«به نام او»

گزارش فاز اول پروژه

درس آزمون‌پذیری

دکتر حسابی

علی صداقی ۴۰۲۲۱۱۶۳۷

مهدی قصاب ۴۰۱۲۱۲۴۴۳

پاییز ۱۴۰۲

فهرست

بخش اول : نحوه پیاده‌سازی

در این بخش، نحوه‌ی پیاده‌سازی قسمت‌های مختلف پروژه مانند چگونگی خواندن فایل bench.، نحوه‌ی اعمال مقادیر ورودی به مدار، تولید خروجی و نحوه پیاده‌سازی و عملکرد Deductive Fault Simulation و الگوریتم‌های نوشته شده برای پیاده‌سازی این پروژه را شرح خواهیم داد.

خواندن فایل bench.

خواندن فایل bench. توسط تابع "process_circuit_file(file_path)" انجام می‌شود. این تابع علاوه بر خوانده فایل bench.، شناسایی و تعریف شاخه‌های fanout، شناسایی و اضافه کردن اجزا و گیت‌های مدار و در نهایت اضافه کردن اطلاعاتی مثل ورودی، خروجی و شاخه‌های fanout به آن را انجام می‌دهد. در ادامه عملکرد و نحوه پیاده‌سازی این تابع را توضیح خواهیم داد.

بطور کلی تابع "process_circuit_file(file_path)" مدار را پردازش کرده و اطلاعات مدار را استخراج می‌کند که فایل مدار، که مسیر آن به عنوان ورودی به تابع process_circuit_file داده می‌شود، شامل توصیفات مدار و اتصالات مختلف است. در ابتدا، فایل مدار باز می‌شود و هر خط آن به عنوان یک المان در لیست lines ذخیره می‌شود. سپس، تابع به دو مرحله عمل می‌کند:

- مرحله اول: شمارش تعداد Fanout

در این مرحله، برای هر خط در lines، اگر عبارت "=" در آن وجود داشته باشد، ورودی‌های مربوط به آن خط استخراج می‌شوند. این ورودی‌ها به صورت رشته‌های جداگانه درون پرانتز قرار دارند و با استفاده از توابع split و isdigit، عناصر عددی موجود در ورودی‌ها شناسایی می‌شوند. سپس تعداد فراوانی هر ورودی در دیکشنری fanout_counts ثبت می‌شود.

- مرحله دوم: ایجاد اجزای مدار

در این مرحله، لیست‌های circuit، inputs و outputs تعریف می‌شوند که به ترتیب برای ذخیره کردن اجزای مدار، ورودی‌ها و خروجی‌ها استفاده می‌شوند.

برای هر خط در lines، خط به صورت strip شده و سپس با استفاده از شروط if و elif، نوع آن تشخیص داده می‌شود. اگر با عبارت "INPUT" آغاز شود، عدد ورودی استخراج می‌شود و به لیست inputs اضافه می‌شود. اگر با عبارت "OUTPUT" آغاز شود، عدد خروجی استخراج می‌شود و به لیست outputs اضافه می‌شود. در غیر این صورت، اجزای مربوط به یک خط از مدار استخراج می‌شوند.

اجزای مربوط به یک خط از مدار شامل شماره خروجی (output_num)، عملیات (operation) و ورودی‌ها (inputs_nums) است. با استفاده از توابع split، این اجزا استخراج می‌شوند. سپس، ورودی‌ها با توجه به وجود fanoutها جایگزین می‌شوند. اگر ورودی مورد نظر در دیکشنری fanouts وجود داشته باشد، اولین عنصر موجود در لیست fanouts[inp] حذف شده و به عنوان ورودی جایگزین می‌شود. در غیر این صورت، ورودی بدون تغییر استفاده می‌شود. سپس اجزای مدار به لیست circuit اضافه می‌شوند.

در انتها، اطلاعات fanout، ورودی‌ها و خروجی‌ها به لیست circuit اضافه می‌شوند. برای هر کلید در دیکشنری fanouts_copy، لیستی از عناصر values به کلید متناظر اضافه می‌شود و به عنوان یک اجزای مدار با نوع "FANOUT" به circuit اضافه می‌شود. سپس لیست inputs به عنوان یک اجزای مدار با نوع "INPUT" و لیست outputs به عنوان یک اجزای مدار با نوع "OUTPUT" به circuit اضافه می‌شوند.

در نهایت، لیست circuit به عنوان خروجی تابع برگردانده می‌شود. این لیست شامل تمام اجزا و اتصالات مدار است که برای استفاده بعدی می‌تواند استفاده شود.

توصیف مدار جهت شبیه‌سازی

تابع "split_circuit(circuit_data)" یک مدار را به بخش‌های مختلف تقسیم می‌کند و هر بخش را در لیست‌های جداگانه قرار می‌دهد. بخش‌هایی که تقسیم می‌شود عبارتند از: ورودی‌ها، گیت‌ها، fanout و خروجی‌ها. ابتدا، چهار لیست جداگانه به نام‌های input_list، gate_list، fanout_list و output_list تعریف می‌شوند. این لیست‌ها جهت نگهداری اطلاعات مربوط به هر بخش از مدار استفاده می‌شوند. سپس، با استفاده از حلقه for، برای هر عنصر e در circuit_data، با استفاده از عبارت شرطی، مشخص می‌شود که عنصر e به کدام بخش از مدار تعلق دارد. اگر عنصر e با "INPUT" شروع شود، یعنی یک ورودی است، مقادیر بعدی آن را به input_list اضافه می‌کند. اگر عنصر e با "FANOUT" شروع شود، یعنی یک fanout است و عنصر e را به fanout_list اضافه می‌کند. اگر عنصر e با "OUTPUT" شروع شود، یعنی یک خروجی است، مقادیر بعدی آن را به output_list اضافه می‌کند. در غیر اینصورت، عنصر e یک گیت است و به gate_list اضافه می‌شود. در نهایت، این تابع لیست‌های input_list، gate_list، fanout_list و output_list را به عنوان خروجی برمی‌گرداند. این لیست‌ها شامل اطلاعات مربوط به ورودی‌ها، گیت‌ها، مولد خروجی و خروجی‌های مدار هستند.

در ادامه برای توصیف مدار جهت شبیه‌سازی، از توابع زیر به ترتیب استفاده کردیم که هر تابع را به طور مختصر در ادامه توضیح خواهیم داد:

create_wires(circuit_data): این تابع لیستی از نام سیم‌ها (wires) را براساس داده‌های مدار ورودی می‌سازد. در این تابع، برای هر عنصر مدار در circuit_data، نام سیم‌های مربوطه استخراج شده و به لیست wires اضافه می‌شوند. نام سیم‌ها بدون پیشوند "w" در لیست قرار می‌گیرند. سپس، لیست wires به صورت مجموعه (set) تبدیل شده و به عنوان خروجی برگردانده می‌شود.

create_simulation_list(wires): این تابع یک دیکشنری شبیه‌سازی را براساس لیست سیم‌ها (wires) می‌سازد. برای هر سیم در wires، یک کلید با پیشوند "w" و مقدار متناظر با آن (شروع‌اش با

"U") در دیکشنری simulation_dict قرار می‌گیرد. در نهایت، دیکشنری simulation_dict به عنوان خروجی برگردانده می‌شود.

create_input_wires(input_list): این تابع لیستی از نام سیم‌های ورودی را براساس لیست input_list ایجاد می‌کند. برای هر سیم در input_list، نام آن سیم با پیشوند "w" در لیست input_wires قرار می‌گیرد. لیست input_wires به عنوان خروجی برگردانده می‌شود.

create_gates_output_wires(gate_list): این تابع لیستی از نام سیم‌های خروجی گیت‌ها را براساس لیست gate_list ایجاد می‌کند. برای هر گیت در gate_list، نام سیم خروجی آن گیت در لیست gates_output_wires قرار می‌گیرد. سپس، لیست gates_output_wires به ترتیب اعداد مرتب می‌شود و به عنوان خروجی برگردانده می‌شود.

find_gate(gate_out_wire, gate_list): این تابع با دریافت نام سیم خروجی یک گیت (gate_out_wire) و لیست gate_list، نوع گیت (عملیات) متناظر با آن سیم خروجی را پیدا می‌کند. برای هر گیت در gate_list، اگر نام سیم خروجی آن با gate_out_wire برابر باشد، نوع گیت را برمی‌گرداند.

find_gate_inputs(gate_out_wire, gate_list): این تابع با دریافت نام سیم خروجی یک گیت و لیست gate_list، لیستی از نام سیم‌های ورودی مربوط به آن سیم خروجی را پیدا می‌کند. برای هر گیت در gate_list، اگر نام سیم خروجی آن با gate_out_wire برابر باشد، نام سیم‌های ورودی آن گیت را به لیست gate_inputs اضافه می‌کند و در نهایت، لیست gate_inputs را به عنوان خروجی برمی‌گرداند.

is_fanin(wire_name, fanout_list): این تابع بررسی می‌کند که آیا یک سیم (wire_name) در لیست fanout_list به عنوان ورودی یک fanout وجود دارد یا خیر. در صورتی که سیم مورد نظر در لیست fanout_list باشد، مقدار True برگردانده می‌شود و در غیر اینصورت، مقدار False برگردانده می‌شود.

find_fanout_lines(fanin_wire, fanout_list): این تابع با دریافت نام سیم ورودی یک fanin_wire و لیست fanout_list، لیستی از خطوط مولد خروجی مربوط به آن سیم ورودی را پیدا می‌کند. برای هر مولد خروجی در fanout_list، اگر نام سیم ورودی آن با fanin_wire برابر باشد، لیست خطوط مربوطه را برمی‌گرداند. در صورتی که fanout مورد نظر پیدا نشود، None برگردانده می‌شود.

اعمال مقادیر ورودی به مدار

برای این منظور ما ابتدا ما فایل که مقدایر ورودی در آن قرار د ارد با استفاده از تابع "process_input_file(filepath)" می‌خوانیم؛ این تابع یک فایل ورودی را پردازش می‌کند و اطلاعات آن را به صورت یک دیکشنری برمی‌گرداند. برای این منظور ابتدا، فایل ورودی با استفاده از تابع open باز می‌شود و تمام خطوط آن در لیست lines ذخیره می‌شوند. سپس، با فرض اینکه فایل دقیقاً دو خط دارد، ابتدا خط اول را استخراج می‌کند. خط اول شامل کلیدهای دیکشنری است، که با استفاده از توابع strip و split، از فاصله‌ها جدا می‌شوند و در لیست keys قرار می‌گیرند. سپس، خط دوم را استخراج می‌کند. خط دوم شامل مقادیر متناظر با کلیدهای دیکشنری است، که نیز با استفاده از توابع strip و split، از فاصله‌ها جدا می‌شوند و در لیست values قرار می‌گیرند. سپس، با استفاده از تابع zip، کلیدها و مقادیر به صورت زوج‌های متناظر گرفته می‌شوند و با استفاده از نمایه‌گذاری، یک دیکشنری با نام result_dict ساخته می‌شود. در نهایت، این دیکشنری به عنوان خروجی تابع برگردانده می‌شود. این دیکشنری شامل اطلاعات موجود در فایل ورودی است که با استفاده از کلیدها و مقادیر متناظر در فایل ساخته شده است.

سپس با استفاده از تابع "give_input_vector" مقادیر ورودی به سیم‌های ورودی مدار توصیف شده که در قسمت قبل نحوه تولید آن را شرح دادیم، داده می‌شود. این تابع یک ورودی برداری را به دیکشنری شبیه‌سازی وارد می‌کند و یک نسخه کپی از دیکشنری شبیه‌سازی را با ورودی‌های متغیر به عنوان مقادیر ورودی برداری تولید می‌کند. ابتدا، تابع با استفاده از تابع copy.deepcopy یک نسخه کپی از دیکشنری simulation_dict می‌سازد و در متغیر simulation_dict_copy ذخیره می‌کند. این کپی به منظور اعمال تغییرات بر روی آن و جلوگیری از تغییرات در دیکشنری اصلی استفاده می‌شود. سپس، با استفاده از

عبارت شرطی if، ابتدا مطمئن می‌شود که تعداد عناصر در لیست input_wires با تعداد عناصر در دیکشنری input_vector برابر است. اگر برابر نباشند، یک خطا به نام ValueError بوجود می‌آید. در صورت برابر بودن تعداد عناصر، با استفاده از حلقه for، برای هر زوج کلید-مقدار در input_vector، کلید را با پیشوند "w" به عنوان مقدار کلید در simulation_dict_copy قرار می‌دهد و مقدار را با مقدار متناظر در input_vector جایگزین می‌کند. به این ترتیب، ورودی‌های متغیر تعیین شده در input_vector بر روی دیکشنری شبیه‌سازی اعمال می‌شوند. در نهایت، نسخه کپی شده از دیکشنری شبیه‌سازی با تغییرات اعمال شده به عنوان خروجی تابع برگردانده می‌شود. این نسخه کپی شامل تغییرات ورودی برداری است که با استفاده از input_wires و input_vector اعمال شده است. توجه کنید که دیکشنری اصلی simulation_dict تغییر نکرده است و تغییرات فقط در نسخه کپی اعمال شده است.

نحوه شبیه‌سازی True-Value و تولید خروجی

برای شبیه‌سازی True-Value یک مدار منطقی با استفاده از ورودی‌ها، یک دیکشنری شبیه‌سازی و لیست‌های fanout از دو تابع gate_simulator، true_value_simulation استفاده می‌شود (درواقع تابع gate_simulator داخل تابع true_value_simulation بکار برده شده است).

تابع gate_simulator برای شبیه‌سازی عملکرد گیت‌های منطقی استفاده می‌شود. این تابع دو ورودی دریافت می‌کند: gate_type که نوع گیت را مشخص می‌کند و gate_inputs که لیست ورودی‌های گیت را نشان می‌دهد. این تابع ابتدا با استفاده از دستورات شرطی بررسی می‌کند که نوع گیت چیست و بر اساس آن به تصمیم‌گیری می‌پردازد.

- اگر نوع گیت "AND" باشد، ابتدا بررسی می‌شود که آیا حداقل یکی از ورودی‌ها مقدار "۰" دارد. اگر داشته باشد، خروجی "۰" است. سپس بررسی می‌شود که آیا حداقل یکی از ورودی‌ها مقدار "U" یا "Z" دارد. اگر داشته باشد، خروجی "U" است. در غیر این صورت، خروجی "۱" است.

- اگر نوع گیت "OR" باشد، ابتدا بررسی می‌شود که آیا حداقل یکی از ورودی‌ها مقدار "۱" دارد. اگر داشته باشد، خروجی "۱" است. سپس بررسی می‌شود که آیا حداقل یکی از ورودی‌ها مقدار "U" یا "Z" دارد. اگر داشته باشد، خروجی "U" است. در غیر این صورت، خروجی "۰" است.

- اگر نوع گیت "NAND" باشد، ابتدا بررسی می‌شود که آیا حداقل یکی از ورودی‌ها مقدار "۰" دارد. اگر داشته باشد، خروجی "۱" است. سپس بررسی می‌شود که آیا حداقل یکی از ورودی‌ها مقدار "U" یا "Z" دارد. اگر داشته باشد، خروجی "U" است. در غیر این صورت، خروجی "۰" است.

- اگر نوع گیت "NOR" باشد، ابتدا بررسی می‌شود که آیا حداقل یکی از ورودی‌ها مقدار "۱" دارد. اگر داشته باشد، خروجی "۰" است. سپس بررسی می‌شود که آیا حداقل یکی از ورودی‌ها مقدار "U" یا "Z" دارد. اگر داشته باشد، خروجی "U" است. در غیر این صورت، خروجی "۱" است.

- اگر نوع گیت "NOT" باشد، ابتدا بررسی می‌شود که تنها یک ورودی دارد. اگر تعداد ورودی‌ها بیشتر یاز یک عدد باشد، یک خطا برگردانده می‌شود. سپس بررسی می‌شود که ورودی تنها "۱" باشد، در این صورت خروجی "۰" است، و اگر ورودی تنها "۰" باشد، خروجی "۱" است. در غیر این صورت، خروجی "U" است.

- اگر نوع گیت "BUFF" باشد، ابتدا بررسی می‌شود که تنها یک ورودی دارد. اگر تعداد ورودی‌ها بیشتر از یک عدد باشد، یک خطا برگردانده می‌شود. سپس بررسی می‌شود که ورودی "۱" باشد، در این صورت خروجی "۱" است، و اگر ورودی "۰" باشد، خروجی "۰" است. در غیر این صورت، خروجی "U" است.

- در صورتی که نوع گیت از نوع "XOR" یا "XNOR" باشد، ابتدا ورودی‌ها به عدد صحیح تبدیل می‌شوند و سپس بررسی می‌شود که مجموع این اعداد به صورت تقسیم عدد بر ۲ باقی‌مانده دارد یا خیر. اگر باقی‌مانده برابر ۱ باشد، خروجی "۱" است و در غیر این صورت، خروجی "۰" است. در صورتی که ورودی‌ها شامل حرف "U" یا "Z" باشند، خروجی "U" است.
- در نهایت، در صورتی که نوع گیت معتبر نباشد، یک خطا برگردانده می‌شود.

```
[('AND', ['1', '1', '1'], '1'),
 ('AND', ['0', '1', 'U'], '0'),
 ('OR', ['0', '0', '0'], '0'),
 ('OR', ['0', 'U', '1'], '1'),
 ('XOR', ['0', '0'], '0'),
 ('XOR', ['1', '0', '1', '1'], '1'),
 ('XOR', ['1', '1', '1', '1'], '0'),
 ('XOR', ['1', '1', 'U'], 'U'),
 ('XNOR', ['1', '0'], '0'),
 ('XNOR', ['1', '1', '1'], '0'),
 ('XNOR', ['1', '0', '1'], '1'),
 ('XNOR', ['0', '0', 'Z'], 'U'),
 ('NAND', ['1', '1'], '0'),
 ('NAND', ['1', 'U', '0'], '1'),
 ('NOR', ['0', '0'], '1'),
 ('NOR', ['U', '0'], 'U'),
 ('NOT', ['1'], '0'),
 ('NOT', ['Z'], 'U'),
 ('BUFF', ['0'], '0'),
 ('BUFF', ['U'], 'U')]
```

شکل شماره ۱. خروجی تولید شده برای گیت‌های مختلف

تابع true_value_simulation با استفاده از یک وکتور ورودی، یک شبیه‌سازی True-Value را برای یک مدار منطقی انجام می‌دهد. تابع ورودی‌های مختلفی دارد که به ترتیب شرح داده می‌شوند:

- input_vector: یک وکتور که مقادیر ورودی‌های مدار منطقی را نشان می‌دهد.
- simulation_dict: یک دیکشنری که شامل مقادیر وضعیت فعلی مدار منطقی است.

- fanout_list: یک لیست از نقاط خروجی که برای هر یک از آنها لیستی از نقاط ورودی مربوط به آن وجود دارد.
- gate_list: یک لیست از گیت‌های موجود در مدار منطقی که شامل نوع گیت، نقطه خروجی و نقاط ورودی است.
- input_wires: یک لیست از نقاط ورودی مدار منطقی.
- gates_output_wires: یک لیست از نقاط خروجی گیت‌های مدار منطقی.

ابتدا تابع یک نسخه کپی از simulation_dict را با استفاده از copy.deepcopy ایجاد می‌کند. سپس با استفاده از تابع give_input_vector مقادیر ورودی را در simulation_dict_copy قرار می‌دهد. سپس تابع make_fanout_equivalence فراخوانی می‌شود. این تابع با استفاده از fanout_list، مقادیر ورودی را به نقاط خروجی متناظر منتقل می‌کند. سپس با استفاده از یک حلقه for، برای هر نقطه خروجی در gates_output_wires، نوع گیت و ورودی‌های مربوط به آن گیت را پیدا می‌کند. سپس با استفاده از تابع gate_simulator مقدار خروجی گیت را محاسبه می‌کند. مقادیر وضعیت خروجی گیت را در simulation_dict_copy ذخیره می‌کند. سپس دوباره تابع make_fanout_equivalence اعمال می‌شود. در نهایت، simulation_dict_copy را برگردانده می‌شود که شامل مقادیر وضعیت جدید مدار منطقی است.

توجه کنید هنگام اعمال شبیه‌سازی، مدار را ابتدا سطح بندی کرده بودیم و مراحل بالا را باید در هر سطح انجام می‌دهیم تا به خروجی برسیم.

نحوه پیاده‌سازی Deductive Fault Simulation

ما برای پیاده‌سازی Deductive fault simulation، از الگوریتم بیان شده در کلاس و مقادیر کنترل c و i استفاده کردیم، شکل شماره ۲ این الگوریتم را نشان می‌دهد.

$$\text{if } S = \phi \quad \text{then}$$
$$L_Z = \{ \cup_{j \in I} L_j \} \cup \{ Z \text{ s-a- } (c \oplus i) \}$$
$$\text{else}$$
$$L_Z = \{ \cap_{j \in S} L_j \} - \{ \cup_{j \in I-S} L_j \} \cup \{ Z \text{ s-a-} (\bar{c} \oplus i) \}$$

شکل شماره ۲ . الگوریتم استفاده شده برای پیاده‌سازی Deductive fault simulation

درواقع تابع gate_dfs، الگوریتم بالا را در پروژه ما پیاده‌سازی می‌کند. این تابع ابتدا با دریافت نوع گیت و ورودی‌های متناظر با آن، نحوه تولید فالت لیست خروجی براساس فالت لیست ورودی‌ها را تعیین می‌کند. این تابع ورودی‌های مختلفی دارد که به ترتیب شرح داده می‌شوند:

- gate: نوع گیت که می‌تواند مقادیر زیر را داشته باشد: 'AND', 'NAND', 'OR', 'NOR', 'XOR', 'XNOR', 'BUFF', 'NOT'.
- inputs: لیستی از مقادیر ورودی‌های گیت که می‌توانند ۰ و ۱ باشند.

تابع در ابتدا بررسی می‌کند که آیا تمام ورودی‌ها مقادیر صحیح (۰ و ۱) هستند یا خیر. اگر حداقل یکی از ورودی‌ها مقدار نامعتبر داشته باشد، یک خطا با پیام "Invalid input values. All inputs must be ۰ or ۱" نمایش می‌دهد.

سپس با استفاده از if-elif-else، نحوه تولید فالت لیست خروجی را محاسبه و آن را به صورت یک لیست برمی‌گرداند. شکل شماره ۳، خروجی این تابع برای گیت‌های مختلف را نشان می‌دهد.

```
# Test cases format: (gate, inputs, expected_output)
('AND', ['0', '0'], ['0', '0', '0']),
('AND', ['0', '1'], ['0', '1', '0']),
('AND', ['1', '0'], ['1', '0', '0']),
('AND', ['1', '1'], ['0', '0', '1']),
('NAND', ['0', '0'], ['0', '0', '0']),
('NAND', ['0', '1'], ['0', '1', '0']),
('NAND', ['1', '0'], ['1', '0', '0']),
('NAND', ['1', '1'], ['0', '0', '1']),
('OR', ['0', '0'], ['0', '0', '1']),
('OR', ['0', '1'], ['1', '0', '0']),
('OR', ['1', '0'], ['0', '1', '0']),
('OR', ['1', '1'], ['0', '0', '0']),
('NOR', ['0', '0'], ['0', '0', '1']),
('NOR', ['0', '1'], ['1', '0', '0']),
('NOR', ['1', '0'], ['0', '1', '0']),
('NOR', ['1', '1'], ['0', '0', '0']),
('AND', ['1', '0', '1'], ['1', '0', '1', '0']),
('AND', ['1', '1', '1'], ['0', '0', '0', '1']),
('NAND', ['0', '0', '1'], ['0', '0', '1', '0']),
('NAND', ['1', '1', '1'], ['0', '0', '0', '1']),
('OR', ['0', '0', '0'], ['0', '0', '0', '1']),
('OR', ['1', '0', '1'], ['0', '1', '0', '0']),
('NOR', ['0', '1', '0'], ['1', '0', '1', '0']),
('NOR', ['1', '1', '1'], ['0', '0', '0', '0']),
('NOT', ['0'], ['1']),
('NOT', ['1'], ['0']),
('BUFF', ['0'], ['0']),
('BUFF', ['1'], ['0']),
```

شکل شماره ۳ . خروجی تابع gate_dfs

همانطور که اشاره کردیم، خروجی این تابع بصورت یک لیست با ابعاد "تعداد ورودی + ۱" می‌باشد، این تابع این لیست را براساس همان منطق الگوریتم مطرح شده در شکل شماره ۲ ایجاد می‌کند، به این صورت که با توجه به نوع گیت، دنبال مقدار کنترلی مختص آن گیت در ورودی‌ها می‌گردد و با توجه به حضور با عدم حضور آن مقدار کنترلی در ورودی‌ها، لیست خروجی را تولید می‌کند که نحوه توصیف لیست تولید شده به صورت زیر است:

- مقدار اول تا n-۱: اگر برابر ۱ بود یعنی مکمل لیست اشکال ورودی متناظر، اگر برابر ۰ یعنی خود لیست اشکال ورودی متناظر در نظر گرفته می‌شود.

- آخرین مقدار یا مقدار nام: تعیین می‌کند باید اشتراک بگیریم یا اجتماع، مقدار ۰ یعنی اشتراک ، مقدار ۱ یعنی اجتماع

برای فهم بیشتر این تابع، جدول زیر نحوه توصیف خروجی آن را برای دو گیت AND و OR نشان می‌دهد:

a	b	خروجی تابع	توصیف منطقی خروجی
0	0	[0 , 0 , 0]	$L_a \cap L_b$
0	1	[0 , 1 , 0]	$L_a \cap \overline{L_b}$
1	0	[1 , 0 , 0]	$\overline{L_a} \cap L_b$
1	1	[0 , 0 , 1]	$L_a \cup L_b$

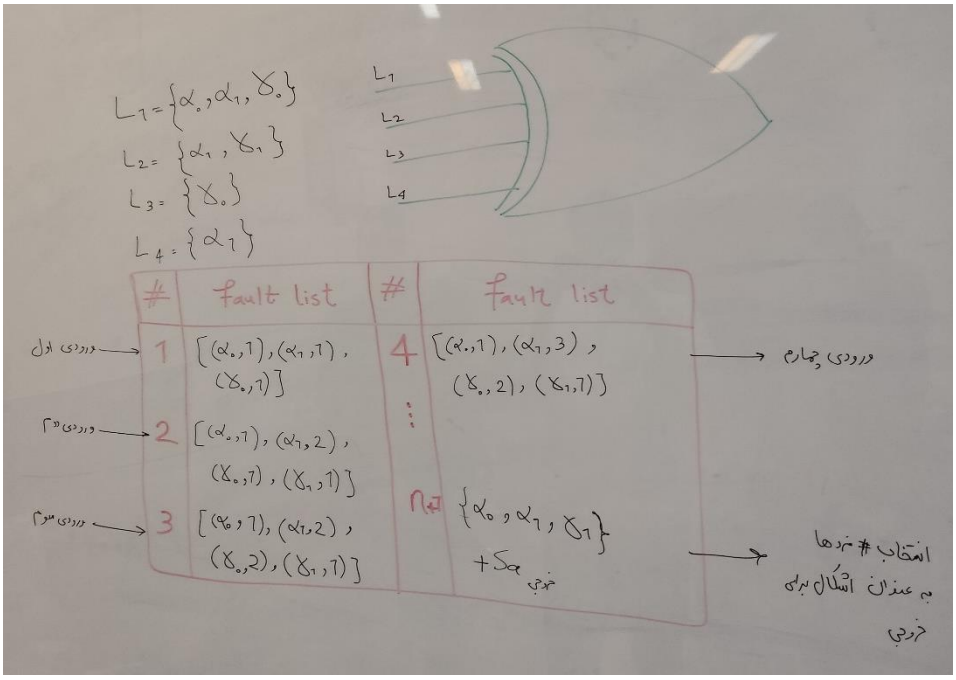
جدول شماره ۱ . خروجی تابع gate_dfs برای گیت AND دو ورودی

a	b	خروجی تابع	توصیف منطقی خروجی
0	0	[0 , 0 , 1]	$L_a \cup L_b$
0	1	[1 , 0 , 0]	$\overline{L_a} \cap L_b$
1	0	[0 , 1 , 0]	$L_a \cap \overline{L_b}$
1	1	[0 , 0 , 0]	$L_a \cap L_b$

جدول شماره ۱ . خروجی تابع gate_dfs برای گیت OR دو ورودی

توجه کنید به دلیل عدم تعریف مقدار کنترلی c و i برای گیت‌های XOR و XNOR، نحوه پیاده‌سازی این دو گیت با گیت‌های OR و AND و NAND و غیره متفاوت است و نمی‌توان از الگوریتم بالا برای آن‌ها استفاده کرد. برای پیاده‌سازی Deductive fault simulation برای گیت‌های XOR و XNOR، ما لیست اشکال تک تک ورودی‌ها را بررسی کرده و تعداد تکرار تمام لیست اشکال‌های تمام ورودی‌ها را میشماریم و در نهایت، اشکالاتی که فرد بار در ورودی‌ها رخ داده باشند را با اشکال خود خروجی، به عنوان لیست اشکال خروجی نهایی برمی‌گردانیم. در واقع دلیل این کار این هست که اشکالاتی منجر به بروز خطا و تغییر در خروجی می‌شوند که تعداد فرد از پایه‌های گیت را تغییر دهند، از طرفی با فرض single stuck at، ما باید

دنبال اشکالاتی باشیم که روی تعداد فردی از ورودی‌ها اثر گذاشته و آن‌ها را تغییر می‌دهند. شکل زیر این الگوریتم را با یک مثال نشان می‌دهد:



شکل شماره ۳ . خروجی تابع gate_dfs

تا الان ما نحوه تعیین لیست اشکال خروجی گیت‌ها را شرح دادیم و در ادامه می‌خواهیم نحوه پیاده سازی deductive fault simulation روی یک مدار با استفاده از الگوریتم‌های مطرح شده را شرح بدهیم. برای این منظور در ابتدا ما با استفاده از تابع init_fault_list، لیست اشکال اولیه در ورودی‌های اصلی یا همان primary input ها را تعیین می‌کنیم، سپس با استفاده از تابع deductive_fault_simulation، deductive fault simulation را به مدار اعمال می‌کنیم. این تابع ورودی‌های مختلفی دارد که به ترتیب شرح داده می‌شوند:

- La_dict: یک فاصله نامعلوم در ورودی به عنوان لغتنامه خطا استفاده می‌شود؛ یعنی یک دیکشنری که برای هر سیم ورودی، مجموعه خطاهای ممکن را نگهدارد.
- input_list: لیستی از سیم‌های ورودی.

- fanout_list: لیستی از سیم‌هایی که به عنوان خروجی گیت‌ها عمل می‌کنند.
- gates_output_wires: لیستی از سیم‌های خروجی گیت‌ها.
- gate_list: لیستی از گیت‌ها که برای هرکدام نوع گیت و ورودی‌های متناظر آن نگهداری می‌شود.
- tvs: تابعی که برای هر سیم ورودی مقدار خروجی تعیین می‌کند.
- U_set: مجموعه خطای نادرست برای تمام سیم‌ها.

ابتدا یک کپی عمیق از La_dict تهیه می‌شود تا تغییرات در آن اعمال شود (La_dict_copy). سپس برای هر سیم ورودی در input_list، لیست خطاها برای سیم‌هایی که به عنوان خروجی گیت‌ها عمل می‌کنند به روزرسانی می‌شود. این به معنی این است که هر خطایی که در سیم ورودی وجود داشته باشد، به لیست خطاهای سیم‌های خروجی متناظر افزوده می‌شود.

سپس برای هر سیم خروجی در gates_output_wires، نوع گیت و ورودی‌های متناظر آن را پیدا می‌کند. سپس مقادیر ورودی در value_gate_inputs ذخیره می‌شوند با استفاده از tvs و عناصر gate_inputs.

در مرحله بعد با استفاده از تابع gate_dfs، قوانین برای گیت مورد نظر محاسبه می‌شوند و در rules_list ذخیره می‌شوند.

سپس مجموعه‌های ورودی در input_sets ذخیره می‌شوند. برای هر ورودی در gate_inputs، اگر مقدار متناظر در rules_list برابر با '۰' باشد، مجموعه خطاهای مربوط به آن ورودی در La_dict_copy به input_set اضافه می‌شود؛ در غیر این صورت، مجموعه خطاهای کامپلمنت La_dict_copy برای آن ورودی در input_set قرار می‌گیرد. سپس input_set به input_sets اضافه می‌شود.

در قدم بعدی، بر اساس عنصر آخر در rules_list، اجتماع یا اشتراک مجموعه‌های ورودی در input_sets انجام می‌شود. اگر عنصر آخر rules_list برابر با '۱' باشد، اجتماع (set.union) مجموعه‌های ورودی انجام می‌شود و نتیجه در result_set ذخیره می‌شود؛ در غیر این صورت، اشتراک (set.intersection) مجموعه‌های ورودی انجام می‌شود و نتیجه در result_set ذخیره می‌شود. سپس، با ساختن کلید مربوطه برای La_dict_copy، که با پیشوند 'L_w' به gate_out_wire اضافه می‌شود، مجموعه result_set به La_dict_copy اضافه می‌شود.

سپس La_dict_copy با استفاده از تابع update_fault_list_for_fanout برای سیم‌های خروجی گیت، به روزرسانی می‌شود. در نهایت، La_dict_copy که شامل تغییرات اعمال شده است، به عنوان خروجی برگردانده می‌شود. شکل شماره ۴، خروجی این تابع را قسمت را نشان می‌دهد.

```
{'L_w1': {'w1_s1'},
 'L_w2': {'w2_s1'},
 'L_w3': {'w3_s0'},
 'L_w3_1': {'w3_1_s0', 'w3_s0'},
 'L_w3_2': {'w3_2_s0', 'w3_s0'},
 'L_w6': {'w6_s0'},
 'L_w7': {'w7_s0'},
 'L_w10': {'w10_s0', 'w1_s1'},
 'L_w11_2': {'w11_2_s1', 'w11_s1', 'w3_2_s0', 'w3_s0', 'w6_s0'},
 'L_w11_1': {'w11_1_s1', 'w11_s1', 'w3_2_s0', 'w3_s0', 'w6_s0'},
 'L_w11': {'w11_s1', 'w3_2_s0', 'w3_s0', 'w6_s0'},
 'L_w16_1': {'w16_1_s0', 'w16_s0'},
 'L_w16': {'w16_s0'},
 'L_w16_2': {'w16_2_s0', 'w16_s0'},
 'L_w19': {'w11_2_s1', 'w11_s1', 'w19_s0', 'w3_2_s0', 'w3_s0', 'w6_s0'},
 'L_w22': {'w10_s0', 'w16_1_s0', 'w16_s0', 'w1_s1', 'w22_s1'},
 'L_w23': {'w11_2_s1',
  'w11_s1',
  'w16_2_s0',
  'w16_s0',
  'w19_s0',
  'w23_s1',
  'w3_2_s0',
  'w3_s0',
  'w6_s0'}}
```

شکل شماره ۴ . خروجی قسمت Deductive Fault Simulation

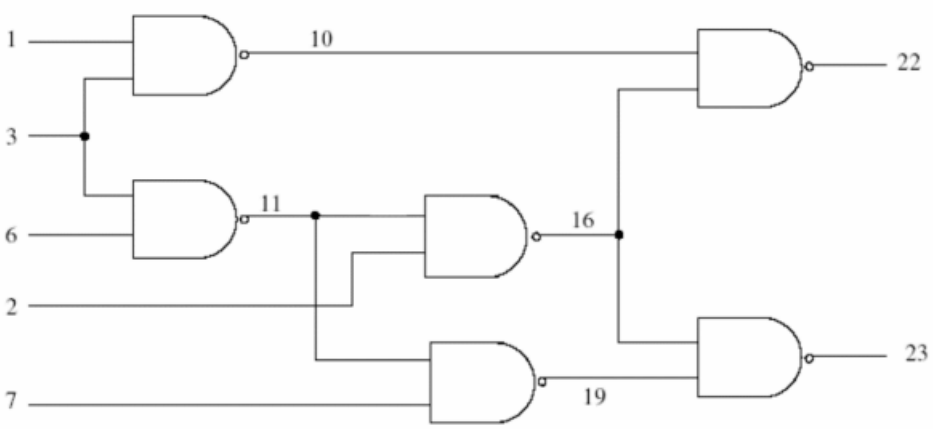
بخش دوم : دو نمونه از خروجی‌های شبیه‌سازی

در این بخش به ازی دو بردار تست مختلف برای حالت‌های True-Value Simulation و Deductive Fault Simulation، مدار C17 را شبیه‌سازی کرده و با مقادیر محاسبه شده از روش تئوری مقایسه می‌کنیم.

True-Value Simulation به ازای ورودی اول

در این قسمت مدار C17 را به ازای بردار تست زیر، شبیه‌سازی می‌کنیم.

$$W_1 = 0, W_3 = 1, W_6 = 1, W_2 = 0, W_7 = 1$$




```
...   {'w1': '0',
       'w2': '0',
       'w3': '1',
       'w3_1': '1',
       'w3_2': '1',
       'w6': '1',
       'w7': '1',
       'w10': '1',
       'w11': '0',
       'w11_1': '0',
       'w11_2': '0',
       'w16': '1',
       'w16_1': '1',
       'w16_2': '1',
       'w19': '1',
       'w22': '0',
       'w23': '0'}
```

شکل شماره ۵. خروجی شبیه‌سازی True-Value برای ورودی اول

در ادامه مقادیر تمامی سیم‌ها را بصورت دستی بدست می‌آوریم:

$W_{3\_1} = 1$

$W_{3\_2} = 1$

$W_{10} = 1$

$W_{11} = 0$

$W_{11\_1} = 0$

$W_{11_2} = 0$

$W_{16} = 1$

$W_{16\_1} = 1$

$W_{16\_2} = 1$

$W_{19} = 1$

$W_{22} = 0$

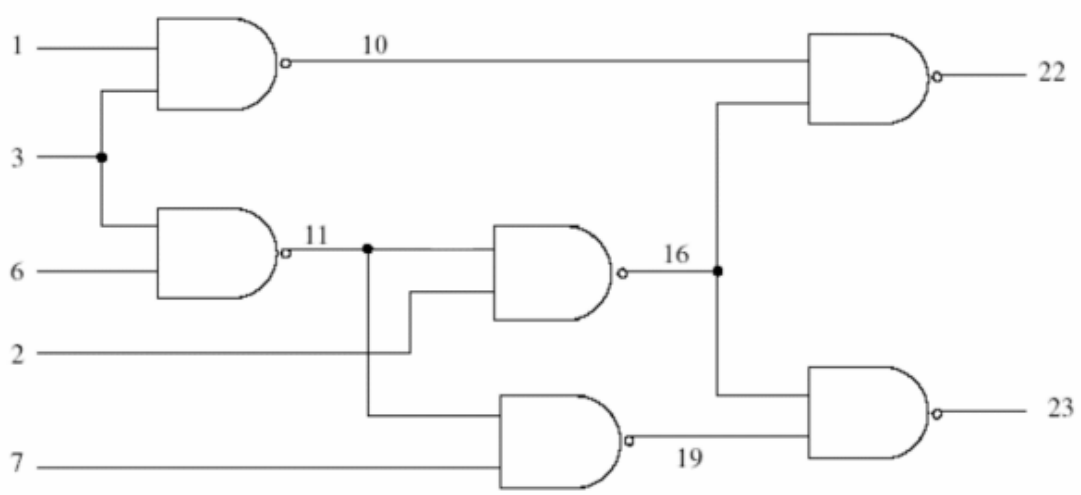
$W_{23} = 0$

با مقایسه مقادیر شبیه‌سازی و مقادیر محاسبه شده، می‌توان نتیجه گرفت که شبیه‌سازی درست انجام شده است.

True-Value Simulation به ازای ورودی دوم

در این قسمت مدار C17 را به ازای بردار تست زیر، شبیه‌سازی می‌کنیم.

$$W_1 = 1, W_3 = 1, W_6 = 0, W_2 = 1, W_7 = 0$$



شکل شماره ۶. مدار تست C17

```
{'w1': '1',
 'w2': '1',
 'w3': '1',
 'w3_1': '1',
 'w3_2': '1',
 'w6': '0',
 'w7': '0',
 'w10': '0',
 'w11': '1',
 'w11_1': '1',
 'w11_2': '1',
 'w16': '0',
 'w16_1': '0',
 'w16_2': '0',
 'w19': '1',
 'w22': '1',
 'w23': '1'}
```

شکل شماره ۷. خروجی شبیه‌سازی True-Value برای ورودی دوم

در ادامه مقادیر تمامی سیم‌ها را بصورت دستی بدست می‌آوریم:

$W_{3\_1} = 1$

$W_{3\_2} = 1$

$W_{10} = 0$

$W_{11} = 1$

$W_{11\_1} = 1$

$W_{11_2} = 1$

$W_{16} = 0$

$W_{16\_1} = 0$

$W_{16\_2} = 0$

$W_{19} = 1$

$W_{22} = 1$

$W_{23} = 1$

با مقایسه مقادیر شبیه‌سازی و مقادیر محاسبه شده، می‌توان نتیجه گرفت که شبیه‌سازی درست انجام شده است.

Deductive Fault Simulation به ازای ورودی اول

در این قسمت مدار C17 را به ازای بردار تست زیر، شبیه‌سازی می‌کنیم.

$$W_1 = 0, W_3 = 1, W_6 = 1, W_2 = 0, W_7 = 1$$

```
{'L_w1': {'w1_s1'},
 'L_w2': {'w2_s1'},
 'L_w3_1': {'w3_1_s0', 'w3_s0'},
 'L_w3': {'w3_s0'},
 'L_w3_2': {'w3_2_s0', 'w3_s0'},
 'L_w6': {'w6_s0'},
 'L_w7': {'w7_s0'},
 'L_w10': {'w10_s0', 'w1_s1'},
 'L_w11_2': {'w11_2_s1', 'w11_s1', 'w3_2_s0', 'w3_s0', 'w6_s0'},
 'L_w11_1': {'w11_1_s1', 'w11_s1', 'w3_2_s0', 'w3_s0', 'w6_s0'},
 'L_w11': {'w11_s1', 'w3_2_s0', 'w3_s0', 'w6_s0'},
 'L_w16_2': {'w16_2_s0', 'w16_s0'},
 'L_w16': {'w16_s0'},
 'L_w16_1': {'w16_1_s0', 'w16_s0'},
 'L_w19': {'w11_2_s1', 'w11_s1', 'w19_s0', 'w3_2_s0', 'w3_s0', 'w6_s0'},
 'L_w22': {'w10_s0', 'w16_1_s0', 'w16_s0', 'w1_s1', 'w22_s1'},
 'L_w23': {'w11_2_s1',
  'w11_s1',
  'w16_2_s0',
  'w16_s0',
  'w19_s0',
  'w23_s1',
  'w3_2_s0',
  'w3_s0',
  'w6_s0'}}
```

شکل شماره ۸. خروجی Deductive Fault Simulation برای ورودی اول

Deductive Fault Simulation به ازای ورودی دوم

در این قسمت مدار C17 را به ازای بردار تست زیر، شبیه‌سازی می‌کنیم.

$$W_1 = 1, W_3 = 1, W_6 = 0, W_2 = 1, W_7 = 0$$

```
{'L_w1': {'w1_s0'},
 'L_w2': {'w2_s0'},
 'L_w3_1': {'w3_1_s0', 'w3_s0'},
 'L_w3': {'w3_s0'},
 'L_w3_2': {'w3_2_s0', 'w3_s0'},
 'L_w6': {'w6_s1'},
 'L_w7': {'w7_s1'},
 'L_w10': {'w10_s1', 'w1_s0', 'w3_1_s0', 'w3_s0'},
 'L_w11_2': {'w11_2_s0', 'w11_s0', 'w6_s1'},
 'L_w11_1': {'w11_1_s0', 'w11_s0', 'w6_s1'},
 'L_w11': {'w11_s0', 'w6_s1'},
 'L_w16_2': {'w11_1_s0', 'w11_s0', 'w16_2_s1', 'w16_s1', 'w2_s0', 'w6_s1'},
 'L_w16': {'w11_1_s0', 'w11_s0', 'w16_s1', 'w2_s0', 'w6_s1'},
 'L_w16_1': {'w11_1_s0', 'w11_s0', 'w16_1_s1', 'w16_s1', 'w2_s0', 'w6_s1'},
 'L_w19': {'w19_s0', 'w7_s1'},
 'L_w22': {'w22_s0'},
 'L_w23': {'w11_1_s0',
  'w11_s0',
  'w16_2_s1',
  'w16_s1',
  'w23_s0',
  'w2_s0',
  'w6_s1'}}
```

شکل شماره ۹. خروجی Deductive Fault Simulation برای ورودی دوم

فاز دوم

تولید بردارهای تست به روش تجزیه تحلیل جدول اشکال

نحوه پیاده‌سازی

توابع کمکی این فاز درون فایل utils_fta.py وجود دارد که توضیح مربوط به هر کدام در ادامه ارائه خواهد شد.

تابع تولید کننده بردارهای تست

این تابع با ورودی گرفتن تعداد ورودی‌های مدار تمامی بردارهای تست را تولید می‌کند. ورودی این تابع به صورت یک عدد طبیعی n و خروجی آن یک لیست به طول دو به توان n می‌باشد. برای تولید بردارهای تست از تابع داخلی product درون پکیج itertools پایتون استفاده شده است که تمامی ترکیبات ممکن از یک رشته ورودی را ایجاد می‌کند.

```
def generate_bit_combinations(n):
    # Use itertools.product to generate all combinations of 0 and 1 for the given length
    bit_combinations = list(product("01", repeat=n))
    # Convert the tuples to strings
    bit_combinations = [''.join(bits) for bits in bit_combinations]
    return bit_combinations
```

تابع مرتب کننده لیست اشکال‌ها

درون پیاده‌سازی کنونی اسامی اشکال‌ها به صورت w_5_s0 و یا در صورت وجود fanout به صورت w_5_1_s0 ذخیره می‌شود. این تابع جهت نمایش مرتب اشکال‌ها نوشته شده است تا در خروجی بتوانیم لیست اشکال مربوط به هر بردار تست را به صورت منظم مشاهده کنید. برای پیاده‌سازی این تابع از چندین split بر اساس عبارت _ استفاده شده است تا مرتب سازی مناسبی صورت گیرد.

```
def fault_sorter(element):
    parts = element.split('_')
    wire_part = parts[0][1:]

    if len(parts) == 2:
        fan_part = "-1"
        stuck_part = parts[1]
    else:
        fan_part = parts[1]
        stuck_part = parts[2]

    return int(wire_part), int(fan_part), stuck_part
```

تابع ایجاد کننده جدول اشکال

این تابع تمامی بردارهای تست ممکن، اشکال‌های کشف شده توسط هر بردار تست و لیست تمامی اشکلات ممکن را ورودی می‌گیرد و سپس یک لیست دو بعدی ایجاد می‌کند. هر ستون این لیست دو بعدی بیانگیر یک اشکال و هر ردیف این لیست بیانگر یک بردار تست می‌باشد. هر سلول این جدول می‌تواند 0 یا 1 باشد. مقدار 0 به این معنی است که بردار تست اشکال مربوطه را کشف نمی‌کند و مقدار 1 یعنی کشف می‌کند.

```
def create_fault_table(input_values, all_discovered_faults, all_faults):
    fault_table = []
    # Create a 2D table where each row corresponds to an input vector, and columns indicate discovered elements
    for binary, output in zip(input_values, all_discovered_faults):
        row = [1 if element in output else 0 for element in all_faults]
        fault_table.append([binary] + row)
    # Add header row
    fault_table = [["-"] + all_faults] + fault_table
    return fault_table
```

تابع یابنده اشکال‌های کشف شونده در یک ردیف جدول اشکال

این تابع با ورودی گرفتن Index مربوط به یک ردیف جدول (تست ورودی)، تمامی Indexهای اشکال‌های کشف شده (شماره ستون) را درون یک لیست خروجی می‌دهد.

```
def find_cols_index(fault_table, row_index):
    cols_index = [index for index, value in enumerate(fault_table[row_index]) if value == 1]
    return cols_index
```

تابع یابنده بردارهای تست ضروری (Essential Tests)

این تابع یکی از قسمت‌های اصلی این پیاده‌سازی می‌باشد. درون این تابع ابتدا هر ستون جدول اشکال بررسی می‌شود. به عبارتی بررسی می‌شود که یک اشکال x توسط چند بردار تست کشف می‌شود. اگر این اشکال x تنها توسط یک بردار تست کشف شد، آنگاه آن بردار تست را ذخیره می‌کنیم. همچنین این بردار تست ذخیره شده تعدادی اشکال دیگر را نیز کشف می‌کند، پس آن‌ها را نیز ذخیره می‌کنیم تا در ادامه جدول اشکال را بتوانیم Prune کنیم. به عبارت دیگر عملیات Fault Collapsing را انجام دهیم.

```
def find_essential(fault_table):
    for col_index in range(1, len(fault_table[0])):
        # Calculate the sum of elements in the current column
        col_sum = sum(row[col_index] for row in fault_table[1:])
        if col_sum == 1:
            # Find the corresponding row index and add it to the list of rows to be removed
            row_index = [i for i, value in enumerate(fault_table[1:]) if value[col_index] == 1][0] + 1
            cols_index = find_cols_index(fault_table, row_index)
            return row_index, cols_index
    return None
```

تابع یابنده بردارهای تست مورد نیاز (Needed Tests)

این تابع مشاهده تابع بالا می‌باشد اما رویکرد آن کمی متفاوت است. در این تابع به ازای هر ردیف (بردار تست) بررسی می‌کنیم که چه تعدادی اشکال کشف می‌شود. در واقع ما به دنبال این هستیم که بردار تستی که بیشترین اشکال را کشف می‌کند پیدا کنیم. پس از پیدا کردن این بردار تست، تمامی اشکال‌هایی که توسط این بردار تست کشف می‌شوند را نیز ذخیره می‌کنیم تا در آینده آن‌ها را از جدول اشکال حذف کنیم.

```
def find_needed(fault_table):
    max_sum = float('-inf')
    row_max_sum_index = None
    for row_index, row in enumerate(fault_table[1:], start=1):
        row_sum = sum(row[1:])
        if row_sum > max_sum:
            max_sum = row_sum
            row_max_sum_index = row_index
    if max_sum < 1:
        return None
    cols_index = find_cols_index(fault_table, row_max_sum_index)
    return row_max_sum_index, cols_index
```

تابع پاک کردن ستون‌ها و ردیف جدول اشکال

این تابع یک ردیف و تعدادی ستون مشخص را از جدول اشکال حذف می‌کند و آن را کوچک تر و Collapse می‌کند.

```python
def prune_table(fault_table, row_index, cols_index):
    new_fault_table = [
        [fault_table[i][j] for j in range(len(fault_table[i])) if j not in cols_index]
        for i in range(len(fault_table)) if i not in [row_index]
    ]
    return new_fault_table
```

تابع کلی جهت انجام تمامی موارد بالا به صورت تکرار شونده

این تابع در واقع یک Wraper و یک Iterator برای انجام تمامی کارهای بالا می‌باشد. در این تابع وابسته به این که در حالت Essential هستیم (اشکالی وجود دارد که تنها توسط یک خطا کشف می‌شود) یا در حالت Needed هستیم توابع Finder را صدا می‌زند و سپس جدول اشکال را Collapse می‌کند. این کار تا زمانی ادامه می‌یابد که یا تمامی اشکالات کشف شوند یا تنها اشکال‌های غیر کشف شونده باقی بمانند.

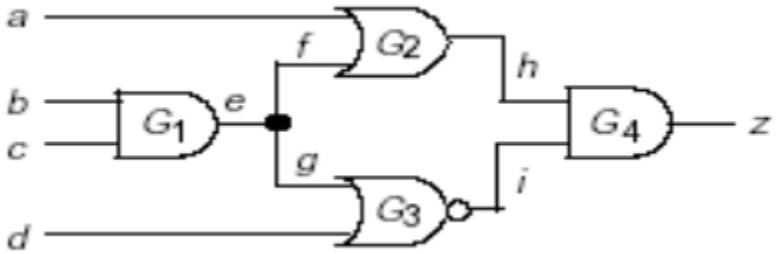
```python
def prune_iterator(fault_table, finder, verbose=0):
    finder_tests = []
    new_fault_table = fault_table
    while True:
        finder_out = finder(new_fault_table)

        if finder_out is None:
            if verbose:
                print_table(new_fault_table[1:])
            return finder_tests, new_fault_table

        row_index, cols_index = finder_out
        finder_test = new_fault_table[row_index][0]
        finder_tests.append(finder_test)

        if verbose:
            print_table(new_fault_table[1:])
            print(f"Test: {finder_test}")
            print(f"Row Num: {row_index}")
            print(f"Cols Num: {cols_index}")
            print("-" * 10)

        new_fault_table = prune_table(new_fault_table, row_index, cols_index)
```

بررسی خروجی مراحل برنامه

نمونه اول

در مثال اول یکی از مدارات موجود در اسلاید درس که برای تدریس همین مبحث مورد استفاده قرار گرفت رو مد نظر قرار می‌دهیم.

شکل این مدار به صورت زیر می‌باشد.



فایل bench مربوط به این مدار را در مسیر bench_files/ex1.bench به صورت زیر ایجاد می‌کنیم.

```
# ex1
# 4 inputs
# 1 outputs
# 0 inverter
# 4 gates ( 2 ANDs + 1 NORs + 1 ORs)

INPUT(1)
INPUT(2)
INPUT(3)
INPUT(4)

OUTPUT(10)

5 = AND(2, 3)
8 = OR(1, 5)
9 = NOR(4, 5)
10 = AND(8, 9)
```

با توجه به پیاده‌سازی فاز اول مدار را می‌خوانیم و تمامی بردارهای تست و تمامی اشکالات را ایجاد می‌کنیم.

Process Circuit and Tests

```
circuit_data = process_circuit_file(bench_filepath)
input_list, gate_list, fanout_list, output_list = split_circuit(circuit_data)
wires = create_wires(circuit_data)
input_wires = create_input_wires(input_list)
gates_output_wires = create_gates_output_wires(gate_list)
```

```
input_values = generate_bit_combinations(n=len(input_list))
```

```
U_set = create_union_set(wires)
all_faults = list(U_set)
all_faults = sorted(all_faults, key=fault_sorter)
```

سپس به ازای تمامی بردارهای تست ممکن عملیات شبیه‌سازی اشکال را با روش Deductive Fault Simulation انجام می‌دهیم تا تمامی اشکالات قابل کشف به ازای هر ورودی به دست آید.

Fault Simulation

```
all_discovered_faults = []
for input_value in input_values:
    input_vector = {key: value for key, value in zip(input_list, list(input_value))}

    simulation_dict = create_simulation_list(wires)
    tvs = true_value_simulation(
        input_vector,
        simulation_dict,
        fanout_list,
        gate_list,
        input_wires,
        gates_output_wires
    )

    La_dict = init_fault_list(wires, tvs)
    dfs = deductive_fault_simulation(
        La_dict,
        input_list,
        fanout_list,
        gates_output_wires,
        gate_list,
        tvs,
        U_set
    )
```

جدول اشکال را با توجه به تابعی که قبلا توضیح داده شد ایجاد می‌کنیم.

Create Fault Table

Add Code Cell | Add Markdown Cell

```
fault_table = create_fault_table(input_values, all_discovered_faults, all_faults)
```

```
df = pd.DataFrame(fault_table[1:], columns=fault_table[0])
df.to_html(fault_table_path, index=False)
df
```

برای نمایش بهتر این جدول از دو حالت DataFrame و حالت HTML استفاده می‌کنیم که خروجی هر کدام در ادامه آورده شده است.

خروجی DataFrame:

-	w1_s0	w1_s1	w2_s0	w2_s1	w3_s0	w3_s1	w4_s0	w4_s1	w5_s0	w5_s1	w5_1_s0	w5_1_s1	w5_2_s0
0	0000	0	1	0	0	0	0	0	0	0	0	0	1
1	0001	0	0	0	0	0	0	0	0	0	0	0	0
2	0010	0	1	0	0	0	0	0	0	0	0	0	1
3	0011	0	0	0	0	0	0	0	0	0	0	0	0
4	0100	0	1	0	0	0	0	0	0	0	0	0	1
5	0101	0	0	0	0	0	0	0	0	0	0	0	0
6	0110	0	0	0	0	0	0	0	0	0	0	0	0
7	0111	0	0	0	0	0	0	0	0	0	0	0	0
8	1000	1	0	0	0	0	0	0	1	0	1	0	0
9	1001	0	0	0	0	0	0	1	0	0	0	0	0
10	1010	1	0	0	1	0	0	0	1	0	1	0	0
11	1011	0	0	0	0	0	0	1	0	0	0	0	0
12	1100	1	0	0	0	0	1	0	1	0	1	0	0
13	1101	0	0	0	0	0	0	1	0	0	0	0	0
14	1110	0	0	1	0	1	0	0	0	1	0	0	0
15	1111	0	0	0	0	0	0	0	0	0	0	0	0

16 rows × 21 columns

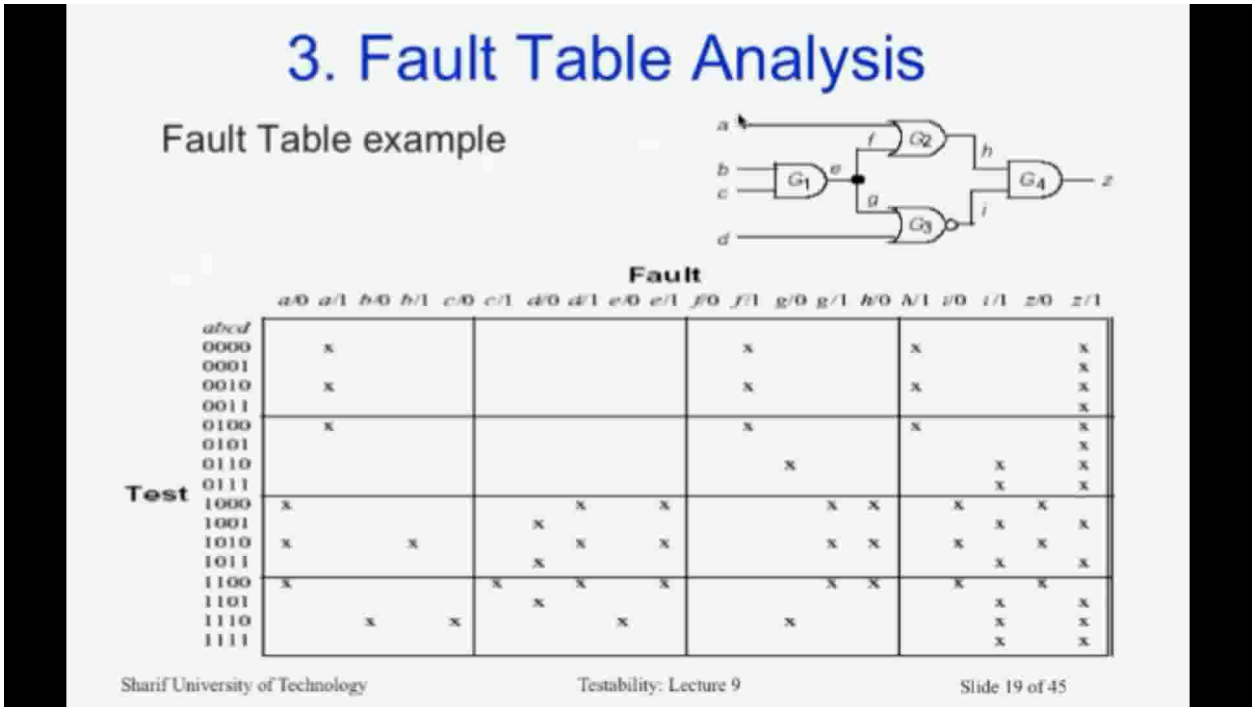
خروجی HTML:

این نمایش درون فایل fault_tables/ex1.html ذخیره شده است.

-	w1_s0	w1_s1	w2_s0	w2_s1	w3_s0	w3_s1	w4_s0	w4_s1	w5_s0	w5_s1	w5_1_s0	w5_1_s1	w5_2_s0	w5_2_s1	w8_s0	w8_s1	w9_s0	w9_s1	w10_s0	w10_s1
0000	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1
0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0010	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1
0011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0100	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1
0101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0110	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1
0111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
1000	1	0	0	0	0	0	0	1	0	1	0	0	0	1	1	0	1	0	1	0
1001	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1
1010	1	0	0	1	0	0	0	1	0	1	0	0	0	1	1	0	1	0	1	0
1011	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1
1100	1	0	0	0	0	1	0	1	0	1	0	0	0	1	1	0	1	0	1	0
1101	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1
1110	0	0	1	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	0	1
1111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

مقایسه با خروجی اسلاید

همانطور که مشاهده می‌شود این جدول دقیقا مطابق جدول درون اسلاید می‌باشد.



به دست آوردن تست‌های Essential

مرحله اول:

['0000', 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]

['0001', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

['0010', 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]

['0011', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

['0100', 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]

['0101', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

['0110', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1]

['0111', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1]

['1000', 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0]

['1001', 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1]

['1010', 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0]

['1011', 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1]

['1100', 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0]

['1101', 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1]

['1110', 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1]

['1111', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1]

Test: 1110

Row Num: 15

Cols Num: [3, 5, 9, 13, 18, 20]

مرحله دوم:

['0000', 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0]

['0001', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

['0010', 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0]

['0011', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

['0100', 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0]

['0101', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

['0110', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

['0111', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

['1000', 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1]

['1001', 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

['1010', 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1]

['1011', 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

['1100', 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1]

['1101', 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

['1111', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Test: 1010

Row Num: 11

Cols Num: [1, 3, 6, 7, 10, 11, 13, 14]

مرحله سوم:

['0000', 1, 0, 0, 0, 1, 1]

['0001', 0, 0, 0, 0, 0, 0]

['0010', 1, 0, 0, 0, 1, 1]

['0011', 0, 0, 0, 0, 0, 0]

['0100', 1, 0, 0, 0, 1, 1]

['0101', 0, 0, 0, 0, 0, 0]

['0110', 0, 0, 0, 0, 0, 0]

['0111', 0, 0, 0, 0, 0, 0]

['1000', 0, 0, 0, 0, 0, 0]

['1001', 0, 0, 1, 0, 0, 0]

['1011', 0, 0, 1, 0, 0, 0]

['1100', 0, 1, 0, 0, 0, 0]

['1101', 0, 0, 1, 0, 0, 0]

['1111', 0, 0, 0, 0, 0, 0]

Test: 1100

Row Num: 12

Cols Num: [2]

مرحله چهارم:

['0000', 1, 0, 0, 1, 1]

['0001', 0, 0, 0, 0, 0]

['0010', 1, 0, 0, 1, 1]

['0011', 0, 0, 0, 0, 0]

['0100', 1, 0, 0, 1, 1]

['0101', 0, 0, 0, 0, 0]

['0110', 0, 0, 0, 0, 0]

['0111', 0, 0, 0, 0, 0]

['1000', 0, 0, 0, 0, 0]

['1001', 0, 1, 0, 0, 0]

['1011', 0, 1, 0, 0, 0]

['1101', 0, 1, 0, 0, 0]

['1111', 0, 0, 0, 0, 0]

با توجه به این که دیگر بردار تست Essential وجود ندارد وارد فاز Needed می‌شویم.

به دست آوردن تست‌های Needed

مرحله اول:

['0000', 1, 0, 0, 1, 1]

['0001', 0, 0, 0, 0, 0]

['0010', 1, 0, 0, 1, 1]

['0011', 0, 0, 0, 0, 0]

['0100', 1, 0, 0, 1, 1]

['0101', 0, 0, 0, 0, 0]

['0110', 0, 0, 0, 0, 0]

['0111', 0, 0, 0, 0, 0]

['1000', 0, 0, 0, 0, 0]

['1001', 0, 1, 0, 0, 0]

['1011', 0, 1, 0, 0, 0]

['1101', 0, 1, 0, 0, 0]

['1111', 0, 0, 0, 0, 0]

Test: 0000

Row Num: 1

Cols Num: [1, 4, 5]

مرحله دوم:

['0001', 0, 0]

['0010', 0, 0]

['0011', 0, 0]

['0100', 0, 0]

['0101', 0, 0]

['0110', 0, 0]

['0111', 0, 0]

['1000', 0, 0]

['1001', 1, 0]

['1011', 1, 0]

['1101', 1, 0]

['1111', 0, 0]

Test: 1001

Row Num: 9

Cols Num: [1]

مرحله سوم: در این مرحله تنها یک اشکال باقی مانده که توسط هیچ برداری کشف نمی‌شود. بنابراین الگوریتم پایان می‌یابد.

['0001', 0]

['0010', 0]

['0011', 0]

['0100', 0]

['0101', 0]

['0110', 0]

['0111', 0]

['1000', 0]

['1011', 0]

['1101', 0]

['1111', 0]

خروجی نهایی:

Essential Tests

1110: ['w2_s0', 'w3_s0', 'w5_s0', 'w5_2_s0', 'w9_s1', 'w10_s1']

1010: ['w1_s0', 'w2_s1', 'w4_s1', 'w5_s1', 'w5_2_s1', 'w8_s0', 'w9_s0', 'w10_s0']

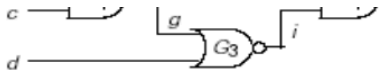
1100: ['w1_s0', 'w3_s1', 'w4_s1', 'w5_s1', 'w5_2_s1', 'w8_s0', 'w9_s0', 'w10_s0']

Needed Tests

0000: ['w1_s1', 'w5_1_s1', 'w8_s1', 'w10_s1']

1001: ['w4_s0', 'w9_s1', 'w10_s1']

این خروجی مشابه خروجی درون اسلاید است.



Representative Fault

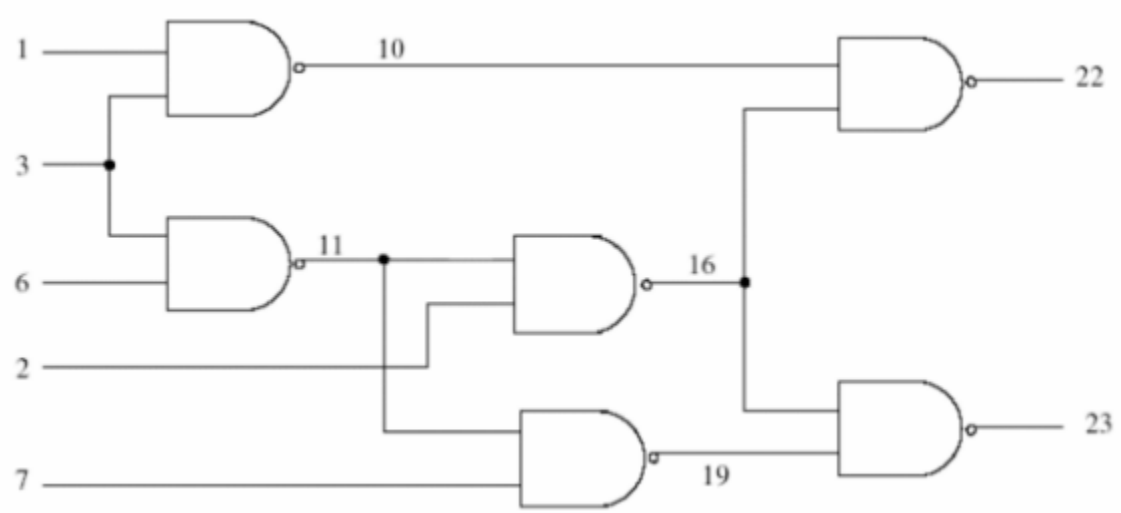
abcd	*a*/0	*a*/1	*b*/0	*b*/1	*c*/1	*d*/0	*g*/0	*i*/1	*z*/1	
0000		x							x	
0001									x	
0010		x							x	
0011									x	
0100		x					x		x	
0101									x	
0110								x	x	
0111								x	x	
1000	x									
1001						x		x	x	
1010	x			x						Essential
1011						x		x	x	
1100	x				x					Essential
1101						x		x	x	
1110			x				x	x	x	Essential
1111								x	x	
	*		*	*	*		*	*	*	

Reduced fault table after "fault collapsing"

Conclusion: 5 tests are needed, of which, 3 are essential.

نمونه دوم

از فایل مرجع c17.bench استفاده خواهیم کرد.



جدول اشکال

درون فایل fault_tables/c17.html موجود است.

-	w1_s0	w1_s1	w2_s0	w2_s1	w3_s0	w3_s1	w3_1_s0	w3_1_s1	w3_2_s0	w3_2_s1	w6_s0	w6_s1	w7_s0	w7_s1	w10_s0	w10_s1	w11_s0	w11_s1	w11_1_s0	w11_1_s1	w11_2_s0	w11_2_s1	w16_s0	w16_s1	w16_1_s0	w16_1_s1	w16_2_s0	w16_2_s1	w19_s0	w19_s1	w22_s0	w22_s1	w23_s0	w23_s1
00000	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0	1	0	1
00001	0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0
00010	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0	1	0	1
00011	0	0	0	1	0	1	0	0	0	1	0	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0
00100	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0	1	0	1
00101	0	1	0	1	0	0	0	0	0	0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0
00110	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0	1	0	1
00111	0	1	0	1	0	0	0	1	0	1	0	0	0	1	1	0	0	1	0	0	0	1	1	0	1	0	1	0	1	0	0	1	0	1
01000	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0
01001	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	1	0	1	0	1
01010	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0
01011	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0
01100	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0
01101	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0
01110	0	1	0	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1	0	0	1	0	1
01111	0	1	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	1	0	1	0	1	1	0	1	0	1	0	1	0	0	1	0	1
10000	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1
10001	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0
10010	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1
10011	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0	0	0	1	0	1	1	0
10100	1	0	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1
10101	1	0	0	0	1	0	1	0	0	0	0	1	1	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	1	0	1	0	1
10110	1	0	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	1
10111	1	0	0	0	1	0	1	0	1	0	1	0	0	0	0	1	0	1	0	0	0	0	1	1	0	0	0	0	1	0	0	1	0	1
11000	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0
11001	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	1	0	1	0	1
11010	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0
11011	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0
11100	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	1	0	1	0	1
11101	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	1	0	1	0	1
11110	1	0	0	0	1	0	1	0	1	0	1	0	0	0	0	1	0	1	0	1	0	0	1	0	0	0	1	0	1	0	1	0	0	1
11111	1	0	0	0	1	0	1	0	1	0	1	0	0	0	0	1	0	1	0	1	0	1	1	0	0	0	1	0	1	0	1	0	0	1

به دست آوردن بردار تست Essential

هیچ برداری وجود ندارد زیر در هر ستون حداقل 2 مقدار 1 داریم.

به دست آوردن بردار تست Needed

با توجه به بزرگ بودن مدار از نشان دادن مراحل خودداری می‌کنیم.

خروجی نهایی به صورت زیر است.

Needed Tests

01111: ['w1_s1', 'w3_s0', 'w3_2_s0', 'w6_s0', 'w10_s0', 'w11_s1', 'w11_1_s1', 'w11_2_s1', 'w16_s0', 'w16_1_s0', 'w16_2_s0', 'w19_s0', 'w22_s1', 'w23_s1']

01010: ['w2_s0', 'w3_s1', 'w3_2_s1', 'w11_s0', 'w11_1_s0', 'w16_s1', 'w16_1_s1', 'w16_2_s1', 'w22_s0', 'w23_s0']

10101: ['w1_s0', 'w3_s0', 'w3_1_s0', 'w6_s1', 'w7_s0', 'w10_s1', 'w11_s0', 'w11_2_s0', 'w19_s1', 'w22_s0', 'w23_s0']

10000: ['w2_s1', 'w3_s1', 'w3_1_s1', 'w7_s1', 'w10_s0', 'w16_s0', 'w16_1_s0', 'w16_2_s0', 'w19_s0', 'w22_s1', 'w23_s1']

تنها با همین 4 بردار تست می‌توانیم به Fault Coverage کامل و 100 درصدی برسیم.

```
1  n_total_faults = len(wires) * 2
2  fault_coverage = len(fault_discovery_set) / n_total_faults
3  print(f"Fault Coverage: {fault_coverage * 100} %")

   Fault Coverage: 100.0 %
```