CS3308 Machine Learning Final Project

Qihui Zhou   Hao Li   Haoxiang Sun

June 13, 2024

Contents

1 Task1: Logic synthesis evaluation prediction 1
  1.1 Problem Description . . . 1
  1.2 Dataset . . . 1
  1.3 Data Preprocessing . . . 1
  1.4 Model . . . 1
  1.5 Experimental Settings . . . 2
  1.6 Experimental Results . . . 2

2 Task2: Logic synthesis decision 3
  2.1 Model Training . . . 3
  2.2 Evaluation . . . 3
  2.3 Search Algorithm . . . 3

3 Task3: High level synthesis with large language model 5
  3.1 Vectorr(Vector Reverse) . . . 5
  3.2 Gatesv(Gates Vector) . . . 6
    3.2.1 Simple Description . . . 7
    3.2.2 Detailed Description . . . 8
  3.3 Code Explaining . . . 8

4 Assignment Proportion 9

1 Task1: Logic synthesis evaluation prediction

1.1 Problem Description

Logic synthesis is to translate the hardware description language (HDL) into logic representation. In logic synthesis, the HDL is first transformed into an And-Inverter Graph (AIG), which is a logic graph containing only and gate and inverter gate. After that, several transformations can be conducted to optimize the representation of AIG, and the area and the time delay of the designed chip is estimated as the evaluation of the AIG. In this task, we build a model to predict the evaluation of the AIG.

1.2 Dataset

Each file in the given dataset is a generation path of the logic synthesis, containing a sequence of input state and their corresponding evaluation values as learning targets. The sequence of input states includes the intermediate states after each action is executed, from the initial state to the final state. Specifically, the input data is a dictionary containing two keys: 'input' and 'label'. Each key's value is a list, where the 'input' list stores a sequence of states and the 'label' list stores the corresponding sequence of evaluation values for each state. Each input state is represented by a string consisting of the circuit name and the actions that have been executed. Take state 'adder_4365' as an example, 'adder' is the circuit's name(which means the initial states is 'adder_'), and the numbers denotes the action sequence, where each number represents an action.

1.3 Data Preprocessing

Since an AIG contains much more information than a state string, We first use the Yosys package to obtain the current AIG from the given state string. In order to make the AIG usable for our model, we then use the abc-py package to represent AIG through the node connectivity and the features for each node. At last, we split the data into 90% training data and 10% testing data. Due to the significant time consumption of data preprocessing, we could only afford to use 1% data in the given dataset(which still cost hours)

1.4 Model

The Graph Attention Network (GAT) integrates the self-attention mechanism into the foundational structure of graph convolutions, providing an effective method for processing graph-structured data. Introduced by Veličković et al. in 2017, its main advantage is dynamically weighting the contributions between nodes based on their features and the graph structure, significantly enhancing the model's performance in tasks like node classification and link prediction.

In GAT, the design of the graph layer strives to use a self-attention strategy to compute the hidden representations of each node https://latex.sjtu.edu.cn/project/666117f0546663080a0da6f9within the graph. This strategy allows the model to update the state of each node based on the features of its neighbors, without relying on the global structure of the graph. Specifically, each node evaluates the importance of its neighbors for its own state through a learnable parametric function. This attention weight calculation involves the features of the node itself and all its neighbors, implemented through a small, parameterized neural network.

In each layer of GAT, nodes first enhance the expression of their features through a linear transformation parameterized by a weight matrix W. Next, for each pair of nodes i and j, the model computes an attention coefficient α_{ij} that signifies the importance of node j to node i. This coefficient is computed through a linear layer with a LeakyReLU activation:

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{a}^T[\mathbf{W}h_i \| \mathbf{W}h_j]\right)\right)}{\sum_{k\in\mathcal{N}(i)} \exp\left(\text{LeakyReLU}\left(\mathbf{a}^T[\mathbf{W}h_i \| \mathbf{W}h_k]\right)\right)}$$

where $\mathbf{a}$ represents the parameters of the attention mechanism, and $\|$ denotes concatenation. The attention coefficients α_{ij} are normalized using the softmax function to ensure that the sum of the weights entering node i equals 1.

The new feature vector for each node is the weighted sum of the transformed features of its neighbors, with the weights being the aforementioned attention coefficients:

$$\mathbf{h}_i' = \sigma \left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} \mathbf{W} \mathbf{h}_j \right)$$

where σ is a nonlinear activation function, commonly the ReLU function.

GAT also incorporates multi-head attention mechanisms, where multiple independent attention mechanisms are executed in parallel in each GAT layer, and the results are then combined (usually by concatenation or averaging). This design not only enhances the model's capacity to capture complex patterns but also improves the stability and efficiency of the learning process.

In this task, we use the GATConv layer from PyTorch Geometric to design our GATNet model. We use two GATConv layers with dropout before each layer and an elu activation after the first layer. At the end of our network we use a global mean pooling layer to get the score of the whole graph. The dropout rate of each dropout function and each layer is 0.6. The first layer has 4 heads, and the second layer has only one head. The hidden dimension is 32. For more details of our model's design, please refer to our code.

1.5 Experimental Settings

We used the Adam optimizer with learning rate 0.005 and MSE loss. The batch size is 32 nd we trained the model for 50 epochs. We found more epochs may cause overfitting through experiments.

1.6 Experimental Results

We record the training loss and test loss over epochs during training, the results are shown below:

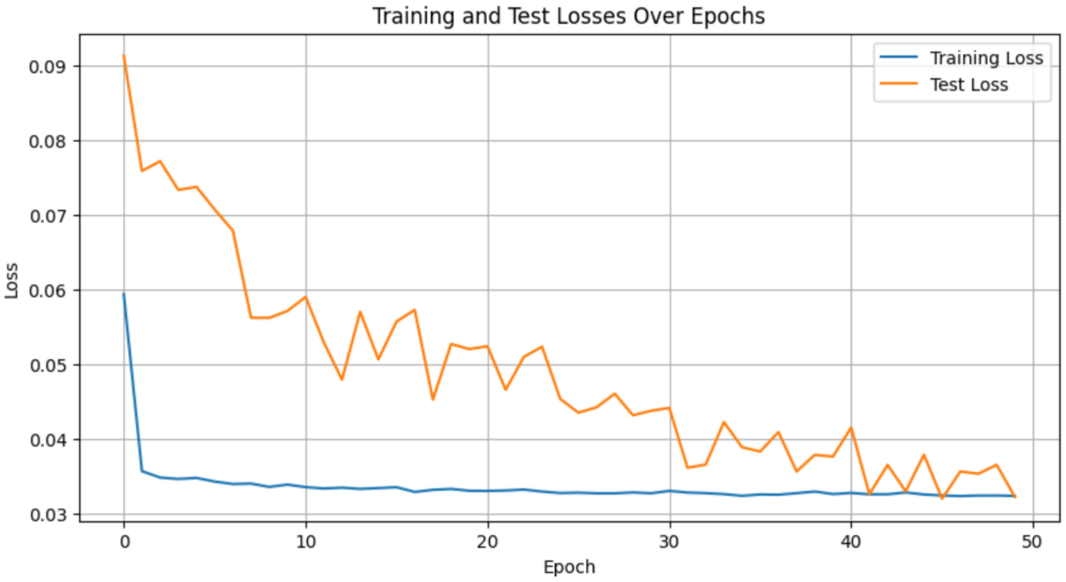

Figure 1: Training and test loss over epochs

From the results, we can see that at the very beginning the training loss decrease rapidly to a low level, which means the model can learn effective representation of the data quickly, indicating the high efficiency of the GAT model. However, The training loss decreases very slowly or remains almost constant, possibly because the task of making more precise evaluations on different circuits presents a significant challenge to the model, and there is limited training data available for each type of circuit under similar states. The test loss continues to decrease throughout the training process, indicating that the model's generalization ability is continuously improving. However, the persistent fluctuations of test loss, reflecting instability, suggest that the model is quite sensitive to the training data.

Overall, the value of the loss does not fully reflect the performance of the model because, for the evaluation of AIG, the relative magnitude of the evaluation values may be more important than the absolute numbers. This is because we need to use these evaluation values to guide the decision-making process for AIG optimization. Therefore, we will further explore this in task 2.

Algorithm 1 Greedy Algorithm for Optimizing AIG

```
Input: AIG graph $G$
Output: Optimal action list $A$
Initialize $A \leftarrow []$
for $i = 1$ to 10 do
    Generate 7 different AIG graphs $\{G_1, G_2, \ldots, G_7\}$ from $G$ using 7 different actions
    Score each $G_j$ $(j = 1, \ldots, 7)$
    Select the action $a^*$ that produces the highest scoring $G_j$
    Append $a^*$ to $A$
    Update $G \leftarrow G_j$ where $j$ corresponds to $a^*$
end for
return $A$ =0
```

2 Task2: Logic synthesis decision

2.1 Model Training

In this part of the work, we trained model two with future rewards as the prediction goal, and the model training work in this part of the work is exactly the same as the model training work in Task1, so we will not repeat it again.The loss of model training is shown in Figure 2

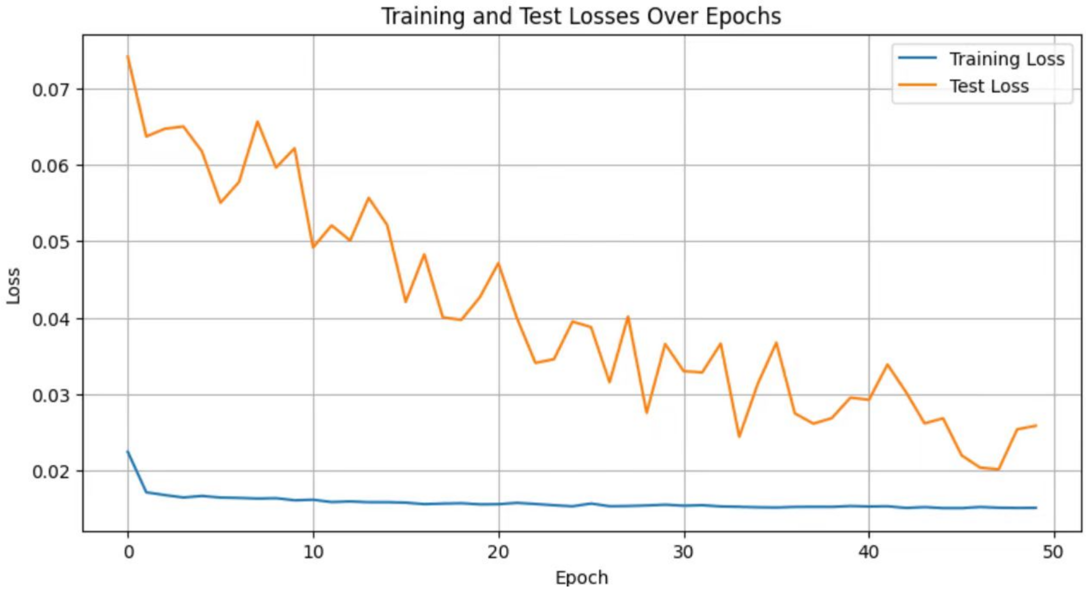

Figure 2: Training and test loss over epochs

2.2 Evaluation

We then designed the Evaluation function using the trained two models, here the score for each AIG graph is calculated using the following formula:

$$AIG\_Scores = model1\_predict * n + model2\_predict * (1 - n)$$

The introduction of the parameter n allows for a better adjustment of the weighting of the two models in evaluation.

2.3 Search Algorithm

In search algorithm we have taken greedy algorithm. The core of the algorithm is to always take the action for each current AIG graph that maximizes its next score. The pseudo-code for the algorithm to run is shown as Alg. 1.

Seventy AIG plots are generated for scoring each test AIG in the greedy algorithm. Although the effect of the future state on the current state has been considered in the model (Model II), it is clear that the effect of the future on the current has not been adequately considered from an algorithmic point of view.

Algorithm 2 N-step Greedy Algorithm for Optimizing AIG

```
**Input:** AIG graph $G$, step length $n$
**Output:** Optimal action list $A$
Initialize $A \leftarrow []$
**for** $i = 1$ to $\lfloor \frac{10}{n} \rfloor$ **do**
    Generate $7^n$ different AIG graphs $\{G_1, G_2, \ldots, G_{7^n}\}$ from $G$ using $n$-step actions
    Score each $G_j$ $(j = 1, \ldots, 7^n)$
    Select the sequence of $n$ actions $a^*$ that produces the highest scoring $G_j$
    Append $a^*$ to $A$
    Update $G \leftarrow G_j$ where $j$ corresponds to $a^*$
**end for**
**return** $A$ =0
```

Algorithm 3 Simulated Annealing Algorithm for Optimizing AIG

```
**Input:** AIG graph $G$, initial temperature $T$, cooling rate $\alpha$
**Output:** Optimal action list $A$
Initialize $A \leftarrow []$
Initialize current state $G_{current} \leftarrow G$
Initialize current score $S_{current} \leftarrow \text{Evaluate}(G_{current})$
**while** $T > 0$ **do**
    Generate 7 different AIG graphs $\{G_1, G_2, \ldots, G_7\}$ from $G_{current}$ using 7 different actions
    Score each $G_j$ $(j = 1, \ldots, 7)$
    Select the best action $a^*$ that produces the highest scoring $G_j$
    Select a random action $a_{rand}$ and corresponding $G_{rand}$
    Calculate acceptance probability $P = \exp((\text{Score}(G_{rand}) - S_{current})/T)$
    **if** $\text{Score}(G_{rand}) > S_{current}$ **or** $\text{random}(0, 1) < P$ **then**
        Update $G_{current} \leftarrow G_{rand}$
        Update $S_{current} \leftarrow \text{Score}(G_{rand})$
        Append $a_{rand}$ to $A$
    **else**
        Update $G_{current} \leftarrow G_{a^*}$
        Update $S_{current} \leftarrow \text{Score}(G_{a^*})$
        Append $a^*$ to $A$
    **end if**
    Reduce temperature $T \leftarrow \alpha \cdot T$
**end while**
**return** $A$ =0
```

Therefore, based on the greedy algorithm we designed the n-step greedy algorithm. The core idea of the algorithm lies in the fact that if the traversal model takes a two-step ACTION, it is acceptable to expand the search space from 7 to 49. When we take a n-step, the search space is expanded from 7 to 7^n. Then from the point of view of computing power can choose the appropriate step length, so that the greedy algorithm also has the same Monte Carlo tree search ”future vision”. The pseudo-code for the algorithm to run is shown as Alg. 2.

The flexibility and optimization of the algorithm is increased while ensuring computational acceptability. The stepwise greedy algorithm must be able to reach an optimal solution without computational power constraints. But arithmetic constraints tend to be more common. Therefore we design the simulated annealing algorithm.

Simulated annealing algorithm, which avoids falling into a local optimum by accepting a worse solution with a certain probability at each step, tries to find a better global solution. The pseudo-code for the algorithm to run is shown as Alg. 3.

Research and experimentation with other algorithms will be conducted as part of our future work. The behavioral paths predicted using our algorithm for the test AIG are as list 1.

File	Action List
m4.aig	2434341333
i9.aig	3123314300
alu4.aig	3464234124
c7552.aig	2132412331
square.aig	3063034222
apex2.aig	3314362340
prom1.aig	2333124334
router.aig	0234000000
pair.aig	3204666260
sqrt.aig	2420000000
bar.aig	0000000000
b9.aig	3643000000
max1024.aig	4341240000
voter.aig	4042440340
apex4.aig	3330232003
c880.aig	3343000000
mem$_{c}trl.aig$	4030341303
apex1.aig	3364024602
cavlc.aig	3412464000
div.aig	2124220000

Table 1: AIG Files and their Corresponding Action Lists

3 Task3: High level synthesis with large language model

In this task, we are required to ask LLM Verilog designing questions. **Situations where LLM serves as a code helper include writing code according to our descriptions (code designing), explaining the functionality of code (code explaining).** Sometimes our descriptions are simple, sometimes detailed.

In this section, we ask LLMs to work out code designing/explaining questions on text-only condition.

3.1 Vectorr(Vector Reverse)

We first try a simple vector reverse problem on GPT4o, Gemini 1.5 Pro, QwenMax and LLaMA3. The former 2 of them are leading multimodal LLMs while the latter 2 are open source text-only LLMs.

This prompt is relatively detailed and 4 LLMs all give the right answer. However, QwenMax may not seize the clock setting in question prompt (**if not stated, signals are positive clock/clk edge triggered**) and give a seemingly complex answer.

Question Prompt:
Implement the Verilog module based on the following description. Assume that signals are positive clock/clk edge triggered unless otherwise stated.

Problem Description:
Given an 8-bit input vector [7:0], reverse its bit ordering.

```
module top_module (
        input [7:0] in,
        output [7:0] out
);
```

GPT4o Solution:

```
    assign {out[0],out[1],out[2],out[3],
        out[4], out[5],out[6],out[7]} =
         in;
endmodule
```

LLaMA3 Solution:

```
assign out = {in[0], in[1], in[2], in
    [3], in[4], in[5], in[6], in[7]};
endmodule
```

Qwen Max Solution:

```
always @(posedge clk) begin
    out[0] <= in[7];
    out[1] <= in[6];
    out[2] <= in[5];
    out[3] <= in[4];
    out[4] <= in[3];
    out[5] <= in[2];
    out[6] <= in[1];
    out[7] <= in[0];
end
endmodule
```

Gemini 1.5 Pro Solution:

```
    assign out[0] = in[7];
    assign out[1] = in[6];
    assign out[2] = in[5];
    assign out[3] = in[4];
    assign out[4] = in[3];
    assign out[5] = in[2];
    assign out[6] = in[1];
    assign out[7] = in[0];
endmodule
```

3.2 Gatesv(Gates Vector)

Now we ask LLMs to tackle a more difficult question with simple description.

3.2.1 Simple Description

Question Prompt:
Implement the Verilog module based on the following description. Assume that signals are positive clock/clk edge triggered unless otherwise stated.

Problem Description:
This module takes in 4-bit inputs and outputs 3-bit outputs based on the logical operations of AND, OR, and XOR.

GPT4o Solution:

```
module logical_operations(
    input clk,
    input [3:0] in1,
    input [3:0] in2,
    output [2:0] out_and,
    output [2:0] out_or,
    output [2:0] out_xor
);

reg [2:0] r_out_and;
reg [2:0] r_out_or;
reg [2:0] r_out_xor;

assign out_and = r_out_and;
assign out_or = r_out_or;
assign out_xor = r_out_xor;

always @(posedge clk) begin
    r_out_and <= in1[2:0] & in2[2:0];
    r_out_or <= in1[2:0] | in2[2:0];
    r_out_xor <= in1[2:0] ^ in2[2:0];
end
endmodule
```

Qwen Max Solution:

```
module logical_operations (
    input wire clk,
    input wire [3:0] a,
    input wire [3:0] b,
    output reg [2:0] and_out,
    output reg [2:0] or_out,
    output reg [2:0] xor_out
);

always @(posedge clk) begin
    and_out[0] <= a[0] & b[0];
    and_out[1] <= a[1] & b[1];
    and_out[2] <= a[2] & b[2];

    or_out[0] <= a[0] | b[0];
    or_out[1] <= a[1] | b[1];
    or_out[2] <= a[2] | b[2];

    xor_out[0] <= a[0] ^ b[0];
    xor_out[1] <= a[1] ^ b[1];
    xor_out[2] <= a[2] ^ b[2];
end
endmodule
```

LLaMA3 Solution:

```
module logical_ops (
    input  clk,
    input [3:0] in_a,
    input [3:0] in_b,
    output [2:0] out_and,
    output [2:0] out_or,
    output [2:0] out_xor
);

reg [2:0] and_out;
reg [2:0] or_out;
reg [2:0] xor_out;

always @(posedge clk) begin
    and_out <= &(in_a & in_b);
    or_out  <= |(in_a | in_b);
    xor_out <= ^(in_a ^ in_b);
end

assign out_and = and_out;
assign out_or  = or_out;
assign out_xor = xor_out;

endmodule
```

Gemini 1.5 Pro Solution:

```
module logic_operations (
    input wire [3:0] in1,
    input wire [3:0] in2,
    output wire [2:0] out_and,
    output wire [2:0] out_or,
    output wire [2:0] out_xor
);
    assign out_and[0] = in1[0] & in2[0];
    assign out_and[1] = in1[1] & in2[1];
    assign out_and[2] = in1[2] & in2[2];

    assign out_or[0] = in1[0] | in2[0];
    assign out_or[1] = in1[1] | in2[1];
    assign out_or[2] = in1[2] | in2[2];

    assign out_xor[0] = in1[0] ^ in2[0];
    assign out_xor[1] = in1[1] ^ in2[1];
    assign out_xor[2] = in1[2] ^ in2[2];
endmodule
```

With simple description i.e. no module code given, we only tell LLMs what function we want to realize, the response can be quite long compared to what you will see in detailed description case.

3.2.2 Detailed Description

Question Prompt:
Implement the Verilog module based on the following description. Assume that signals are positive clock/clk edge triggered unless otherwise stated.

Problem Description:
This Verilog module takes four input bits and produces three output bits. The first output bit is the result of a bitwise AND operation between the two least significant bits of the input. The second output bit is the result of a bitwise OR operation between the two least significant bits of the input. The third output bit is the result of a bitwise XOR operation between the two least significant bits of the input and all the other bits of the input except for the least significant bit.

GPT4o Solution:

```
module logical_operations(
    input clk,
    input [3:0] in,
    output reg out_and,
    output reg out_or,
    output reg out_xor
);

always @(posedge clk) begin
    out_and <= in[1] & in[0];
    out_or <= in[1] | in[0];
    out_xor <= (in[1] ^ in[0]) ^ in[3] ^
         in[2] ^ in[0];
end
endmodule
```

Qwen Max Solution:

```
module bitwise_operations (
    input wire clk,
    input wire [3:0] input_bits,
    output reg and_result,
    output reg or_result,
    output reg xor_result
);
always @(posedge clk) begin
    and_result <= input_bits[0] &
        input_bits[1];
    or_result <= input_bits[0] |
        input_bits[1];
    xor_result <= (input_bits[0] ^
        input_bits[1]) ^ (input_bits[2]
         ^ input_bits[3]);
end
endmodule
```

LLaMA3 Solution:

```
module bitwise_ops (
    input  clk,
    input [3:0] in,
    output [2:0] out
);

reg [2:0] out_reg;

always @(posedge clk) begin
    out_reg[0] <= in[0] & in[1];
    out_reg[1] <= in[0] | in[1];
    out_reg[2] <= (in[0] ^ in[1]) ^ (in
        [2] ^ in[3]);
end

assign out = out_reg;

endmodule
```

Gemini 1.5 Pro Solution:

```
module bit_operations (
    input [3:0] in,
    output [2:0] out
);
    assign out[0] = in[0] & in[1];
    assign out[1] = in[0] | in[1];
    assign out[2] = (in[0] ^ in[1]) ^ (
        in[2] | in[3]);

endmodule
```

We can see that LLMs show great instruction following capability. If we give specific steps, they can understand and follow the lead. A more detailed description leads to more reduced and concrete answer. But we can also observe some redundancy in LLaMA3's response which define an extra `reg [2:0] out_reg`.

3.3 Code Explaining

Like text summary, sometimes we want to have a quick scanning of code, which means we want some natural language to tell us what purpose a program is used for or a simple input-output example.

It's an interesting try to see whether LLMs can understand the code written by themselves. Here we clean the conversation memory and wait to see how LLMs think of feasible gatesv code.

We use the code generated by GPT4o in the former section as our problem.

System Prompt:
Explain the high-level functionality of the Verilog module. Use as many high-level concepts that are directly applicable to describe the code, say at the level of an undergraduate EECS major, but do not include extraneous details that aren't immediately applicable. Use text-based truth tables and state transition graphs when necessary. Speak concisely as if this was a specification for a circuit designer to implement. You should only reply with descriptive natural language and not use any code. Use no more than 50 words.

Few-Shot Example:
Question: Explain the high-level functionality of the Verilog module.

```
module top_module (
input [7:0] in,
output [7:0] out
);
assign {out[0],out[1],out[2],out[3],out[4],out[5],out[6],
    out[7]} = in;
endmodule
```

Answer: This top Verilog module is a 8-bit unidirectional data port. It takes an 8-bit input in, and outputs an 8-bit signal out. The output of the module is assigned to out based on the value of in. If in = a,b,c,d,e,f,g,h, then out = h,g,f,e,d,c,b,a

User Prompt:
Question: Explain the high-level functionality of the Verilog module.

```
module logical_operations(
    input clk,
    input [3:0] in,
    output reg out_and,
    output reg out_or,
    output reg out_xor
);

always @(posedge clk) begin
    out_and <= in[1] & in[0];
    out_or <= in[1] | in[0];
    out_xor <= (in[1] ^ in[0]) ^ in[3] ^ in[2] ^ in[0];
end
endmodule
```

GPT4o Solution:
This Verilog module performs basic logical operations on a 4-bit input vector ‘in‘. On the positive edge of the clock signal ‘clk‘, it computes AND, OR, and XOR operations involving specific bits of ‘in‘, and outputs the results through ‘out_and‘, ‘out_or‘, and ‘out_xor‘ respectively.

LLaMA3 Solution:
This Verilog module performs logical operations on a 4-bit input in synchronous with a clock signal clk. It outputs three 1-bit signals: out_and (AND operation on bits 0 and 1), out_or (OR operation on bits 0 and 1), and out_xor (XOR operation on all four input bits).

Qwen Max Solution:
This Verilog module performs bitwise logical operations on a 4-bit input in, synchronized to a clock signal clk. It produces three output registers: out_and for bit-wise AND, out_or for bit-wise OR, and out_xor for a combined XOR operation involving all input bits. The XOR operation is a customized function, not a simple pair-wise operation.

Gemini 1.5 Pro Solution:
This Verilog module implements three logical operations: AND, OR, and XOR. It takes a 4-bit input 'in' and a clock signal 'clk' as inputs. On the rising edge of the clock, the module calculates the AND, OR, and XOR of the first two bits of 'in' and outputs the results to 'out_and', 'out_or', and 'out_xor', respectively. The XOR operation also includes the remaining bits of 'in'.

Do you still remember our simple descriptioin of this problem ?

This module takes in 4-bit inputs and outputs 3-bit out-puts based on the logical operations of AND, OR, and XOR.

Basically I think 4 LLMs seize the key of the program. Attention that I limit the word count within 50 and give an example, LLMs reflect obviously to these commands. You may try dierectly use the user prompt, you will get a extremely detailed answer.

4 Assignment Proportion

We list our assignment proportion in the following table. Everyone in our group actively participate in the whole process, thus we are all equal to 1/3.

Name	**ID**	**Work**	**Proportion**
Qihui Zhou	521021910909	task1 and writing	1/3
Hao Li	520021910660	task2 and writing	1/3
Haoxiang Sun	518030910156	task3 and writing	1/3

Table 2: Assignment Proportion