of the BSD expansion.** In the k_{th} iteration of the expansion, the approach first find the speculated nodes to be expanded with the chosen node. Then, it detects the functional equivalence of the child-functions and get the $k + 1_{th}$ BSD.

as $\mathcal{F}_k$) by any input bit x_i to $\mathcal{F}_{k+1}$, the accuracy of expansion ended with $\mathcal{F}_k$ will be no larger than the accuracy of expansion ended with $\mathcal{F}_{k+1}$, that is,

$$Acc(\mathcal{F}_k) \leq Acc(\mathcal{F}_{k+1}). \quad (1)$$

Therefore, the challenge of achieving the strict accuracy constraint of a large-scale Boolean function can be addressed by iteratively expanding the BSD for a more accurate representation of the Boolean function. As shown in Fig. 3, the BSD is initialized as a root speculated node, and the desired Boolean function is a complete black box with only IO examples. The BSD gradually expands to increase the design accuracy, and some of the circuit logic is designed during expansion (the bottom line in the figure). Until the BSD passes the verification, the entire Boolean function is formulated by the BSD, and the circuit logic is designed.

Compared with other circuit representations (e.g., Karnaugh maps [Brown, 1990], canonical sum-of-products form [Geetha *et al.*, 2015], and neural network [Liang and Van den Broeck, 2019]), BSD shows the following key advantages. BSD is a compact form of the Boolean function and can reduce the description complexity of general Boolean functions from exponential to polynomial formally [Drechsler and Becker, 2013]. Furthermore, different from probabilistic ML methods, every part of the BSD representation is formal and can be checked and debugged with SAT solvers.

3.2 BSD Expansion Method

For a specific Boolean function, there exists infinite BSD representations which can guarantee the design accuracy but their sizes are quite different. Most of these BSD expansions are inefficient, i.e. a large amount of redundant BSD nodes are used to represent a Boolean function which can be represented with very few nodes. Unfortunately, existing methods[Qian *et al.*, 2023; Haaswijk *et al.*, 2018] need the formal expression of Boolean function which is unknown in automated CPU design flow, and fail to find the efficient representation with only IO examples. Thus, a new approach to efficiently represent the Boolean functions with fewer BSD nodes is urgently needed.

Algorithm 1 Learn the circuit logic with BSD Expansion

```
Require: Input-Output examples of the circuit logic
Ensure: Circuit logic represented by BSD
  Initialize $NODE_{spec} \leftarrow \{root\}$ , $BSD \leftarrow \emptyset$
  repeat
    choose a speculated node $n \in NODE$ to expand
    Initialize $CLUSTER \leftarrow \emptyset$
    for all speculated node $n_i \in NODE$ do
      if $n_i$ can be clustered with $n$ then
        CLUSTER.append($n_i$)
      end if
    end for
    Expand the nodes in $CLUSTER$,update $BSD$
        check functional equivalence,
        get new speculated nodes $CHILD$
    Update $NODE \leftarrow NODE \cup CHILD \setminus CLUSTER$
    Verify $Circuit$
  until Verification Pass
```

According to Definition 2, during BSD expansion, the BSD node number increases only if all nodes in the current BSD are not functionally equivalent to the child function. Therefore, it is necessary to detect BSD nodes which are functionally equivalent during expansion.

Based on this analysis, we proposed a BSD expansion method to iteratively detect the functional equivalence approximately to reduce the newly introduced BSD nodes. Algorithm 1 shows the learning process. The BSD is initialized as only one speculated node, which is the root of the BSD. In every single iteration, there are two steps, i.e. *clustering* and *expansion*. First, it chooses a node to be expanded, and finds other nodes to *cluster* with it based on the Boolean Distance according to Section 3.3. Then, the algorithm *expands* the clustered nodes together, according to Boole's Expansion Theorem, with the same variable to maximize the accuracy. If the expanded child node is functionally equivalent to the function represented by an existing BSD node, it is represented by the existing node. Otherwise, this child function is represented by a newly inserted node in the BSD. The functional equivalence is checked with a Monte Carlo method relying on the same expanded variables, i.e. sample a large set of inputs and check if all the outputs are the same, After the iteration, the speculated node-set and the BSD are updated, the expanded nodes are changed from speculated nodes to decision nodes, and the new child nodes are speculated nodes for further expansion. The iterative process continues until the BSD passes the verification.

Fig. 4 illustrates an iteration in the BSD expansion. The k_{th} iteration, i.e. $Iter.k$, start from the expanded BSD, BSD_k. The method first finds the nodes which are clustered with the chosen node, the red node, to be expanded, and then expands with the approximate functional equivalence. Fig. 4 shows an example, that if the right child node of the first expanded node, and the left child node of the second expanded node are functional equivalent, only one speculated node is inserted in the BSD expansion. After the BSD expansion iteration, the BSD updates, i.e. BSD_{k+1}, which increases the design accuracy.

The proposed design flow maintains the design accuracy, because the error introduced in the expansion, i.e. approximating the functional equivalence according to IO examples, is convergent, as shown in Thoerem 2 (proof in Appendix).

Theorem 2 (The convergence of expansion error)**.** *When we detect the functional equivalent nodes in the BSD expansion, the probability that the error of generated BSD larger than δ will no more than $\frac{T}{K\delta}$, where T is the total number of node reduction, K is the number of IO examples during calculating the functional equivalence, and δ is a very small value. Thus, the probability that the error of generated BSD is larger than a very small value will converge to zero by increasing the number of sampled IO examples.*

3.3 Node Clustering with Boolean Distance

In the BSD expansion process, a significant problem is how to cluster the nodes, i.e. what kind of nodes should be clustered to more efficient BSD simplification. If too many nodes are mis-clustered, the expansion is inefficient because of mixing up simplify patterns in the same cluster. On the contrary, if too few nodes are clustered, it is also inefficient because of missing merging opportunities between nodes in different clusters. An efficient cluster should contain nodes with structurally similar Boolean functions so that there are more child functions with functional equivalence.

To efficiently cluster the BSD nodes, we propose Boolean Distance to guide the BSD expansion process. Boolean Distance is a new metric to quantitatively measure the structural similarity between two Boolean functions, i.e. how many BSD nodes can be reused to represent these two Boolean functions, relying on the observation that many sub-graph structures can be reused to represent these functions. For a pair of Boolean functions, if the Boolean Distance $Dist{>}0$, showing that there are structures that can be reused, and these functions benefit from clustering. The larger Boolean Distance shows that the clustering is more efficient.

Definition 3 (Boolean Distance)**.** *Given two Boolean functions, f and g, the Boolean distance $Dist(f, g)$ is calculated based on the circuit complexity of f, g, and their combination τ.*

$$Dist(f, g) = C_\Omega(f) + C_\Omega(g) - C_\Omega(\tau), \tag{2}$$

where $C_\Omega(\cdot)$ is the circuit complexity of a Boolean function (defined by the circuit size), τ is the combination of f and g, the input bits of τ are the union of input bits of f and g, the output of τ is the concatenation of the output bits of f and g, formulated as

$$\tau(\mathbf{x}_{f\setminus g}, \mathbf{x}_{g\setminus f}, \mathbf{x}_{f\cap g}) = (f(\mathbf{x}_f), g(\mathbf{x}_g)), \tag{3}$$

where $\mathbf{x}_{f\setminus g}$ are the input bits that in f but not in g, $\mathbf{x}_{g\setminus f}$ are the input bits that in g but not in f, $\mathbf{x}_{f\cap g}$ are the input bits that in both f and g, $(\cdot, \cdot)$ means the concatenation of the output bits of two Boolean functions.

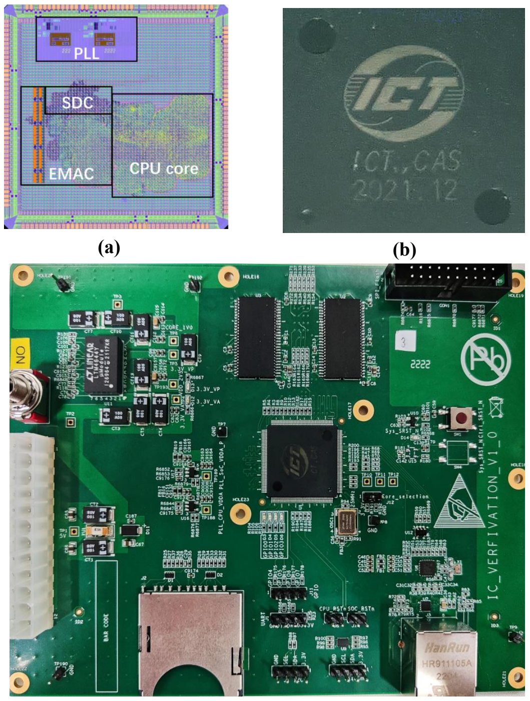


Figure 5: **The (a) layout, (b) manufactured chip, and (c) printed circuit board of `Enlightenment-1`.**

Straightforwardly, the Boolean Distance can be calculated accurately given concrete circuit logic according to the definition. However, in the learning process, without prior knowledge of the circuit structure, we use a Monte Carlo method to approximately calculate both the circuit complexity and the Boolean distance. For every Boolean function f, the circuit complexity $C_\Omega(f)$ can be recursively calculated by another BSD with few layers expanded and lower accuracy requirement, allowing to estimate the BSD size of function f in a short time. During the calculation of the circuit complexity $C_\Omega(f)$, the complexity of the leaf node of the BSD is approximated as the information entropy of the corresponding function. The approximate error of $C_\Omega(f)$ is also gradually reduced to 0 with the size of the BSD growing. When the BSD is fully expanded and thus exactly the same as f, the circuit complexity $C_\Omega(f)$ is an exact rather than an approximate value.

4 Evaluation

We use the proposed approach to automatically generate a 32-bit RISC-V CPU, `Enlightenment-1`, within 5 hours, and demonstrate that the approach can discover human knowledge of von Neumann architecture.

4.1 Automatically Design a RISC-V CPU

We use the proposed approach to generate the CPU design from a relatively small set of IO examples. Concretely, the

Target Circuit	Gate No.	Methods	Tape-out Time
Adder [Roy *et al.*, 2021]	118	RL	NA
Circuit Modules [Chen *et al.*, 2020]	186	DT	NA
Circuit Modules [Rai *et al.*, 2021]	$\sim$2500	EL	NA
8-bit CPU [Blocklove *et al.*, 2023]	999	LLM	2023-05
`Enlightenment-1`	4,272,680	BSD	2021-12

Table 1: Comparison with automated circuit design tasks.

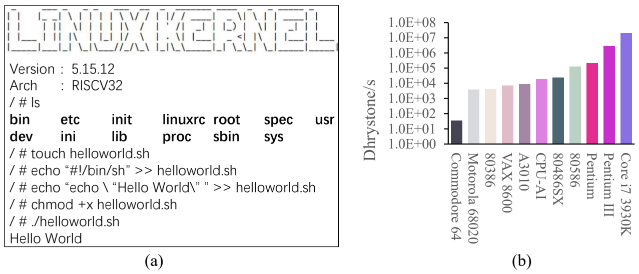
CPU has 1789 input bits and 1826 output bits, and thus the total number of IO examples is 1826×2^{1798}, while only less than 2^{40} IO examples are randomly sampled for training. The training process takes less than 5 hours to achieve the accuracy of $> 99.99999999999\%$ for validation tests (Detailed settings of the algorithm are in Appendix B). The generated CPU design then undergoes the physical design process with scripts at 65nm technology to generate the layout for fabrication. Figure 5 illustrates the layout of the entire chip with major components marked, the manufactured chip with a frequency of 300 MHz, and the printed circuit board containing the chip. Although we demonstrate the capability of our approach with the RISC-V32IA instruction set, it can generate the circuit logic of other CPUs with different instruction sets as long as we can obtain the IO examples.

We successfully run the Linux (kernel 5.15) operating system and SPEC CINT2000 [Henning, 2000] on `Enlightenment-1` to validate the functionality (see Figure 6(a)). We also use the widely-used Dhrystone [Weicker, 1984] to evaluate the performance. Figure 6(b) compares the performance of `Enlightenment-1` against different generations of commercial CPUs, e.g., Intel 80386 (1980s), Intel 80486SX (1990s), and Intel Pentium III (2000s). On the evaluated program, it performs comparably to Intel 80486SX, designed in mid-1991. Though `Enlightenment-1` performs worse than recent processors such as Intel Core i7 3930K, it is the world's first automatically designed CPU, and its performance could be significantly improved with augmented algorithms, which is left as our future work.

Reduction of development costs. To demonstrate that the proposed method reforms the CPU design flow, we further compare the development costs of `Enlightenment-1` and human-designed CPU (i.e., CPU-Man), where the internal registers of CPU-Man exhibit exactly the same behaviour as `Enlightenment-1` based on the same instruction set specification. The CPU-Man takes $\sim$ 5000 man-hours to complete the entire design, while `Enlightenment-1` only takes less than 5 hours to obtain the design by training from the IO examples. The reduction of development costs is further validated by the design and verification costs of an Intel 486-compatible CPU, K486, which takes more than 190 days (i.e., 4560 hours) merely for the verification process [Yim *et al.*, 1997]. The reason is that manual efforts in programming, debugging, and verification of circuit logic in the conventional CPU design flow is completely eliminated.

4.2 Comparison with State-of-the-art

Automated circuit design is an extremely challenging task, and thus there are only several recent studies on using machine learning methods to generate small-scale circuits, including LLMs, NNs, Reinforcement Learning (RL), Decision Trees (DTs) and Ensemble Learning (EL) methods.



```
Version :  5.15.12
Arch    :  RISCV32
/ # ls
bin     etc     init    linuxrc root    spec    usr
dev     ini     lib     proc    sbin    sys
/ # touch helloworld.sh
/ # echo "#!/bin/sh" >> helloworld.sh
/ # echo "echo \ "Hello World\" " >> helloworld.sh
/ # chmod +x helloworld.sh
/ # ./helloworld.sh
Hello World
```

Figure 6: **Functional validation and performance comparison.** (a) The outputs of booting up the Linux operating system. (b) The performance of `Enlightenment-1` is compared against commercial CPUs on the Dhrystone benchmark, and `Enlightenment-1` performs comparably to the human-designed Intel 80486SX CPU.

Experiment on small-scale circuit designs. As listed in Table 1, automatically designed by our approach is more than $1700\times$ larger than existing work, because they cannot yet handle the accuracy challenge of large-scale circuits such as CPUs. Besides, these designs are only verified on limited number of test cases, and cannot guarantee the strict design accuracy for taped-out industrial CPUs. In comparison, the proposed method can functionally design all the benchmark circuits solved by the SOTA methods in the ICCAD contest and Opencores accurately. The experiment benchmarks in these tasks are components of a large-scale processor, such as the logic/arithmetic unit or controllers in the processor, or over-simplified CPUs cannot even run the modern essential programs, such as the Linux operation system or other modern file systems.

Experiment on the large-scale CPU design. We apply state-of-the-art algorithms in the proposed CPU design flow to design the circuit logic, replacing the proposed BSD expansion method. However, all these methods fail due to the strict accuracy constraint. With only IOs as prompt without any structural description from the human designer, the LLM-based methods can only read 1 pair of IO examples with 2000 tokens, and impossible to design. The RL-based method does not have a proper initialization without a formal expression and fails to design due to the sparse rewards in the vast design space. All the DT-based methods fail to design, because without the merging possibility in the data structure, for such large-scale circuit design, it fails because of state explosion. NN methods can fit the IOs, but even the training errors are not close to 10^{-13}, and drops more in the quantization to produce the circuit logic.

4.3 Discovering the von Neumann Architecture

By detailing the generated circuit logic of `Enlightenment-1`, we demonstrate that our approach discovers human knowledge of von Neumann architecture only from the IO examples. Concretely, the generated CPU design in terms of BSD has the key component of the von Neumann architecture, which mainly consists of the control unit generated first in the BSD for global control, and the

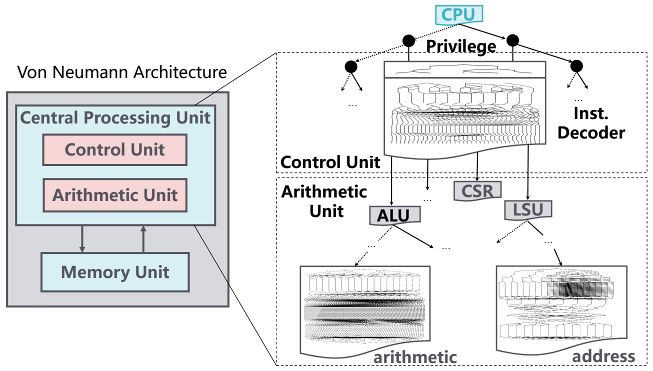


Figure 7: **Discovering von Neumann architecture from scratch.** The generated BSD mainly consists of the control unit and arithmetic unit, which can be further decomposed into sub-modules in the BSD, e.g., the control unit contains the privilege controller and instruction decoder, and the arithmetic unit contains ALU and LSU.

arithmetic unit (see Figure 7). The control unit generates the controlling signals for the entire CPU, and the arithmetic unit accomplishes arithmetic operations (e.g., ADD and SUB) and logic operations (e.g., AND and OR). Moreover, we observe that both the control unit and arithmetic unit can be recursively decomposed into smaller functional modules such as the instruction decoder, ALU, and LSU (load/store unit) by expanding more BSD layers.

5 Related Work

The design and verification process in conventional CPU design flow takes 60%-80% of the design efforts and resources [Bergeron, 2012]. To alleviate human efforts, various attempts have been practiced. High-level synthesis (HLS) has been proposed to agilely program the circuit logic, and automatic test generation has been proposed to automate the verification process. Besides, machine learning techniques have been developed to design the circuit logic without human programming automatically. However, all these methods cannot yet design a CPU-scale circuit without human participants.

High-level synthesis. The HLS tools emerged to generate RTL (register-transfer-level) description from behaviour specifications (e.g., C/C++ programs)[Cong *et al.*, 2022]. Since the early 2000s, it has been introduced by many EDA vendors commercially, and most of them used C, C++, SystemC, or Matlab as input [Nane *et al.*, 2015; Pursley and Yeh, 2017], producing both dataflow and control logic with reasonable performance. Recently, machine learning, especially deep learning techniques, have been employed to improve the efficiency of HLS, e.g., HLS quality estimation [Dai *et al.*, 2018], circuit performance predication [Ustun *et al.*, 2020; Yang *et al.*, 2022], overhead estimation [Mohammadi Makrani *et al.*, 2019], and search space optimization [Goswami *et al.*, 2023; Wang and Schafer, 2020; Kwon and Carloni, 2020; Liu and Carloni, 2013]. All the HLS methods require a formal description of the circuit logic, such as C/System C, while our approach completely eliminates manual efforts to develop such formal inputs. Instead, the circuit design is automated by directly reusing empirical IO examples from automatic test generation in the conventional verification process.

Automatic test generation. Test cases for verification can be either directly obtained from a large number of legacy programs with particular functionalities (e.g., SPEC CPU benchmarks or repositories with high-quality test cases), or automatically generated with constrained test generation techniques [Bose *et al.*, 2001; Fine and Ziv, 2003; Corno *et al.*, 2004]. Specifically, the test program generator takes the constrained directives as the input to produce the test cases. After simulating the test cases on the designed circuit logic with EDA tools, the coverage reports are collected and updated, which are used to guide the generation of new directives with AI techniques (e.g., genetic algorithms). Such a coverage-directed constrained test generation has already been used for the verification of commercial CPUs such as Intel X86 [Corno *et al.*, 2004], PowerPC [Fine and Ziv, 2003], and Godson [Shen *et al.*, 2008]. In short, the test cases, along with their IO examples, are easily accessible in traditional CPU design flow.

Automated circuit design without programming. There are several recent studies on using machine learning methods to generate small-scale circuits with human programming. The decision tree (DT) [Chen *et al.*, 2020] and ensemble learning (EL) [Rai *et al.*, 2021] are also used to generate small functional modules. Though Blocklove et al. proposed to generate an 8-bit CPU with intensive interactions between the large language model (LLM) and human engineers, the design is quite small with only ~ 1000 logic gates, and the usage of natural language prompts does not alleviate human efforts in the design[Blocklove *et al.*, 2023]. In short, the circuit automatically designed by our approach is more than $1700\times$ larger than existing work, and they cannot yet handle the accuracy challenge of large-scale circuits such as CPUs. More importantly, these small-scale designs are only verified on a limited number of test cases, without a sound accuracy guarantee for tape-out industrial implementation. Once the generated circuit logic cannot pass a given test, such machine learning methods, neural networks and reinforcement learning in particular cannot automatically debug and refine the generated circuit logic in order to pass the failed test.

6 Conclusion and Future Work

In this work, we propose a novel AI approach based on BSD to reform the traditional CPU design flow and thus obtain the world's first automatically generated CPU, i.e., `Enlightenment-1`. `Enlightenment-1` successfully runs the Linux operating system and performs comparably against the human-designed Intel 80486SX CPU. Moreover, compared to human-designed CPUs, our approach reduces the design cycle by about $1000\times$, because the manual programming and verification process of traditional CPU design is completely eliminated. In addition to offering human-like design abilities, our approach even autonomously discovers human knowledge of von Neumann architecture. In future work, the performance of the design can be further optimized with human knowledge, such as the module circuit library and design quality measurements for restructuring.

A Circuit Representation with BDDs

Binary Decision Diagram (BDD), a rooted, directed acyclic graph which consists of internal decision nodes and leaf nodes [Akers, 1978], is one of the most well-known and efficient data structures for representing large-scale circuits in form of Boolean functions. The internal decision node indicates a Boolean variable with the assignment of value 0 or 1 to its two child nodes, and the leaf nodes are labeled with 0 and 1. The node-operation of the BDD can be formulated as the Boolean Expansion Theorem [Boole, 1854]: left node with $x_i = 0$ and right with $x_i = 1$. Hence the corresponding Boolean function of the BDD node $\mathcal{F}()$ can be represented into two sub-functions with one less variables $\mathcal{F}(\mathrm{x}) = \overline{x_i}\mathcal{F}(\mathrm{x}|x_i = 0) + x_i\mathcal{F}(\mathrm{x}|x_i = 1)$ when expanding variable x_i.

The advantages of BDD over other circuit representations (e.g., Karnaugh maps [Brown, 1990], canonical sum-of-products form [Geetha *et al.*, 2015], and neural network [Liang and Van den Broeck, 2019]) is three-fold. The first is that BDD can reduce the description complexity of general Boolean functions from exponential to polynomial [Drechsler and Becker, 2013]. The second is that the tree-based structure of BDD is inherently interpretable and makes it possible to efficiently and accurately identify the incorrect circuit logic (i.e., sub-tree of the BDD) for potential refinement when the generated BDD cannot pass specific tests. The third and most important advantage is that as the BDD is expanded with more nodes, the represented partial Boolean function is closer to the final target, which might have the potential to provide an accuracy guarantee.

Nevertheless, traditional methods for BDD generation cannot be directly applied to solve our problem. The reason is that it requires a formal specification of the circuit logic or full observation of all possible IO examples [Drechsler and Becker, 2013].

B Theorem Proof

Theorem 1 (The accuracy of BSD boosts after expansion)**.** *After expanding the generated BSD $\mathcal{F}_k$ by any input bit x_i to $\mathcal{F}_{k+1}$, the accuracy of expansion ended with $\mathcal{F}_k$ will be no larger than the accuracy of expansion ended with $\mathcal{F}_{k+1}$, that is,*

$$Acc(\mathcal{F}_k) \leq Acc(\mathcal{F}_{k+1}). \tag{4}$$

Proof. Consider a node f_k in $\mathcal{F}_k$ when the expansion is ended with $\mathcal{F}_k$. Denote $Q_0(f_k)$ and $Q_1(f_k)$ as the proportions that the value of f_k are equal to 0 and 1, respectively. Thus, the larger one between $Q_0(f_k)$ and $Q_1(f_k)$ decides the value of f_k

$$Acc(f_k) = \max(Q_0(f_k), Q_1(f_k)). \tag{5}$$

We continue expand f_k to f_{k+1}^0 and f_{k+1}^1 with a selected input bit x_i assigning to 0 and 1, respectively. Denote $P_0(f_k)$ and $P_1(f_k)$ as the proportions that f_k are expanded with x_i assigning to 0 and 1, respectively, then we have

$$\begin{array}{rl} & Acc(\{f_{k+1}^0, f_{k+1}^1\}) \\ = & P_0(f_k)Acc(f_{k+1}^0) + P_1(f_k)Acc(f_{k+1}^1) \\ = & P_0(f_k)\max(Q_0(f_{k+1}^0), Q_1(f_{k+1}^0)) \\ + & P_1(f_k)\max(Q_0(f_{k+1}^1), Q_1(f_{k+1}^1)). \end{array} \tag{6}$$

We also have

$$Q_0(f_k) = P_0(f_k)Q_0(f_{k+1}^0) + P_1(f_k)Q_0(f_{k+1}^1), \tag{7}$$

$$Q_1(f_k) = P_0(f_k)Q_1(f_{k+1}^0) + P_1(f_k)Q_1(f_{k+1}^1). \tag{8}$$

Combine Eqn. (5) (7) (8), we can get

$$\begin{array}{rcl} Acc(f_k) & = & \max(P_0(f_k)Q_0(f_{k+1}^0) + P_1(f_k)Q_0(f_{k+1}^1), \\ & & P_0(f_k)Q_1(f_{k+1}^0) + P_1(f_k)Q_1(f_{k+1}^1)) \\ & \leq & \max(P_0(f_k)Q_0(f_{k+1}^0), P_0(f_k)Q_1(f_{k+1}^0)) \\ & + & \max(P_1(f_k)Q_0(f_{k+1}^1), P_1(f_k)Q_1(f_{k+1}^1)) \\ & = & P_0(f_k)\max(Q_0(f_{k+1}^0), Q_1(f_{k+1}^0)) \\ & + & P_1(f_k)\max(Q_0(f_{k+1}^1), Q_1(f_{k+1}^1)) \\ & = & Acc(\{f_{k+1}^0, f_{k+1}^1\}). \end{array} \tag{9}$$

When we apply Eqn. (9) to all nodes in $\mathcal{F}_k$, we can directly obtain Eqn. (4). □

Theorem 2 (The convergence of merging error)**.** *When we perform the merging stage during the process of generating BSD, the probability that the error of generated BSD larger than δ will no more than $\frac{T}{K\delta}$, where T is the total number of merging stages, K is the number of IO examples during calculating the Boolean distance, and δ is a very small value. Thus, the probability that the error of generated BSD is larger than a very small value will converge to zero by increasing the number of sampled IO examples.*

Proof. Assuming in one merging stage, we merge two leaf nodes that correspond to two not exactly the same Boolean functions. We denote r as the proportion of incorrect results of merged nodes after the merging stage, namely the proportion of different results of the two corresponding Boolean functions given the same input. Denote R as the error rate of merged nodes after any merging stage, then the distribution of R is

$$Pr(R) = \begin{cases} (1-r)^K & R = r \\ 1-(1-r)^K & R = 0 \end{cases}. \tag{10}$$

So the expected value of R is

$$E(R) = r(1-r)^K. \tag{11}$$

Considering the range of r is $(0, \frac{1}{2}]$, $E(R)$ increases monotonically at $r \in (0, \frac{1}{K+1})$, and decreases monotonically at $r \in (\frac{1}{K+1}, \frac{1}{2}]$. Therefore, $E(R)$ takes the maximum value $\frac{1}{K+1}\left(\frac{K}{K+1}\right)^K$ when r takes $\frac{1}{K+1}$, so

$$E(R) \leq \frac{1}{K+1}\left(\frac{K}{K+1}\right)^K < \frac{1}{K}. \tag{12}$$

Given a small value δ, using Markov's inequality, we can obtain

$$Pr(R \geq \delta) \leq \frac{E(R)}{\delta} < \frac{1}{K\delta}. \tag{13}$$

Eqn. (13) means that in each merging stage, the probability that the error rate after merging two leaf nodes larger than δ is less than $\frac{1}{K\delta}$. Thus, the probability that the accuracy of a merged node larger than $1-\delta$ is no less than $1-\frac{1}{K\delta}$. When

Component	**Area** (um^2)	**%**	**Power** (mW)	**%**
CPU core	275933.53		14.46	
Combinational	264476.88	95.85	8.70	60.14
Register	11456.64	4.15	5.76	39.86

Table 2: Hardware characteristics of `Enlightenment-1`.

finishing the process of generating the BSD, which contains T merging stages, the probability that the accuracy of the final BSD larger than $1-\delta$ is no less than $(1-\frac{1}{K\delta})^T$ which is no less than $1-\frac{T}{K\delta}$. In conclusion, the probability that the error of generated BSD is larger than δ will be no more than $\frac{T}{K\delta}$. □

C Detailed Design Configurations

We introduce detailed configurations of our approach, including IO samples, algorithm parameter settings, and the hardware implementation, to generate the 32-bit RISC-V CPU.

Data for BSD generation. The IO examples for automated CPU design are directly borrowed from the verification of traditional CPU design flow. There are two main kinds of IO examples for traditional verification. The first kind is the randomly generated input stimuli and their corresponding outputs, and the second kind is the input-outputs of legacy programs with particular functionalities, such as high-quality test cases. We start from RISC-V official ISA documents and end with a verification-passed GTECH netlist after synthesis.

Algorithmic parameters. During the generation of circuit logic in the form of BSD, there are multiple parameters to set. For the partition stage, the maximal number of clusters for each output bit is set as 10 to control the scale of the generated BSD. For the expansion stage, the maximal width of BSD is set as $10,000$ to balance the BSD accuracy and expansion efficiency. For the merging stage, to determine the similarity between different nodes, the maximal sampling number per node is set as $1,000,000$ to balance the reduction accuracy and sampling efficiency. The implemented program is executed on a Linux cluster including 68 servers, each of which is equipped with 2 Intel Xeon Gold 6230 CPUs.

Hardware implementation. After the generation of BSD, we tape out the CPU chip to demonstrate the effectiveness of our approach. Specifically, the generated BSD is converted into a Verilog module by traversing every node in it. Hence, the BSD generation is compatible with the existing back-end design flow. We use commercial tools to transform the Verilog module to a GTECH netlist, better supported by back-end EDA flows. We verify our output netlist on the FPGAs and tape out the chip with 65nm technology, and the detailed hardware characteristics are listed in Table 2. The automatically designed CPU, `Enlightenment-1`, was sent to the manufacturer in December 2021.

D An Illustrate Example

Here we use an 8-bit adder, $c[8:0]=a[7:0]+b[7:0]$, as an illustrative example to detail the process of the BSD expansion. The initialized BSD is only 9 root nodes, indicating the

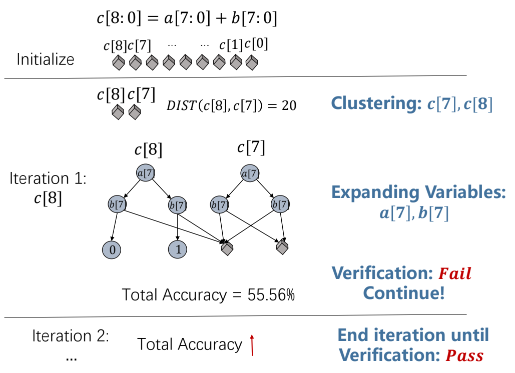


Figure 8: **An 8-bit full adder example of BSD expansion.** An iteration in the BSD expansion.

output bits $c[8:0]$. Figure 8(a) illustrates one single iteration of the BSD expand.

First, the highest bit $c[8]$ is chosen to be expanded first, and by calculating the Boolean Distance, $c[7]$ can be chosen to cluster with c[8], because many child nodes of these nodes can be reused. If $c[7]$ and $c[8]$ are individually designed, the total BSD nodes $C_\Omega(c_8)+C_\Omega(c_7)=43+23=66$; But if they are clustered, the total BSD nodes $C_\Omega(c_8,c_7)=46$. The Boolean Distance $Dist=C_\Omega(c_8)+C_\Omega(c_7)-C_\Omega(c_8,c_7)=$ 20 0 indicates that 20 BSD nodes can be reused, and it is efficient to cluster these nodes with each other.

Next, in this cluster with $c[8]$ and $c[7]$, the variable which expanded first is the variable which increases the accuracy the most. With the IO examples, it is clear that the most significant bit $a[7]$ and $b[7]$ should be expanded first. While expanding these nodes, merging possibilities can be found with Monte Carlo method according to Theorem 2. Specifically, $c[8]|(a[7]=0,b[7]=0)=0$, i.e. when $a[7]=0$ and $b[7]=0$ $c[8]=0$, thus the node is constant 0 and end its expansion. $c[8]|(a[7]=1,b[7]=1)=1$, thus the node is constant 0 and end its expansion. With Monte Carlo method, it is found that $c[8]|(a[7]=0,b[7]=1)=c[8]|(a[7]=1,b[7]=0)=c[7]|(a[7]=0,b[7]=1)=c[7]|(a[7]=1,b[7]=0)$, and thus these four sub-functions can be merged into one speculated node. Similarly, $c[7]|(a[7]=0,b[7]=0)=c[7]|(a[7]=1,b[7]=1)$ and thus these two nodes can also be merged. Therefore, after the 2-variable expansion, there remains 2 speculated nodes. Differently, if these nodes were not in the same cluster and expanded seperately, there would be 4 instead of 2 speculated nodes remaining, and 20 more nodes after the entire expansion, showing that the node clustering is efficient to reduce the BSD nodes.

Then after the iteration expanding two variables, the circuit logic updates with the BSD and checked if it can pass the verification. The iteration ends until it passes the verification. In this case, the current BSD cannot pass the verification, because the speculated nodes are not yet constant 0 or 1. Therefore the method choose another BSD node to start another iteration. Finally, when the designed BSD passes the verification, we output the BSD in Verilog, and it is the automated circuit logic design from our method.

Acknowledgments

This work is partially supported by the NSF of China(under Grants 61925208, U22A2028, 62222214, 62341411, 62102398, 62102399, U20A20227, 62372436, 62302478, 62302482, 62302483, 62302480), Strategic Priority Research Program of the Chinese Academy of Sciences, (Grant No. XDB0660200, XDB0660201, XDB0660300, XDB0660301), CAS Project for Young Scientists in Basic Research(YSBR-029), Youth Innovation Promotion Association CAS and Xplore Prize.

References

[Adir *et al.*, 2004] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2):84–93, 2004.

[Akers, 1978] Akers. Binary decision diagrams. *IEEE Transactions on computers*, 100(6):509–516, 1978.

[Bachrach *et al.*, 2012] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of Design Automation Conference*, pages 1212–1221, 2012.

[Bentley, 2001] Bob Bentley. Validating the intel pentium 4 microprocessor. In *Proceedings of Design Automation Conference*, pages 244–248, 2001.

[Bentley, 2005] Bob Bentley. Validating a modern microprocessor. In *Proceedings of International Conference on Computer Aided Verification*, pages 2–4, 2005.

[Bergeron, 2012] Janick Bergeron. *Writing testbenches: functional verification of HDL models*. Springer Science & Business Media, 2012.

[Blocklove *et al.*, 2023] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. Chip-chat: Challenges and opportunities in conversational hardware design. *arXiv preprint arXiv:2305.13243*, 2023.

[Boole, 1854] George Boole. *An Investigation of the Laws of Thought: On which are Founded the Mathematical Theories of Logic and Probabilities*. 1854.

[Bose *et al.*, 2001] Mrinal Bose, Jongshin Shin, Elizabeth M. Rudnick, Todd Dukes, and Magdy Abadir. A genetic approach to automatic bias generation for biased random instruction generation. In *Proceedings of the Congress on Evolutionary Computation*, pages 442–448 vol. 1, 2001.

[Brown, 1990] Frank Markham Brown. *Boolean Reasoning: The Logic of Boolean Equations*. 1990.

[Chen and Wang, 2012] Yu-Fang Chen and Bow-Yaw Wang. Learning boolean functions incrementally. In *Proceedings of International Conference on Computer Aided Verification*, pages 55–70, 2012.

[Chen *et al.*, 2020] Pei-Wei Chen, Yu-Ching Huang, Cheng-Lin Lee, and Jie-Hong Roland Jiang. Circuit learning for logic regression on high dimensional boolean space. In *Proceedings of Design Automation Conference*, pages 1–6. IEEE, 2020.

[Collen, 2011] Morris F Collen. *Computer medical databases: the first six decades (1950–2010)*. Springer Science & Business Media, 2011.

[Cong *et al.*, 2022] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. Fpga hls today: Successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems*, 15(4):1–42, 2022.

[Corno *et al.*, 2004] Fulvio Corno, Ernesto Sánchez, Matteo Sonza Reorda, and Giovanni Squillero. Automatic test program generation: a case study. *IEEE Design & Test of Computers*, 21(2):102–109, 2004.

[Coussy *et al.*, 2009] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[Dai *et al.*, 2018] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline F.Y. Young, and Zhiru Zhang. Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning. In *Proceedings of International Symposium on Field Programmable Custom Computing Machine*, pages 129–132, April 2018.

[Drechsler and Becker, 2013] Rolf Drechsler and Bernd Becker. *Binary decision diagrams: theory and implementation*. Springer Science & Business Media, 2013.

[Fine and Ziv, 2003] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Proceedings of Design Automation Conference*, pages 286–291, 2003.

[Fu *et al.*, 2023] Yonggan Fu, Yongan Zhang, Zhongzhi Yu, Sixu Li, Zhifan Ye, Chaojian Li, Cheng Wan, and Yingyan Celine Lin. Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models. In *International Conference on Computer Aided Design*, pages 1–9. IEEE, 2023.

[Gajski *et al.*, 2012] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[Geetha *et al.*, 2015] V Geetha, N Devarajan, and PN Neelakantan. Network structure for testability improvement in exclusive-or sum of products reed–muller canonical circuits. *International Journal of Engineering Research and General Science*, 3(3):368, 2015.

[Goswami *et al.*, 2023] Pingakshya Goswami, Benjamin Carrion Schaefer, and Dinesh Bhatia. Machine learning based fast and accurate high level synthesis design space exploration: From graph to synthesis. *Integration*, 88:116–124, 2023.

[Haaswijk *et al.*, 2018] Winston Haaswijk, Edo Collins, Benoit Seguin, Mathias Soeken, Frédéric Kaplan, Sabine Süsstrunk, and Giovanni De Micheli. Deep learning for logic optimization algorithms. In *International Symposium on Circuits and Systems*, pages 1–4. IEEE, 2018.

[Henning, 2000] John L Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[IEEE, 1996] IEEE. Ieee standard description language based on the verilog hardware description language, 1364-1995, 1996.

[Intel, 1993] Intel. Intel's sl enhanced intel486 (tm) microprocessor family. 1993.

[Kabylkas *et al.*, 2021] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. Effective processor verification with logic fuzzer enhanced co-simulation. In *Proceedings of the International Symposium on Microarchitecture*, pages 667–678, 2021.

[Kwon and Carloni, 2020] Jihye Kwon and Luca P Carloni. Transfer learning for design-space exploration with high-level synthesis. In *Proceedings of the ACM/IEEE Workshop on Machine Learning for CAD*, pages 163–168, 2020.

[Lahti *et al.*, 2018] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. Are we there yet? a study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):898–911, 2018.

[Liang and Van den Broeck, 2019] Yitao Liang and Guy Van den Broeck. Learning logistic circuits. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[Liu and Carloni, 2013] Hung-Yi Liu and Luca P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *Proceedings of Design Automation Conference*, pages 1–7, May 2013.

[McFarland *et al.*, 1990] Michael C McFarland, Alice C Parker, and Raul Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, 1990.

[Mohammadi Makrani *et al.*, 2019] Hosein Mohammadi Makrani, Farnoud Farahmand, Hossein Sayadi, Sara Bondi, Sai Manoj Pudukotai Dinakarrao, Houman Homayoun, and Setareh Rafatirad. Pyramid: Machine Learning Framework to Estimate the Optimal Timing and Resource Usage of a High-Level Synthesis Design. In *Proceedings of the International Conference on Field Program-mable Logic and Applications*, pages 397–403, 2019.

[Nane *et al.*, 2015] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2015.

[Pursley and Yeh, 2017] David Pursley and Tung-Hua Yeh. High-level low-power system design optimization. In *Proceedings of International Symposium on VLSI Design, Automation and Test*, pages 1–4, 2017.

[Qian *et al.*, 2023] Yu Qian, Xuegong Zhou, Hao Zhou, and Lingli Wang. An efficient reinforcement learning based framework for exploring logic synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 2023.

[Rai *et al.*, 2021] Shubham Rai, Walter Lau Neto, Yukio Miyasaka, Xinpei Zhang, Mingfei Yu, Qingyang Yi, Masahiro Fujita, Guilherme B Manske, Matheus F Pontes, Leomar S da Rosa, et al. Logic synthesis meets machine learning: Trading exactness for generalization. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition*, pages 1026–1031, 2021.

[Roy *et al.*, 2021] Rajarshi Roy, Jonathan Raiman, Neel Kant, Ilyas Elkin, Robert Kirby, Michael Siu, Stuart Oberman, Saad Godil, and Bryan Catanzaro. Prefixrl: Optimization of parallel prefix circuits using deep reinforcement learning. In *Proceedings of Design Automation Conference*, pages 853–858. IEEE, 2021.

[Rudell, 1989] Richard L Rudell. *Logic synthesis for VLSI design*. University of California, Berkeley, 1989.

[Shen *et al.*, 2008] Haihua Shen, Wenli Wei, Yunji Chen, Bowen Chen, and Qi Guo. Coverage directed test generation: Godson experience. In *Proceedings of Asian Test Symposium*, pages 321–326, 2008.

[Ustun *et al.*, 2020] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. Accurate operation delay prediction for FPGA HLS using graph neural networks. In *Proceedings of International Conference on Computer-Aided Design*, pages 1–9. Association for Computing Machinery, November 2020.

[Von Neumann, 1993] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[Wang and Schafer, 2020] Zi Wang and Benjamin Carrion Schafer. Machine Leaming to Set Meta-Heuristic Specific Parameters for High-Level Synthesis Design Space Exploration. In *Proceedings of Design Automation Conference*, pages 1–6, July 2020.

[Weicker, 1984] Reinhold P Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[Yang *et al.*, 2022] Chenghao Yang, Yinshui Xia, Zhufei Chu, and Xiaojing Zha. Logic synthesis optimization sequence tuning using rl-based lstm and graph isomorphism network. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69(8):3600–3604, 2022.

[Yim *et al.*, 1997] Joon-Seo Yim, Chang-Jae Park, In-Cheol Park, and Chong-Min Kyung. Design verification of complex microprocessors. *Journal of Circuits, Systems, and Computers*, 7(04):301–318, 1997.