

# Machine Learning for Integrated Circuit Design

**Anonymous Author<sup>1</sup>**

52102191XXXX

**Anonymous Author<sup>1</sup>**

52102191XXXX

**Anonymous Author<sup>1</sup>**

52102191XXXX

## Abstract

In this project, our group explore the application of machine learning techniques to integrated circuit design, focusing on three main areas: logic synthesis evaluation prediction, logic synthesis decision-making, and high-level synthesis with large language models. By leveraging graph neural networks, heuristic search algorithms, and both commercial and fine-tuned local language models, we aim to improve the efficiency and accuracy of translating high-level descriptions into hardware implementations, especailly focused on evaluating and optimizing middle representation AIG. The experiments demonstrate the potential of these methods in optimizing logic circuits and generating hardware description language code, contributing to advancements in automated IC design. Our code is available at <https://github.com/UNikeEN/CS3308-ML-Project>.

**Keywords** Integrated Circuit Design, Logic Synthesis, Graph Neural Networks, Heuristic Search, LLM

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Preliminaries</b>	<b>3</b>
2.1	And-Inverter Graph . . . . .	3
2.2	Graph Neural Networks . . . . .	4
<b>3</b>	<b>Part 1: Logic Synthesis Evaluation Prediction</b>	<b>5</b>
3.1	Methods . . . . .	5
3.1.1	Network Structure . . . . .	5
3.2	Experiments and Results . . . . .	5

---

<sup>1</sup>All authors contributed equally to this work.

<b>4</b>	<b>Part 2: Logic Synthesis Decision</b>	<b>7</b>
4.1	Methods . . . . .	7
4.1.1	DFS & BFS . . . . .	7
4.1.2	A* Search . . . . .	7
4.1.3	Monte Carlo Tree Search . . . . .	8
4.2	Experiments and Results . . . . .	9
4.2.1	Model Training . . . . .	9
4.2.2	Path Decision . . . . .	10
<b>5</b>	<b>Part 3: High Level Synthesis with LLM</b>	<b>11</b>
5.1	Methods . . . . .	11
5.1.1	Commercial LLM with In-context Learning . . . . .	11
5.1.2	Local Small-scale Language Model with Fine-tuning . . . . .	12
5.2	Experiments and Results . . . . .	12
5.2.1	Commercial LLM For Code Generation . . . . .	12
5.2.2	Fine-tuning GPT-2 For Code Generation . . . . .	13
<b>6</b>	<b>Conclusion and Discussion</b>	<b>16</b>
<b>A</b>	<b>Supplementary Materials</b>	<b>18</b>
A.1	Screenshot of Running Result . . . . .	18
A.2	Code Generation of GPT-2 . . . . .	18
A.3	Code Generation of LLMs . . . . .	19

# 1 Introduction

In integrated circuit design, logic synthesis and high-level synthesis are critical steps that directly impact chip performance and manufacturing costs. High-level synthesis translates natural language or C/C++ code into hardware description languages (HDL), while logic synthesis converts HDL into And-Inverter Graphs (AIG) and further into logical representations. These tasks typically demand substantial effort and extensive experience from developers. However, the rapid advancements in machine learning and large language models in recent years have provided new approaches for integrated circuit design [1] [2].

This project is divided into three parts:

- **Part 1:** Logic Synthesis Evaluation Prediction. We aim to utilize graph neural networks to build a model that predicts the evaluation of an AIG based on the AIG generated from logic synthesis.
- **Part 2:** Logic Synthesis Decision. It involves training a prediction model to forecast the future expected rewards of the current AIG state and using search algorithms to identify the optimal action sequence for optimizing the AIG.
- **Part 3:** High-Level Synthesis, which explores the use of large language models to automatically generate Verilog code from natural language descriptions.

## 2 Preliminaries

### 2.1 And-Inverter Graph

The And-Inverter Graph (AIG) is an important logical representation in integrated circuit design. It is derived from hardware description languages (HDL) and can be converted into more fundamental logical representations and implementable circuit structures. An AIG is a type of logic graph where nodes represent AND gates and inverters (NOT gates). According to relevant theorems in logic analysis, any combinational logic circuit can be converted into a form that uses only AND gates and inverters, which form a complete set of logical connectives in discrete mathematics. The edges in an AIG represent signal paths.

Once an AIG is obtained, various transformations can be applied to optimize its representation. Techniques such as rewrites and refactoring can be used to reduce the number of logic gates and the lengths of signal paths within the AIG. These optimizations help minimize the final circuit's area and power consumption while improving its operational speed. After logical optimization is completed, technology mapping is performed to convert the AIG into a final netlist that corresponds to a specific manufacturing process (e.g., a 7nm technology library), ready for actual chip fabrication.

In our project, we utilize open-source framework Yosys(<https://github.com/YosysHQ/yosys>) to process and transition the states of the AIGs in our dataset. Additionally, we use the abc\_py([https://github.com/abc0/abc\\_py](https://github.com/abc0/abc_py))

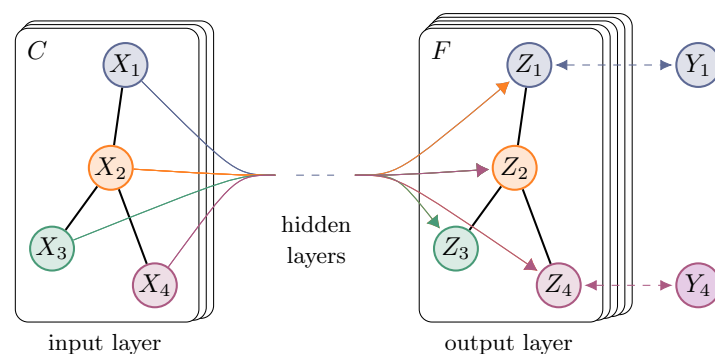
`//github.com/krzhu/abc_py`) library to convert the AIGs into a general graph form (nodes, features, and edges), which facilitates their input into subsequent algorithms.

## 2.2 Graph Neural Networks

Graph Neural Networks (GNNs) are a class of neural network models specifically designed to handle graph-structured data. In many practical applications, data naturally exists in the form of graphs, such as user relationship graphs in social networks, compound structure graphs in biochemistry, and entity relationship graphs in knowledge graphs. Traditional neural networks struggle to directly process such graph-structured data, whereas GNNs are designed to capture the complex relationships between nodes and edges within a graph.

The core idea of GNNs is to update the representation (embedding) of each node through information propagation from its neighbors. This process typically involves multiple iterative steps, with each step comprising the following two main operations:

- **Message Aggregation:** Each node collects information from its neighboring nodes. The specific aggregation method can be simple summation, averaging, or more complex aggregation functions. Through aggregation, nodes can integrate information from their neighbors, thereby enriching their own feature representations.
  - **Node Update:** After aggregating the neighbor information, each node uses a neural network (such as a multi-layer perceptron, MLP) to update its feature representation. This update process typically combines the node's own features with the aggregated neighbor information.
- By iterating the above process multiple times, the representation of each node gradually incorporates information from more distant nodes, thereby capturing more global graph structural features.



**Fig. 1.** Graph Convolutional Network [3]

One common model of GNNs is the Graph Convolutional Network (GCN), shown as Fig. 1, which uses convolution operations to aggregate information from neighboring nodes. The core

formula of GCN is:

$$H^{(l+1)} = \sigma \left( \tilde{D}^{-1/2} \tilde{A} \tilde{D}^{-1/2} H^{(l)} W^{(l)} \right) \quad (1)$$

where  $\tilde{A}$  is the adjacency matrix with added self-loops;  $\tilde{D}$  is the degree matrix;  $H^{(l)}$  is the node feature matrix at layer  $l$ ;  $W^{(l)}$  is the weight matrix at layer  $l$ ; and  $\sigma$  is the activation function.

### 3 Part 1: Logic Synthesis Evaluation Prediction

#### 3.1 Methods

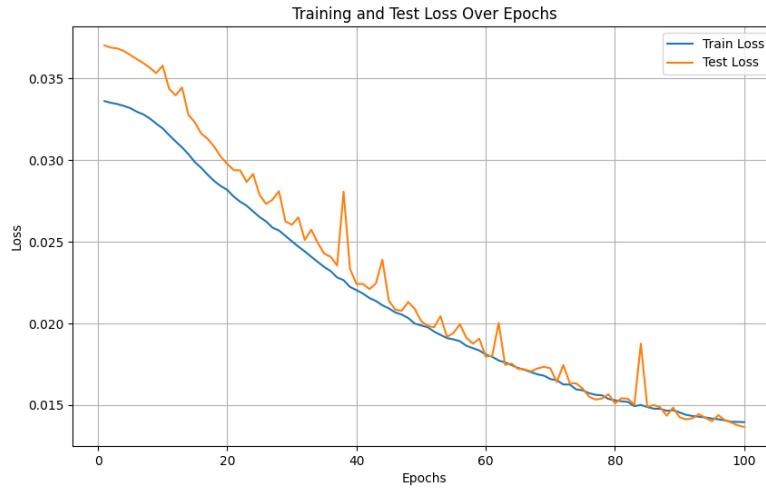
##### 3.1.1 Network Structure

In this project, the AIG dataset is represented as a generic graph structure. The `abc_py` library converts the AIG into a sequence of nodes and edges, where each node is characterized by its node type(`node_type`) and the number of inverted predecessors(`num_inverted_predecessors`). These attributes are converted into torch tensors and stacked to form the node features, resulting in a feature dimension of 2.

The primary task is a graph regression problem. The designed Graph Neural Network (GNN) consists of two main parts. The first part includes two graph convolution layers, which extract structural information from the graph. These layers transform the input feature dimension from 2 to a specified hidden dimension. The graph features processed by the convolution layers undergo global mean pooling, aggregating the node features of each graph into a fixed-length vector. The second part consists of two fully connected layers, which further refine these features. The final output is a single scalar value. ReLU activation functions are used between each layer to introduce non-linearity.

#### 3.2 Experiments and Results

We implemented the neural network model using `PyTorch` and incorporated the GCN module via the `torch_geometric` extension. In our experiments, we randomly selected 2000 PKL files for model training and testing, with the dataset split into 90% training and 10% testing sets. The hidden layer dimension was set to 256, and the batch size was set to 8. We used `MSELoss`, a commonly used loss function for univariate regression, and the Adam optimizer. The learning rate was initially set to  $3e-5$  for training the first 100 epochs, and then adjusted to  $3e-6$  for training the subsequent 100 epochs. The loss function curves for the training and testing sets over the first 100 epochs are shown as Fig. 2.



**Fig. 2.** Loss vs. Epochs

We tested two training methods with the same randomly selected dataset size: one using only the final state of each data sequence and the other using all states of each data sequence. For both methods, the number of epochs was set to 100, and the learning rate was set to  $1e-4$ . A state sequence not included in the training set was randomly selected for testing. The model was used to predict each state in the sequence, and the MAPE (mean absolute percentage error) between the predicted results and the true results was calculated, as shown in the Fig. 3. The average MAPE results on a larger-scale random sampling test dataset are shown in the Table 1.

```
Test state: max_
Predicted and true value: 0.21436017751693726, 0.25358468294143677, MAPE: 15.47%
Test state: max_0
Predicted and true value: 0.21436017751693726, 0.25358468294143677, MAPE: 15.47%
Test state: max_01
Predicted and true value: 0.21396169066429138, 0.23966245353221893, MAPE: 10.72%
Test state: max_011
Predicted and true value: 0.20210132002830505, 0.208133265376091, MAPE: 2.90%
Test state: max_0111
Predicted and true value: 0.19605305790901184, 0.21386702358722687, MAPE: 8.33%
Test state: max_01112
Predicted and true value: 0.19571831822395325, 0.2176172286272049, MAPE: 10.06%
Test state: max_011121
Predicted and true value: 0.20356342196464539, 0.2173653542995453, MAPE: 6.35%
Test state: max_0111213
Predicted and true value: 0.1307138806210327, 0.23544159531593323, MAPE: 44.48%
Test state: max_01112136
Predicted and true value: 0.13057631254196167, 0.18729206919670105, MAPE: 30.28%
Test state: max_011121364
Predicted and true value: 0.13057631254196167, 0.18729206919670105, MAPE: 30.28%
Test state: max_0111213646
Predicted and true value: 0.13223198056221008, 0.1859588623046875, MAPE: 28.89%
```

(a) with Only the Last State

```
Test state: max_
Predicted and true value: 0.21273183822631836, 0.25358468294143677, MAPE: 16.11%
Test state: max_0
Predicted and true value: 0.21273183822631836, 0.25358468294143677, MAPE: 16.11%
Test state: max_01
Predicted and true value: 0.2006916105747223, 0.23966245353221893, MAPE: 16.26%
Test state: max_011
Predicted and true value: 0.19518190622329712, 0.208133265376091, MAPE: 6.22%
Test state: max_0111
Predicted and true value: 0.19711187481880188, 0.21386702358722687, MAPE: 7.83%
Test state: max_01112
Predicted and true value: 0.19674727320671082, 0.2176172286272049, MAPE: 9.59%
Test state: max_011121
Predicted and true value: 0.1919599175453186, 0.2173653542995453, MAPE: 11.69%
Test state: max_0111213
Predicted and true value: 0.18019318580627441, 0.23544159531593323, MAPE: 23.47%
Test state: max_01112136
Predicted and true value: 0.19280999898910522, 0.18729206919670105, MAPE: 2.95%
Test state: max_011121364
Predicted and true value: 0.19280999898910522, 0.18729206919670105, MAPE: 2.95%
Test state: max_0111213646
Predicted and true value: 0.1928883194923401, 0.1859588623046875, MAPE: 3.73%
```

(b) with Entire State

**Fig. 3.** Example Test Result of Task 1

Table 1: Average MAPE Result of Task 1

Train with	Last State	All State
MAPE(All State,%)	20.96	11.25
MAPE(Last State,%)	22.41	7.33

It can be observed that the model trained using the entire state sequence outperformed the

model trained using only the final state for most individual state prediction. This difference was particularly pronounced in the prediction of the final state. We speculate that this is because training with the entire optimization sequence allows the training data to encompass the full optimization process of the AIG, thus more closely reflecting the true data distribution. Our graph neural network is sensitive to capturing the relationship between single-step optimization-induced differences in AIG and score changes. However, even after the model shows overall convergence, there is still some prediction error, which can be attributed to the relatively small and insufficient randomly selected dataset.

## 4 Part 2: Logic Synthesis Decision

### 4.1 Methods

#### 4.1.1 DFS & BFS

At the beginning, we try DFS and BFS to find the best path. However, DFS and BFS are uninformed search algorithms, which means they do not use any domain-specific knowledge or heuristics to guide the search. Additionally, because the only known information are the expected future reward and the the evaluation of current AIG, we can't define the cost of two adjacent node, which makes the DFS and BFS methods impossible. Therefore, we abandoned these methods and chose methods that can use the information we already have.

#### 4.1.2 A\* Search

A\* search algorithm is a heuristic search algorithm. It combines the strengths of Dijkstra's Algorithm and Greedy Best-First-Search by using a heuristic to guide the search. A\* evaluates nodes by combining the cost to reach the node (known as the g-cost) and an estimated cost from the node to the goal (known as the h-cost). This combined cost (f-cost = g-cost + h-cost) helps the algorithm efficiently find the shortest path to the goal by prioritizing nodes with the lowest f-cost. The heuristic used must be admissible, meaning it never overestimates the actual cost to reach the goal, ensuring the optimality of the solution.

In our project, the original A\* algorithm cannot be directly used for path decision-making, so we made some modifications to it. We set the g-eval to the actual evaluation value of the current state (obtained using `Yosys`) and the h-eval to the expected reward value of the current state (obtained through model output), prioritizing future states with the highest sum of these two values (f-eval). Obviously, such a setting of h-eval does not always satisfy the condition of overestimating the actual evaluation value, thus the A\* algorithm in our project cannot guarantee finding the optimal path, but rather a relatively good path.

### 4.1.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [4] is a search algorithm used for decision-making processes within domains that require a sequence of decisions, such as logic synthesis [5]. MCTS is instrumental in generating multiple decision sequences and then using statistical analysis to select the best one as the final result. The strength of MCTS lies in its use of both precision and randomness.

MCTS involves four main steps:

1. **Selection:** Starting at the root node, the algorithm traverses the tree based on a policy that balances exploration and exploitation until it reaches a leaf node.
2. **Expansion:** Unless the leaf node is a terminal node, the tree is expanded by adding one or more child nodes.
3. **Simulation:** From the new nodes, the algorithm simulates random play-outs to the end (or a predefined depth) to estimate their potential for success.
4. **Backpropagation:** The results of the simulation are then propagated back through the tree to update the statistics of the nodes visited during the selection phase.

To implement MCTS, we first need to create a *Node* class representing each node in the decision tree. It has multiple attributes:

- **state:** Current path;
- **children:** A list to store children of the node;
- **visits:** Node's visited times;
- **value:** Node's value;
- **depth:** The depth of the node in decision tree.

Then we need to implement the four procedures. Expansion and backpropagation are similar to the normal MCTS, so we will focus on the remaining two steps. In simulation, similar to A\* search, we set the reward to the combination of the cost to reach the node and an estimated cost from the node to the goal. To select the best node, we set up the evaluation function:

$$\text{Evaluation} = \frac{\text{value}_c}{\text{visit}_c} + c \sqrt{\frac{\ln \text{visit}_p}{\text{visit}_c}} \quad (2)$$

At the beginning, we attempted to return the best path when the search depth reached 10. However, due to each node in the decision tree having 7 children, the tree was enormous, and running one file required over 10 hours. Therefore, we decided to set a maximum number of iterations, returning the best state at the time when it reached 1,000 iterations, which greatly reduced the runtime. The accompanying issue was that the executed actions did not reach 10, which we considered acceptable.



## 4.2 Experiments and Results

### 4.2.1 Model Training

In task 2, we need to train a model to predict the future expected rewards for each state. Similar to task 1, due to the large size of the complete dataset, we randomly selected 2000 pkl files for training and testing the model, with the dataset split into 80% training and 20% testing sets. After training for 400 epochs with a learning rate of  $1e-4$ , the loss function curves for the training and testing sets over the 400 epochs are shown in Fig. 4.



**Fig. 4.** Loss vs. Epochs

Then we randomly choose a PKL file in the train folder for model testing, the test results are shown in the Fig. 5. It can be observed that the results are not much good. We think that one reason is that we only selected a very small part of the complete data set, so the information learned by the model is naturally insufficient. Another reason is attributed to the smaller target value of the dataset in task 2. Consequently, the dataset is more sensitive to parameter changes, making it difficult to train a model that predicts the future expected rewards precisely.

```

Test state: i8_
Predicted and true value: 0.6880858540534973, 0.8451433181762695, MAPE: 18.58%
Test state: i8_4
Predicted and true value: 0.7073859572410583, 0.8451433181762695, MAPE: 16.30%
Test state: i8_44
Predicted and true value: 0.7076323628425598, 0.8451433181762695, MAPE: 16.27%
Test state: i8_446
Predicted and true value: 0.6454363465309143, 0.8451433181762695, MAPE: 23.63%
Test state: i8_4464
Predicted and true value: 0.6186902523040771, 0.8451433181762695, MAPE: 26.79%
Test state: i8_44645
Predicted and true value: 0.6186902523040771, 0.8451433181762695, MAPE: 26.79%
Test state: i8_446454
Predicted and true value: 0.6186902523040771, 0.8451433181762695, MAPE: 26.79%
Test state: i8_4464545
Predicted and true value: 0.6186902523040771, 0.8451433181762695, MAPE: 26.79%
Test state: i8_44645451
Predicted and true value: 0.025019511580467224, 0.26553231477737427, MAPE: 90.58%
Test state: i8_446454515
Predicted and true value: 0.028212614357471466, -0.011361689306795597, MAPE: 348.31%
Test state: i8_4464545156
Predicted and true value: 5.7056546211242676e-05, 0.0, MAPE: inf%

```

**Fig. 5.** Example Test Result of the Model Prediction

#### 4.2.2 Path Decision

We applied A\* search and Monte Carlo Tree Search (MCTS) for path decision-making. To evaluate the performance of the methods, we tested and compared the performance of the new path with that of the traditional classic path, taking their relative error as the evaluation metric. Using both methods to search the same initial AIG, the search results of them are shown in Table 2. It can be observed that there is a significant difference between the paths found by these two methods. We attribute this mainly to poor predictive performance of the model using for future expected rewards evaluation, thereby affecting the overall search performance. Additionally, deficiencies within the search algorithms themselves may also contribute to the decline in search performance.

Table 2: Example Path Decision of Two Methods

Method	Initial AIG	Decision	Evaluation	Time
A*	prom1	3136602400	0.0936	~15min
	i7	2136361200	0.3411	
	apex4	3624644444	0.0659	
MCTS	i7	6125361200	0.0895	~10h

Additionally, to test our methods' performance, we apply these methods to the test set of the initial AIG, compare the generated decision paths with those created by classic methods, and calculate the relative error between the results. The result is shown in Table. 3 and Fig. 6. It can be observed that for most AIGs, the relative error of MCTS is around 0.3, indicating it has a good performance. However, there are also some outliers. We believe this is because MCTS limits the maximum number of iterations and cannot completely search all paths. Additionally, this can be attributed to the inaccuracies in model estimation and poor generalization, which is demonstrated before. On the contrary, A\*'s performance is so good: the worst case is around 0.5. We believe this is because A\* takes the maximum number of actions to optimize AIG and it considered both the

current evaluation value and the expected reward.

Table 3: Test Evaluation of Two Methods

	<b>c7552</b>	<b>apex1</b>	<b>voter</b>	<b>m4</b>	<b>div</b>	<b>router</b>	<b>bar</b>	<b>apex2</b>	<b>square</b>	<b>i9</b>
<b>A*</b>	0.0068	0.0694	0.2512	0.0314	0.1769	0.1698	0.2148	0.0028	-0.1182	0.5473
<b>MCTS</b>	-0.2587	-0.0545	-0.5837	-0.2312	0.0206	0.2148	-0.1587	0.0887	-0.3110	0.1679
	<b>cavlc</b>	<b>max1024</b>	<b>b9</b>	<b>pair</b>	<b>prom1</b>	<b>alu4</b>	<b>apex4</b>	<b>c880</b>	<b>sqrt</b>	<b>mem_ctrl</b>
<b>A*</b>	0.0950	0.1408	0.0966	0.0364	0.0936	0.1540	0.0659	0.1215	0.0433	0.1762
<b>MCTS</b>	0.3281	-0.4012	-0.3515	-0.0516	-0.5621	-2.2856	-0.1778	-0.1746	-0.4097	0.1426

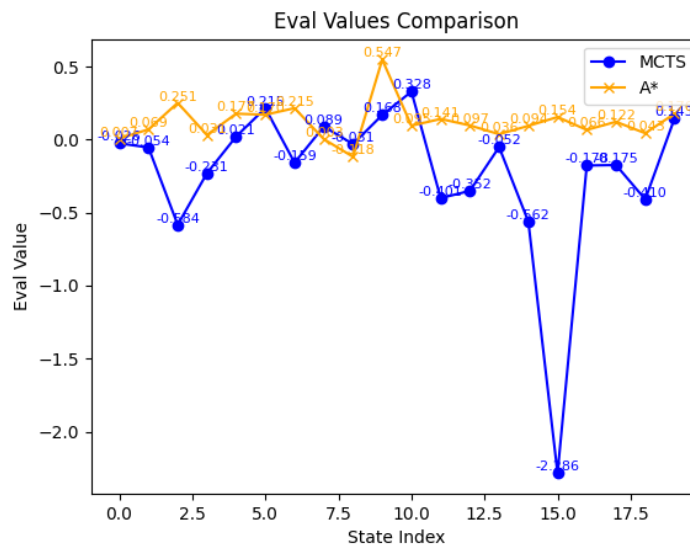


Fig. 6. Test Result of Two Methods

## 5 Part 3: High Level Synthesis with LLM

### 5.1 Methods

#### 5.1.1 Commercial LLM with In-context Learning

Code generation models are a significant research direction in the fields of software engineering and artificial intelligence. They can greatly enhance development efficiency, reduce repetitive tasks, and have immense potential in areas such as automated testing and code review. Traditional code generation models, which adopt Sequence-to-Sequence and Encoder-Decoder structures, face challenges related to the quality of generated code and the generalization ability required for different programming language syntax.

In recent years, commercial general-purpose large language model(LLMs) such as OpenAI's ChatGPT, GPT-4, Anthropic's Claude, and Baidu's Wenxin Yiyan have provided new approaches

for code generation task [6] [7]. These large models, trained on vast amounts of data from various programming languages, can understand and generate high-quality corresponding code based on code snippets or natural language descriptions.

In-context learning is an important technique in LLMs, allowing them to learn and generate content based on context without parameter adjustments. Specifically, the model understands task requirements by providing examples (i.e., context) and generates corresponding outputs based on these examples. For code generation tasks, developers can initially provide the model with some code examples and requirement descriptions, enabling the model to generate new code that meets subsequent needs and specific formats more accurately.

### 5.1.2 Local Small-scale Language Model with Fine-tuning

Commercial large language models typically run on servers, but an alternative approach is to fine-tune smaller-scale general language models locally on the client side. This ensures the privacy of the training data and provides stable response times. GPT-2, an early general language model based on the Transformer architecture released by OpenAI, has a relatively small number of parameters and weaker code generation capabilities. Our team experimented with fine-tuning GPT-2 [8] using a self-collected dataset of Verilog code.

## 5.2 Experiments and Results

### 5.2.1 Commercial LLM For Code Generation

We tried three online commercial LLMs: ChatGPT(GPT-3.5), GPT-4, and WenXinYiYan and one locally deployed commercial LLMs: Internlm2. We had them each attempt to generate Verilog implementations for two tasks: an **8-bit Adder** and a **Selection Sort** algorithm. The 8-bit Adder task is more aligned with traditional applications of hardware description languages, while the Selection Sort algorithm task represents a more innovative attempt. We provided the same prompts to all four models and observed their Verilog code outputs. For each task, due to the limitation of locally deployed LLMs, we just input all three Verilog code outputs back into the three online LLMs and let them to evaluate the strengths and weaknesses of each implementation. Finally, we request each LLM to provide its own ranking of the implementations.

In the task of the 8-bit adder, the Verilog code implementation by GPT-4 received the best overall evaluation from the three LLMs. None of the implementations from the three LLMs had logical or functional errors. The code structures generated by ChatGPT and GPT-4 were quite similar, but the code produced by GPT-4 was more standardized. A major difference between implementation of WenXinYiYan and those of the former two is that it directly utilized the built-in addition operation syntax for implementing full-adders and half-adders, rather than the lower-level AND, OR, and XOR operations. This difference was considered an advantage in LLMs' evaluation, making the code more concise. However, we think that ChatGPT's approach aligns better with hardware logic, making it more suitable for this task.

In the task of Selection Sort, the implementation by GPT-4 received the best overall evaluation from the LLMs once again. The code generated by ChatGPT performed the worst, encountering syntax errors (attempting to modify the input array within the function). The implementation by WenXinYiYan was correct, but it merely “translate” the selection sort algorithm into Verilog syntax. Conversely, the implementation by GPT-4 effectively utilized hardware logic, incorporating clock cycles into the code and using non-blocking assignments ( $\leq$ ) as much as possible.

After conversing with different commercial LLMs, we discovered significant differences in their performance and focus capabilities. Now we will discuss the performance differences of ChatGPT, GPT-4, WenXinYiYan-3.5 and Internlm2 from several aspects.

- **Generation Speed:** When using GPT-4, we can clearly feel that its response and text generation speed is slower than ChatGPT. This is because the former has a much larger number of parameters, so the inference speed will be relatively slower without significant changes in architecture. WenXinYiYan has the slowest generation speed, which may be due to its model architecture. Internlm2 has the fastest generation speed, which is because of its especially small parameter size.
- **Code Generation Quality:** In this task of Verilog code generation, GPT-4 demonstrated absolute superiority. Its generated code achieved the best scores in both correctness and standardization. WenXinYiYan ranked second, with generated code ensuring correctness but lacking sufficient standardization. Next is ChatGPT, whose generated code has no fundamental logic issues but may contain some syntax errors. Last is Internlm2, whose generated code still has issues with code duplication and incorrect module nesting compared to ChatGPT.
- **Associative Ability:** In this aspect, we find GPT-4 performed the best, as it can describe its generated code in great detail. Moreover, during code evaluation, it can provide thoughtful suggestions for code improvement, which really surprised us. WenXinYiYan-3.5 also demonstrates associative abilities close to GPT-4 to some extent but lacks the same level of detail. Compared to the former two, ChatGPT and Internlm2 show a significant weakness, essentially just completing tasks as instructed.

The code generation result of 8-bit adder task and these four LLMs are shown in supplementary materials Sec. A.3. The complete conversation logs are saved in the attached file.

### 5.2.2 Fine-tuning GPT-2 For Code Generation

Our team also experimented with locally fine-tuning a smaller-scale general language model, GPT-2 (specifically, `GPT2LMHeadModel`, which adds a linear layer in front of the GPT-2 model to store weights for the input matrix), for Verilog code generation. Our dataset was sourced from various GitHub repositories containing basic Verilog block code. Each file was split into multiple data entries based on a predefined block size. The training and testing data were divided in a 70%:30% ratio.

The training process utilized the **Trainer** from the **transformers** library. We employed the AdamW optimizer, an improved version of Adam that includes weight decay for efficient large-scale data processing and overfitting prevention, commonly used in natural language processing models. The remaining training parameters are as follows:

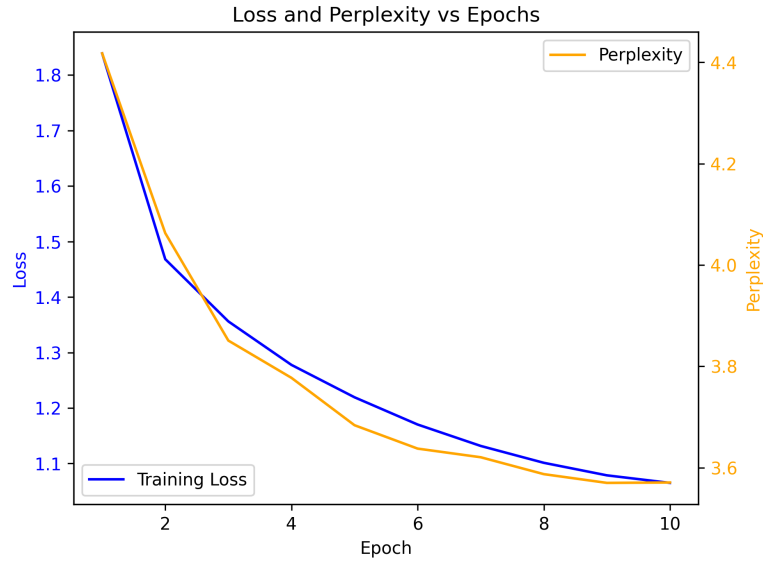
- **num\_train\_epochs**: Number of training epochs, set to 10
- **per\_device\_train\_batch\_size**: Batch size, set to 16
- **learning\_rate**: Learning rate, set to 5e-5
- **warmup\_steps**: Number of warmup steps, set to 500
- **lr\_scheduler\_type**: Learning rate scheduler type, set to **linear**

Combining the warmup process and learning rate scheduling, the learning rate linearly increases from zero to the initial value at the start of training to prevent large gradient updates and drastic changes in model parameters. It then linearly decreases to avoid gradient oscillations that prevent finding the optimal solution. Additionally, we introduced **TrainLossCallback** to automatically record results, evaluate, and plot curves for each epoch. The experimental platform was a single NVIDIA RTX 4070 GPU.

We compared the performance of the model in generating Verilog code before and after fine-tuning through both quantitative and qualitative evaluations. The quantitative metric used is perplexity, which is one of the commonly used evaluation metrics in natural language processing, particularly for assessing the performance of language models. Perplexity measures the prediction accuracy of a language model on a given dataset and can be interpreted as the average uncertainty of the language model on the dataset. Specifically, perplexity indicates how perplexed the language model is when predicting the next word; a lower value indicates more accurate predictions. Mathematically, perplexity is defined as:

$$\text{Perplexity} = \exp \left( -\frac{1}{N} \sum_{i=1}^N \log P(w_i | w_{1:i-1}) \right) \quad (3)$$

where  $P(w_i | w_{1:i-1})$  is the model's predicted probability of the  $i$ -th word, and  $N$  is the total number of words in the dataset. Perplexity is particularly important in evaluating large language models because it provides an intuitive metric for assessing the model's generative capability on unseen data. The curves showing the changes in test set perplexity and training set error as the training progresses are as follows:



**Fig. 7.** Loss and Perplexity vs. Epochs

The results of the manual question-and-answer evaluation can be found in the supplementary materials Sec. A.2. It can be observed that the fine-tuned GPT-2 generates more contextually rich content and adheres more closely to Verilog syntax compared to the pre-trained model. The output from the original model either does not contain code or consists of meaningless symbols. The fine-tuned model is capable of generating “docstrings”, but the code itself remains unusable. This is likely because the early GPT-2 model, with only 117 million parameters, is insufficient for the task of code generation. However, the exploration of fine-tuning remains valuable.

We also used the ROUGE metric series (ROUGE-N, ROUGE-L), commonly employed in natural language processing to evaluate machine translation and content generation, to assess the performance of the model before and after fine-tuning. ROUGE compares the similarity between generated text and reference text using methods like n-grams and the longest common subsequence (LCS). Despite the observed improvements in manual evaluation, the fine-tuning process resulted in a decline in the model’s ROUGE scores. We hypothesize that this is because ROUGE is primarily based on natural language tasks, whereas code differs significantly from natural language. Code has strict syntax and structure, such as extensive indentation, identifiers, and frequently occurring keywords. ROUGE’s lexical matching methods may not adequately capture the syntactical and structural information inherent in code.

Table 4: ROUGE Score of GPT-2 Models

	Original	Fine-tuned(BPE Tokenizer)	Fine-tuned(Char Tokenizer)
<b>ROUGE-1</b>	0.20244	0.17115	0.12115
<b>ROUGE-2</b>	0.08127	0.07137	0.04843
<b>ROUGE-L</b>	0.14396	0.13264	0.10026

## 6 Conclusion and Discussion

In task1, we explored the use of graph neural networks (GNNs) to predict the evaluation of And-Inverter Graphs (AIGs) derived from logic synthesis. We employed a network structure consisting of graph convolution layers followed by global mean pooling and fully connected layers. Our experiments demonstrated the capability of the model to capture significant features from the AIGs, resulting in promising predictions that align with the actual outcomes when training set contains whole optimization state sequence, although the model's accuracy can be further improved with a more comprehensive dataset.

In task2, we train a model to predict future rewards and implement two search algorithms: A\* and Monte Carlo Tree Search to make a decision of action sequence. We apply these methods to the test set of initial AIG and compare the performance with classic method. A\* has better performance than MCTS and is very close to the baseline because it takes the maximum number of actions to optimize AIG and considered both the current evaluation value and the expected reward. Due to the limited computing resources, we cannot run MCTS up to maximum depth. We believe that given enough time, MCTS can achieve more than A\* performance because of its randomness characteristic.

In task3, we tried several commercial LLMs, such as ChatGPT, GPT-4, WenXinYiYan and Internlm2, letting them to generate verilog code based on specific requirements. After obtaining their respective generated results, we asked them to rank all the code implementations, observing the performance differences and focus capabilities among different commercial LLMs. In the end, GPT-4 demonstrated an excellent Verilog code generation capability far beyond other LLMs. In addition, we experimented with fine-tuning GPT-2 using a self-collected Verilog code dataset.

The integration of advanced machine learning models into logic and high-level synthesis presents a promising avenue for reducing manual effort and improving the efficiency of circuit design processes. However, the success of such integration heavily relies on the quality of the training data and the sophistication of the models. Future work should focus on expanding the datasets, incorporating more diverse synthesis scenarios, and refining model architectures to capture deeper insights into the synthesis process.

## Acknowledgments

This work was conducted as a group assignment for the course CS3308 Machine Learning at Shanghai Jiao Tong University.

Our source code is available at <https://github.com/UNikeEN/CS3308-ML-Project>.



## References

- [1] E. Afacan, N. Lourenço, R. Martins, and G. Dündar, “Machine learning techniques in analog/rf integrated circuit design, synthesis, layout, and test,” *Integration*, vol. 77, pp. 113–130, 2021.
- [2] G. Huang, J. Hu, Y. He, J. Liu, M. Ma, Z. Shen, J. Wu, Y. Xu, H. Zhang, K. Zhong, *et al.*, “Machine learning for electronic design automation: A survey,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 5, pp. 1–46, 2021.
- [3] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” 2017.
- [4] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A survey of monte carlo tree search methods,” *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.
- [5] A. B. Chowdhury, M. Romanelli, B. Tan, R. Karri, and S. Garg, “Retrieval-guided reinforcement learning for boolean circuit minimization,” *arXiv preprint arXiv:2401.12205*, 2024.
- [6] Y. Fu, Y. Zhang, Z. Yu, S. Li, Z. Ye, C. Li, C. Wan, and Y. C. Lin, “Gpt4aigchip: Towards next-generation ai accelerator design automation via large language models,” in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2023.
- [7] K. Chang, Y. Wang, H. Ren, M. Wang, S. Liang, Y. Han, H. Li, and X. Li, “Chipgpt: How far are we from natural language hardware design,” *arXiv preprint arXiv:2305.14019*, 2023.
- [8] J.-S. Lee and J. Hsiang, “Patent claim generation by fine-tuning openai gpt-2,” *World Patent Information*, vol. 62, p. 101983, 2020.

## A Supplementary Materials

### A.1 Screenshot of Running Result

Here are the screenshot of example running result of A\* and MCTS algorithm in Part 2.

```
Processing i7_
Processing i7_2
Processing i7_21
Processing i7_213
Processing i7_2136
Processing i7_21363
Processing i7_213636
Processing i7_2136361
Processing i7_21363612
Processing i7_213636120
Processing i7_2136361200
Final decision: i7_2136361200, eval: 0.3411027084118715
```

Fig. 8. Search Result of A\* Search

```
(ml_project) oni0n@DESKTOP-07K9NGH:/mnt/c/Users/thy030702/Desktop/project/src/task2$ python mcts.py --config-config.yaml --filename=i7.aig
Final decision: i7_6125361200, eval: 0.68948290930229337
```

Fig. 9. Search Result of MCTS

### A.2 Code Generation of GPT-2

User Prompt:

Write an adder in verilog code.

Response of the original GPT-2:

Write an adder in verilog code.

```
$ add_adder = new Adder ( $name, $value );
. add ('$ ', adders ); // add a new addor
, addrs, and addors
( function ( ) {
:adders ( addr, ' ' ); } )
- ( var add = $add_get_sub ( " $ ", " add " ); var sub = add. sub ( sub. name,
subr ); $sub = sub ;
and
( var $ = " Add " + $ ) ; $ sub += $ ; // Add a sub
/
)
```

Response of the fine-tuned GPT-2:

```
/**
 * Simple 4-bit Adder
 * Adds two 4-bit binary numbers
 */

module adder (
    input [3:0] a, input [3:0] b,
    output [3:0] sum,
    output carry_out
);

    wire c1, c2, c3;

    assign sum[0] = a[0] ^ b[0]; // Sum of least significant bit
    assign c1 = a[0] & b[0]; // Carry from LSB

    assign sum[1] = a[1] ^ b[1] ^ c1; // Sum of second bit
    assign c2 = (a[1] & b[1]) | (c1 & (a[1] ^ b[1])); // Carry from second bit

    sum[2] = ( a[2] ^ b[2] ^ c2; // Sum of third bit
endmodule

/*
Carry from third bit
/* Sum of most significant bit
```

### A.3 Code Generation of LLMs

Here is the code generation result of 8-bit adder task from different LLMs. The complete conversation logs are saved in the attached file.

User Prompt:

请使用verilog基于全加器和半加器编写一个8位加法器

Response of ChatGPT(GPT-3.5):

好的，下面是一个使用Verilog编写的8位加法器，其中包含全加器和半加器：

```
```verilog
```

```
module half_adder(
    input a,
    input b,
    output sum,
    output carry
);
    assign sum = a ^ b;
    assign carry = a & b;
endmodule

module full_adder(
    input a,
    input b,
    input cin,
    output sum,
    output cout
);
    wire sum1, carry1, carry2;
    half_adder HA1(.a(a), .b(b), .sum(sum1), .carry(carry1));
    half_adder HA2(.a(sum1), .b(cin), .sum(sum), .carry(carry2));
    assign cout = carry1 | carry2;
endmodule

module eight_bit_adder(
    input [7:0] A,
    input [7:0] B,
    output [7:0] Sum,
    output Cout
);
    wire [7:0] carry;
    full_adder FA0(.a(A[0]), .b(B[0]), .cin(1'b0), .sum(Sum[0]), .cout(carry[0]));
    full_adder FA1(.a(A[1]), .b(B[1]), .cin(carry[0]), .sum(Sum[1]),
        .cout(carry[1]));
    full_adder FA2(.a(A[2]), .b(B[2]), .cin(carry[1]), .sum(Sum[2]),
        .cout(carry[2]));
    full_adder FA3(.a(A[3]), .b(B[3]), .cin(carry[2]), .sum(Sum[3]),
        .cout(carry[3]));
    full_adder FA4(.a(A[4]), .b(B[4]), .cin(carry[3]), .sum(Sum[4]),
        .cout(carry[4]));
```

```

    full_adder FA5(.a(A[5]), .b(B[5]), .cin(carry[4]), .sum(Sum[5]),
        .cout(carry[5]));
    full_adder FA6(.a(A[6]), .b(B[6]), .cin(carry[5]), .sum(Sum[6]),
        .cout(carry[6]));
    full_adder FA7(.a(A[7]), .b(B[7]), .cin(carry[6]), .sum(Sum[7]), .cout(Cout));
endmodule
...

```

这个Verilog代码包括了半加器 (`half_adder`) 和全加器 (`full_adder`) 模块，然后使用这些模块构建了一个八位加法器 (`eight_bit_adder`)。这个加法器可以对两个八位二进制数进行加法运算，并产生八位的结果和一个进位输出。

Response of GPT-4:

当然，编写一个8位加法器可以通过组合全加器和半加器来实现。以下是Verilog代码的实现，其中包含全加器 (`Full Adder`) 和半加器 (`Half`

`Adder`) 的模块定义，然后使用这些模块来构建一个8位加法器。

### 半加器模块

```

```verilog
module half_adder (
    input wire a,
    input wire b,
    output wire sum,
    output wire carry
);
    assign sum = a ^ b; // Sum is the XOR of a and b
    assign carry = a & b; // Carry is the AND of a and b
endmodule
...

```

### 全加器模块

```

```verilog
module full_adder (
    input wire a,
    input wire b,
    input wire cin,

```

```
        output wire sum,
        output wire carry
    );
    wire sum1, carry1, carry2;

    // Use two half adders to create a full adder
    half_adder ha1 (
        .a(a),
        .b(b),
        .sum(sum1),
        .carry(carry1)
    );

    half_adder ha2 (
        .a(sum1),
        .b(cin),
        .sum(sum),
        .carry(carry2)
    );

    assign carry = carry1 | carry2; // Final carry out is the OR of the two carry outs

endmodule
```

### 8位加法器模块

```verilog
module adder_8bit (
    input wire [7:0] a,
    input wire [7:0] b,
    input wire cin,
    output wire [7:0] sum,
    output wire cout
);
    wire [6:0] carry; // Internal carry signals

    // Instantiate 8 full adders

```

```
full_adder fa0 (  
    .a(a[0]),  
    .b(b[0]),  
    .cin(cin),  
    .sum(sum[0]),  
    .carry(carry[0])  
);  
  
full_adder fa1 (  
    .a(a[1]),  
    .b(b[1]),  
    .cin(carry[0]),  
    .sum(sum[1]),  
    .carry(carry[1])  
);  
  
full_adder fa2 (  
    .a(a[2]),  
    .b(b[2]),  
    .cin(carry[1]),  
    .sum(sum[2]),  
    .carry(carry[2])  
);  
  
full_adder fa3 (  
    .a(a[3]),  
    .b(b[3]),  
    .cin(carry[2]),  
    .sum(sum[3]),  
    .carry(carry[3])  
);  
  
full_adder fa4 (  
    .a(a[4]),  
    .b(b[4]),  
    .cin(carry[3]),  
    .sum(sum[4]),  
    .carry(carry[4])  
);
```

```
full_adder fa5 (  
    .a(a[5]),  
    .b(b[5]),  
    .cin(carry[4]),  
    .sum(sum[5]),  
    .carry(carry[5])  
);
```

```
full_adder fa6 (  
    .a(a[6]),  
    .b(b[6]),  
    .cin(carry[5]),  
    .sum(sum[6]),  
    .carry(carry[6])  
);
```

```
full_adder fa7 (  
    .a(a[7]),  
    .b(b[7]),  
    .cin(carry[6]),  
    .sum(sum[7]),  
    .carry(cout)  
);
```

```
endmodule
```

```
...
```

这段代码定义了一个8位加法器 ``adder_8bit``，它利用了全加器模块 ``full_adder`` 和半加器模块

``half_adder``。每个全加器负责处理一位的加法，并传递进位给下一个全加器，从而完成8位二进制数的加法运算。

---

#### Response of WenXinYiYan-3.5:

在Verilog中，一个8位加法器可以通过级联多个全加器（Full Adders，

FA）来实现。全加器可以视为由半加器（Half Adder，

HA）和进位处理逻辑组成。但是，在实际设计中，我们通常直接使用现成的全加器模块，因为全加器可以直接处理两个加数和来自低位的进位。



以下是一个基于全加器实现8位加法器的Verilog代码示例：

```
``verilog
module half_adder(
    input a,
    input b,
    output sum,
    output carry
);

assign {carry, sum} = a + b; // 隐含的位扩展和截断

endmodule

// 注意：在实际应用中，我们不会真的这样实现全加器，因为上面的half_adder并不能处理进位
// 下面的full_adder是一个完整的全加器实现
module full_adder(
    input a,
    input b,
    input cin,
    output sum,
    output cout
);

assign {cout, sum} = a + b + cin; // 隐含的位扩展和截断

endmodule

// 8位加法器，使用全加器级联实现
module eight_bit_adder(
    input [7:0] a,
    input [7:0] b,
    output [7:0] sum,
    output cout
);

wire [7:0] c_in; // 进位输入信号，从最低位开始
assign c_in[0] = 0; // 最低位的进位输入为0
```

```

full_adder fa0(a[0], b[0], c_in[0], sum[0], c_in[1]); // 实例化第一个全加器
full_adder fa1(a[1], b[1], c_in[1], sum[1], c_in[2]);
full_adder fa2(a[2], b[2], c_in[2], sum[2], c_in[3]);
full_adder fa3(a[3], b[3], c_in[3], sum[3], c_in[4]);
full_adder fa4(a[4], b[4], c_in[4], sum[4], c_in[5]);
full_adder fa5(a[5], b[5], c_in[5], sum[5], c_in[6]);
full_adder fa6(a[6], b[6], c_in[6], sum[6], c_in[7]);
full_adder fa7(a[7], b[7], c_in[7], sum[7], cout); // 最高位的进位输出

```

```

endmodule

```

```

...

```

请注意，在上面的代码中，`half\_adder` 模块实际上并没有在 `eight\_bit\_adder` 中使用，因为它不足以实现一个全加器。我保留了它作为一个示例，但真正的全加器实现是在 `full\_adder` 模块中。在 `eight\_bit\_adder` 模块中，我们级联了 8 个 `full\_adder` 模块来实现 8 位加法器。最低位的进位输入 `c\_in[0]` 设置为 0，而每个全加器的进位输出都连接到下一个全加器的进位输入。最高位的进位输出 `cout` 是整个 8 位加法器的最高进位。

Response of Internlm2:

```

```verilog
module adder (
    input [7:0] a,
    input [7:0] b,
    output [7:0] sum
);

    assign sum = a + b;
endmodule
...

```

这个加法器有两个输入：`a` 和 `b`，输出为 `sum`，它将 `a` 和 `b` 相加并输出结果。

在 `adder` 模块中，我们使用了 `assign` 语句来将 `a + b` 的结果赋值给 `sum`。注意，这里使用了 `[7:0]` 来表示输入 `a` 和 `b` 的最低位，这是因为 Verilog 中的输入必须是二进制数，而 `a` 和 `b` 的最高位是 0。

您可以将这个 `adder` 模块与其它模块一起使用，例如一个 `mux`

模块来控制加法器的行为。例如，下面是一个简单的 `mux` 模块，它将加法器的输出与 `1` 进行比较，如果相等则输出 `1`，否则输出 `0`：

```
```verilog
module compare (
    input [7:0] a,
    input [7:0] b,
    output [1:0] out
);

    assign out = a == b ? 1 : 0;

endmodule
```
```

这个 `compare` 模块的输入是 `a` 和 `b`，输出为 `out`，它将比较 `a` 和 `b` 是否相等，并输出相应的值。

现在，您可以将 `adder` 模块和 `compare` 模块组合在一起，构建一个完整的8位加法器。例如：

```
```verilog
module main (
    input [7:0] a,
    input [7:0] b,
    input [1:0] cmp,
    output [7:0] sum
);

    assign sum = a + b;

    module adder (
        input [7:0] a,
        input [7:0] b,
        output [7:0] sum
    );

        assign sum = a + b;

    module compare (
        input [7:0] a,
```

```
    input [7:0] b,
    output [1:0] out
);

assign out = a == b ? 1 : 0;

module main (
    input [7:0] a,
    input [7:0] b,
    input [1:0] cmp,
    output [7:0] sum
);

assign sum = a + b;

assign out = cmp == 1 ? 1 : 0;

endmodule
`
```