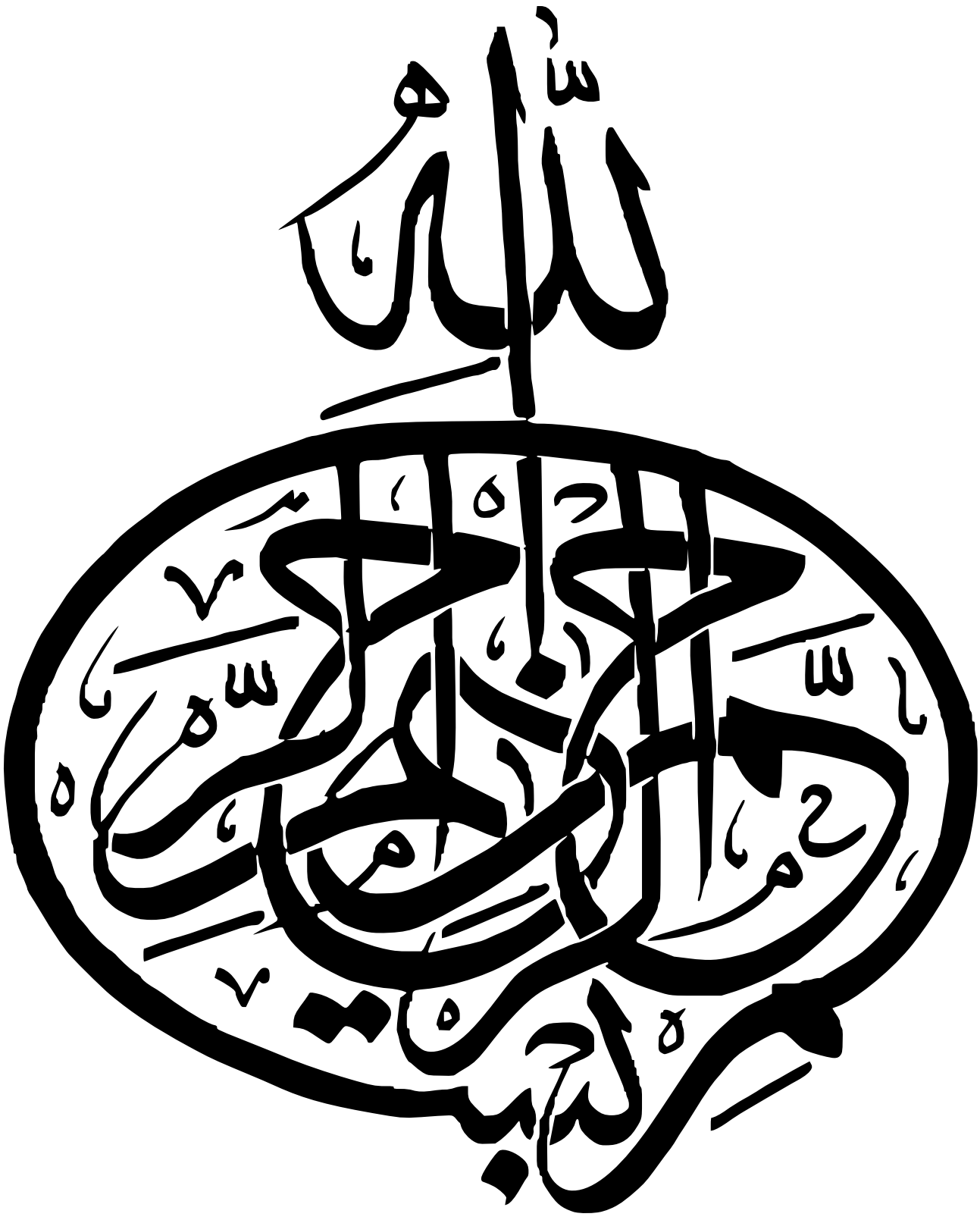


دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

درس آزمون پذیری
مهندسی کامپیوتر

گزارش پروژه اول

نگارش
سیدمحمد روزگار - ۴۰۱۲۰۳۳۸۲

استاد درس
دکتر شاهین حسابی

آبان ۱۴۰۲

فهرست مطالب

۱ مقدمه و شرح مسئله

در این گزارش قصد داریم به بررسی پروژه اول درس آزمون پذیری بپردازیم. در این پروژه فایل هایی با پسوند bench که حاوی اطلاعاتی در مورد ورودی، خروجی و همچنین گیت های مدار است را خوانده و در ادامه دو گام اصلی را انجام می دهیم: ۱. در گام اول یک ورودی که می تواند مقادیر ۰، ۱ ، U و Z هستند را به مدار اعمال می کنیم و مقادیر نت ها و خروجی ها را مشخص می کنیم. ۲. در گام بعدی یک ورودی که می توانند مقادیر باینری ۰ و ۱ هستند را به مدار تزریق می کنیم و اشکالات مدار که از نوع stack-at هستند را بر اساس الگوریتم استنتاجی [۱] و قواعد مجموعه ها در هر نت و همچنین خروجی های مدار، مشخص می کنیم.

۲ گرامر های فایل bench

پیش از آنکه به بررسی خواندن ورودی ها و تزریق آن ها به شبکه کنیم، نیاز است به بررسی گرامر های فایل های bench بپردازیم. برای نمونه فایل c17.bench به شکل زیر است:

```
1  # c17
2  # 5 inputs
3  # 2 outputs
4  # 0 inverter
5  # 6 gates ( 6 NANDs )
6
7  INPUT(1)
8  INPUT(2)
9  INPUT(3)
10 INPUT(6)
11 INPUT(7)
12
13 OUTPUT(22)
14 OUTPUT(23)
15
16 10 = NAND(1, 3)
17 11 = NAND(3, 6)
18 16 = NAND(2, 11)
19 19 = NAND(11, 7)
20 22 = NAND(10, 16)
21 23 = NAND(16, 19)
```

در این پروژه، برای این که چنین فایلی را پردازش [۲] کنیم، از کتابخانه **pyparsing** استفاده می کنیم. گرامرهای فایل های bench را می توان به شکل زیر تعریف کرد:

[1] Deductive algorithm
[2] parse

```
1  IDGrammer = pp.Word(pp.nums)
2
3
4  @IDGrammer.set_parse_action
5  def set_ID_action(results: pp.ParseResults):
6      return results[0]
7
8
9  IDsGrammer = IDGrammer + pp.OneOrMore(pp.Suppress(',') + IDGrammer)
10
11 InputGrammer = pp.Suppress(
12     'INPUT') + pp.Suppress('(') + IDGrammer + pp.Suppress(')')
13
14 OutputGrammer = pp.Suppress(
15     'OUTPUT') + pp.Suppress('(') + IDGrammer + pp.Suppress(')')
16
17 BufferGrammer = IDGrammer + pp.Suppress('=') + pp.Suppress('BUFF') + \
18     pp.Suppress('(') + IDGrammer + pp.Suppress(')')
19
20 NotGrammer = IDGrammer + pp.Suppress('=') + pp.Suppress('NOT') + \
21     pp.Suppress('(') + IDGrammer + pp.Suppress(')')
22
23 AndGrammer = IDGrammer + pp.Suppress('=') + pp.Suppress('AND') + \
24     pp.Suppress('(') + IDsGrammer + pp.Suppress(')')
25
26 NandGrammer = IDGrammer + pp.Suppress('=') + pp.Suppress('NAND') + \
27     pp.Suppress('(') + IDsGrammer + pp.Suppress(')')
28
29 OrGrammer = IDGrammer + pp.Suppress('=') + pp.Suppress('OR') + \
30     pp.Suppress('(') + IDsGrammer + pp.Suppress(')')
31
32 NorGrammer = IDGrammer + pp.Suppress('=') + pp.Suppress('NOR') + \
33     pp.Suppress('(') + IDsGrammer + pp.Suppress(')')
34
35 XorGrammer = IDGrammer + pp.Suppress('=') + pp.Suppress('XOR') + \
36     pp.Suppress('(') + IDsGrammer + pp.Suppress(')')
37
38 XnorGrammer = IDGrammer + pp.Suppress('=') + pp.Suppress('XNOR') + \
39     pp.Suppress('(') + IDsGrammer + pp.Suppress(')')
```

در واقع به ازای هر گرامر موجود، در خروجی یک لیست از اعداد برای ما تولید می کند و آن را توسط تابع دیگری که مسئولیت ساخت

شبکه را دارد، تزریق می کنیم.

۳ واحد های منطقی و محاسبات مربوطه بر روی آن ها

۱.۳ گیت ها و سیم ها

هر مدار منطقی موجود اعم از فایل های bench. که در پروژه موجود است، حاوی دو عنصر کلیدی هستند: ۱. **گیت های منطقی** : در واقع گیت های منطقی جزء جدایی ناپذیر یک مدار منطقی به شمار می آیند. در این پروژه هر گیت منطقی می تواند یکی از حالات AND, NAND, OR, NOR, XOR, XNOR, FANOUT باشد. ۲. **سیم ها** : که به عنوان پلی بین گیت های منطقی به حساب می آیند. ، می باشد.

از آنجایی که در فایل های bench فقط در مورد گیت ها و ورودی های آن صحبت کرده، لذا نیاز است پس از اینکه گیت های منطقی را از ورودی خواندیم، در ادامه گیت های FANOUT را تشکیل دهیم. پس از چنین روندی نیاز است از یک ساختار داده WIRE جهت اتصال بین گیت ها استفاده کنیم. اگر بخواهیم خیلی جزئی به بررسی اینکه چگونه گیت های FANOUT را تشکیل می دهیم، صحبت کنیم، این روند به این صورت است که اگر یک گیت دارای بیش از چند خروجی داشته باشد، آن گیت را به یک گیت FANOUT متصل می کنیم و خروجی های آن گیت را به عنوان خروجی های FANOUT در نظر می گیریم.

کدی که برای هر گیت موجود است، به شکل زیر است. البته توجه داشته باشید که این روند برای هر گیت تقریبا یکسان است و تنها در تابع _specific_validation متفاوت هستند و این تابع مسئولیت این را دارد که یک اعتبار سنجی برای تعداد ورودی و خروجی های گیت داشته باشد:

```
1  class Gate:
2      def __init__(self, id: str) -> None:
3          self.__id: str = id
4          self.__value: str = LogicValueEnum.UNKNOWN.value
5          self.__has_set_value: bool = False
6          self.__level: int = -1
7
8          self.__input_wires: list[Wire] = []
9          self.__output_wires: list[Wire] = []
10
11     @property
12     def id(self) -> str:
13         return self.__id
14
15     @property
16     def value(self) -> str:
17         return self.__value
18
19     @value.setter
20     def value(self, value_: str) -> None:
21         assert value_ in LogicValueEnum.list_values()
22
23         self.__value = value_
```

```
24          self.has_set_value = True
25
26      @property
27      def has_set_value(self) -> bool:
28          return self.__has_set_value
29
30      @has_set_value.setter
31      def has_set_value(self, has_set_value_: bool) -> None:
32          self.__has_set_value = has_set_value_
33
34      @property
35      def level(self) -> int:
36          return self.__level
37
38      @level.setter
39      def level(self, level_: int) -> None:
40          assert level_ >= 0
41
42          self.__level = level_
43
44      @property
45      def input_wires(self) -> list[Wire]:
46          return self.__input_wires
47
48      @input_wires.setter
49      def output_wires(self, input_wires_: list[Wire]) -> None:
50          self.__input_wires = input_wires_
51
52      def add_input_wire(self, input_wire_: Wire) -> None:
53          assert input_wire_.output_gate == self
54          assert input_wire_ not in self.input_wires
55
56          self.input_wires.append(input_wire_)
57
58      @property
59      def __input_wires_values(self) -> list[str]:
60          return [
61              input_wire.value for input_wire in self.input_wires
62          ]
63
64      @property
```

```
65      def output_wires(self) -> list[Wire]:
66          return self.__output_wires
67
68      @output_wires.setter
69      def output_wires(self, output_wires_: list[Wire]) -> None:
70          self.__output_wires = output_wires_
71
72      def add_output_wire(self, output_wire_: Wire) -> None:
73          assert output_wire_.input_gate == self
74          assert output_wire_ not in self.output_wires
75
76          self.output_wires.append(output_wire_)
77
78      @property
79      def __output_wires_values(self) -> list[str]:
80          return [
81              output_wire.value for output_wire in self.output_wires
82          ]
83
84      def _specific_validation(self) -> None:
85          """
86          This validation is different for each gate
87          """
88          pass
89
90      def __validate_before_operation(self) -> None:
91          assert self.has_set_value == False
92
93          for input_wire in self.input_wires:
94              assert input_wire.has_set_value == True
95
96          for output_wire in self.output_wires:
97              assert output_wire.has_set_value == False
98
99      def __propagate_to_output_wires(self) -> None:
100         assert self.has_set_value == True
101
102         for output_wire in self.output_wires:
103             output_wire.value = self.value
104
105     def __validate_after_operation(self) -> None:
```

```
106         assert self.has_set_value == True
107
108         for output_wire in self.output_wires:
109             assert output_wire.has_set_value == True
110
111     def set_value_and_propagate(self, value_: str = None) -> None:
112         self._specific_validation()
113
114         self.__validate_before_operation()
115
116         if value_:
117             assert value_ in LogicValueEnum.list_values()
118
119             self.value = value_
120         else:
121             operation_type = OperationTypeEnum[self.__class__.__name__.
                    replace(
122                 'Gate', '')]
123
124             self.value = LogicOperationController(
125                 input_vector=self.__input_wires_values,
126                 operation_type=operation_type
127             ).run()
128
129         self.__propagate_to_output_wires()
130
131         self.__validate_after_operation()
132
133     def reset(self) -> None:
134         self.value = LogicValueEnum.UNKNOWN.value
135         self.has_set_value = False
```

در واقع روند تولید هر گیت به این صورت است که در ابتدا مقدار هر گیت به صورت نامشخص یا UNKNOWN تعریف می شود. در ادامه پس از محاسبات مربوطه گیت بر اساس سیم های ورودی اش، خروجی گیت محاسبه و به سیم های خروجی اش منتشر می شود.
در ادامه به بررسی ساختار هر WIRE می پردازیم:

```
1  class Wire:
2      def __init__(self, input_gate, output_gate) -> None:
3          self.__id: str = None
4
5          self.__value: str = LogicValueEnum.UNKNOWN.value
6          self.__has_set_value: bool = False
```

```
7
8           assert input_gate and output_gate
9           self.__input_gate = input_gate
10          self.__output_gate = output_gate
11
12      @property
13      def id(self) -> str:
14          return self.__id
15
16      @id.setter
17      def id(self, id_: str):
18          self.__id = id_
19
20      @property
21      def value(self) -> str:
22          return self.__value
23
24      @value.setter
25      def value(self, value_: str) -> None:
26          assert value_ in LogicValueEnum.list_values()
27
28          self.__value = value_
29          self.has_set_value = True
30
31      @property
32      def has_set_value(self) -> bool:
33          return self.__has_set_value
34
35      @has_set_value.setter
36      def has_set_value(self, has_set_value_: bool) -> None:
37          self.__has_set_value = has_set_value_
38
39      @property
40      def input_gate(self):
41          return self.__input_gate
42
43      @input_gate.setter
44      def input_gate(self, input_gate_) -> None:
45          assert input_gate_
46
47          self.__input_gate = input_gate_
```

```
48
49      @property
50      def output_gate(self):
51          return self.__output_gate
52
53      @output_gate.setter
54      def set_output_gate(self, output_gate_) -> None:
55          assert output_gate_
56
57          self.__output_gate = output_gate_
58
59      @property
60      def gates(self):
61          return (
62              self.input_gate,
63              self.output_gate
64          )
65
66      def reset(self) -> None:
67          self.value = LogicValueEnum.UNKNOWN.value
68          self.has_set_value = False
```

همانطور که گفته شد، پس از اینکه گیت ها تشکیل شدند، نیاز است از طریق سیم ها یا همان WIRE ها اتصال بین آن ها برقرار شود و این امر مشابه با گیت ها، ابتدا مقادیر سیم ها نیز به صورت نامشخص یا UNKNOWN تعریف می شود و پس از اینکه مقادیر گیت محاسبه شد، خروجی آن گیت بر روی این سیم ها منتشر می شود و این سیم ها مقادیر گیت های سطح بعدی را تامین می کنند.

۲.۳ محاسبات منحصر به فرد هر گیت

حال در ادامه به بررسی محاسبات برای هر گیت می پردازیم. کد محاسبات مربوط به هر گیت به شکل زیر است:

```
1  class LogicOperation(Operation):
2      class BasicLogicOperaion(Operation):
3          @classmethod
4          def not_operation(cls, bit_1: str) -> str:
5              if bit_1 == LogicValueEnum.ZERO.value:
6                  return LogicValueEnum.ONE.value
7
8              if bit_1 == LogicValueEnum.ONE.value:
9                  return LogicValueEnum.ZERO.value
10
11             if bit_1 == LogicValueEnum.HIGH_IMPEDANCE.value:
```

```
12                 return LogicValueEnum.HIGH_IMPEDANCE.value
13
14             if bit_1 == LogicValueEnum.UNKNOWN.value:
15                 return LogicValueEnum.UNKNOWN.value
16
17         @classmethod
18         def and_operation(cls, bit_1: str, bit_2: str) -> str:
19             if bit_1 == LogicValueEnum.ZERO.value:
20                 return LogicValueEnum.ZERO.value
21
22             if bit_1 == LogicValueEnum.ONE.value:
23                 return bit_2
24
25             if bit_1 == LogicValueEnum.HIGH_IMPEDANCE.value:
26                 if bit_2 == LogicValueEnum.ZERO.value:
27                     return LogicValueEnum.ZERO.value
28
29                 if bit_2 == LogicValueEnum.ONE.value or bit_2 ==
                       LogicValueEnum.HIGH_IMPEDANCE.value:
30                     return LogicValueEnum.HIGH_IMPEDANCE.value
31
32                 return LogicValueEnum.UNKNOWN.value
33
34             if bit_1 == LogicValueEnum.UNKNOWN.value:
35                 if bit_2 == LogicValueEnum.ZERO.value:
36                     return LogicValueEnum.ZERO.value
37
38                 return LogicValueEnum.UNKNOWN.value
39
40         @classmethod
41         def or_operation(cls, bit_1: str, bit_2: str) -> str:
42             if bit_1 == LogicValueEnum.ZERO.value:
43                 return bit_2
44
45             if bit_1 == LogicValueEnum.ONE.value:
46                 return LogicValueEnum.ONE.value
47
48             if bit_1 == LogicValueEnum.HIGH_IMPEDANCE.value:
49                 if bit_2 == LogicValueEnum.ZERO.value or bit_2 ==
                       LogicValueEnum.HIGH_IMPEDANCE.value:
50                     return LogicValueEnum.HIGH_IMPEDANCE.value
```

```
51
52                  if bit_2 == LogicValueEnum.ONE.value:
53                      return LogicValueEnum.ONE.value
54
55                  return LogicValueEnum.UNKNOWN.value
56
57              if bit_1 == LogicValueEnum.UNKNOWN.value:
58                  if bit_2 == LogicValueEnum.ONE.value:
59                      return LogicValueEnum.ONE.value
60
61                  return LogicValueEnum.UNKNOWN.value
62
63          @classmethod
64          def xor_operation(cls, bit_1: str, bit_2: str) -> str:
65              return cls.or_operation(
66                  bit_1=cls.and_operation(
67                      bit_1=cls.not_operation(
68                          bit_1=bit_1
69                      ),
70                      bit_2=bit_2
71                  ),
72                  bit_2=cls.and_operation(
73                      bit_1=bit_1,
74                      bit_2=cls.not_operation(
75                          bit_1=bit_2
76                      )
77                  )
78              )
79
80      @classmethod
81      def input_operation(cls, bits: list[str]) -> str:
82          assert len(bits) == 1
83
84          return None
85
86      @classmethod
87      def output_operation(cls, bits: list[str]) -> str:
88          assert len(bits) == 1
89
90          return bits[0]
91
```

```
92      @classmethod
93      def buffer_operation(cls, bits: list[str]) -> str:
94          assert len(bits) == 1
95
96          return bits[0]
97
98      @classmethod
99      def not_operation(cls, bits: list[str]) -> str:
100         assert len(bits) == 1
101
102         return cls.BasicLogicOperaion.not_operation(
103             bit_1=bits[0]
104         )
105
106     @classmethod
107     def and_operation(cls, bits: list[str]) -> str:
108         assert len(bits) >= 2
109
110         if len(bits) == 2:
111             return cls.BasicLogicOperaion.and_operation(
112                 bit_1=bits[0],
113                 bit_2=bits[1]
114             )
115
116         return cls.BasicLogicOperaion.and_operation(
117             bit_1=bits[0],
118             bit_2=cls.and_operation(bits=bits[1:])
119         )
120
121     @classmethod
122     def nand_operation(cls, bits: list[str]) -> str:
123         assert len(bits) >= 2
124
125         return cls.BasicLogicOperaion.not_operation(
126             bit_1=cls.and_operation(
127                 bits=bits
128             )
129         )
130
131     @classmethod
132     def or_operation(cls, bits: list[str]) -> str:
```

```
133         assert len(bits) >= 2
134
135         if len(bits) == 2:
136             return cls.BasicLogicOperaion.or_operation(
137                 bit_1=bits[0],
138                 bit_2=bits[1]
139             )
140
141         return cls.BasicLogicOperaion.or_operation(
142             bit_1=bits[0],
143             bit_2=cls.or_operation(bits=bits[1:])
144         )
145
146     @classmethod
147     def nor_operation(cls, bits: list[str]) -> str:
148         assert len(bits) >= 2
149
150         return cls.BasicLogicOperaion.not_operation(
151             bit_1=cls.or_operation(
152                 bits=bits
153             )
154         )
155
156     @classmethod
157     def xor_operation(cls, bits: list[str]) -> str:
158         assert len(bits) >= 2
159
160         if len(bits) == 2:
161             return cls.BasicLogicOperaion.xor_operation(
162                 bit_1=bits[0],
163                 bit_2=bits[1]
164             )
165
166         return cls.BasicLogicOperaion.xor_operation(
167             bit_1=bits[0],
168             bit_2=cls.xor_operation(bits=bits[1:])
169         )
170
171     @classmethod
172     def xnor_operation(cls, bits: list[str]) -> str:
173         assert len(bits) >= 2
```

```
174
175             return cls.BasicLogicOperaion.not_operation(
176                 bit_1=cls.xor_operation(
177                     bits=bits
178                 )
179             )
180
181         @classmethod
182         def fanout_operation(cls, bits: list[str]) -> str:
183             assert len(bits) == 1
184
185             return bits[0]
```

روند کلی این کد به صورت بازگشتی تعریف شده است. بدین معنا برای مثال گیت AND دارای n ورودی باشد، خروجی آن گیت به صورت روبرو بدست می آید: $AND(a[n]) = AND(a[۰], AND(A[۱ : n]))$

اگر فرض کنیم گیت های اصلی به صورت گیت های AND و OR و NOT هستند، گیت NAND به صورت NOT کردن AND ورودی ها بدست می آید و به صورت مشابه گیت NOR به صورت NOT کردن OR ورودی ها بدست می آید. از طرفی گیت XOR دو ورودی (حاوی ورودی های a و b) به صورت $XOR(a, b) = \overline{a} \times b + a \times \overline{b}$ بدست می آید. همچنین گیت XNOR به صورت NOT کردن XOR ورودی ها بدست می آید.

از این رو در ادامه به بررسی جدول درستی گیت های AND, OR , NOT به ازای مقادیر 0,1,Z,U می پردازیم.

جدول درستی گیت NOT به صورت زیر تعریف می شود:

NOT Operation	
a	**NOT(a)**
0	1
1	0
Z	Z
U	U

همچنین جدول درستی گیت دو ورودی AND به صورت زیر تعریف می شود:

AND Operation		
a	**b**	**AND(a,b)**
0	0	0
0	1	0
0	Z	0
0	U	0
1	0	0
1	1	1
1	Z	Z
1	U	U
Z	0	0
Z	1	Z
Z	Z	Z
Z	U	U
U	0	0
U	1	U
U	Z	U
U	U	U

همچنین جدول درستی گیت OR دو ورودی به شکل زیر است:

OR Operation		
a	**b**	**OR(a,b)**
0	0	0
0	1	1
0	Z	Z
0	U	U
1	0	1
1	1	1
1	Z	1
1	U	1
Z	0	Z
Z	1	1
Z	Z	Z
Z	U	U
U	0	U
U	1	1
U	Z	U
U	U	U

همانطور که اشاره شد، سایر گیت ها را می توان از طریق این سه گیت اصلی NOT, AND , OR بدست آورد و همچنین برای گیت های بیش از دو ورودی می توان از مکانیزمی بازگشتی گفته شده، استفاده کرد.

در ادامه نیاز است ورودی ها را از فایل خوانده و به شبکه گیت ها تزریق کنیم و در هر گام خروجی گیت های آن سطح را محاسبه کنیم تا در نهایت به خروجی برسیم. به پراکندگی کد این قسمت، از آوردن کد آن صرف نظر کرده و تنها به همین توضیحات مختصر بسنده می کنیم.

۴ شبیه ساز اشکال استنتاجی

پس از اینکه گرامر های مربوط به فایل های bench خوانده شد و در ادامه گیت ها تشکیل و اتصال سیم ها با گیت ها برقرار شد و ورودی ها به سطح صفر مدار (یعنی پایانه های ورودی) تزریق می شود و در هر گام محاسبات روی گیت ها انجام شد و روی سیم های خروجی آن منتشر شد، نیاز است الگوریتم استنتاجی را اعمال کنیم.

بر اساس صورت پروژه نیاز است که این الگوریتم را تنها برای منطق دو مقداری ۰ و ۱ اعمال کنیم و از منطق چهار مقداری که برای مرحله قبل بود صرف نظر کنیم.

اگر بخواهیم این الگوریتم شبیه سازی اشکال استنتاجی را خلاصه کنیم، می توان به شکل زیر عمل کنیم:

۱. **گیت BUFFER و NOT و FANOUT** : در واقع این گیت ها هر اشکال ورودی خود را به خروجی (ها)ی خود منتشر می کنند و در نهایت با فالت مربوط به سیم خروجی خود اجتماع می گیرند.

۲. **گیت AND و NAND** : برای این گیت ها به این صورت است که اگر مقدار خود گیت ۱ (۰ برای (NAND باشد، هر کدام از ورودی ها تغییر کند، خروجی تغییر می کند لذا لیست اشکال نهایی به صورت اجتماع لیست اشکال ورودی ها تعریف می شود. ولی اگر مقدار خود گیت ۰ (۱ برای (NAND باشد، برای تغییر خروجی ، نیاز است تمامی ورودی های صفر به یک تغییر کنند به شرطی که این تغییر روی سایر ورودی های یک تاثیری نداشته باشد. لذا لیست اشکال خروجی نهایی به صورت اشتراک لیست اشکال ورودی های صفر منهای اجتماع لیست اشکال ورودی های یک، تعریف می شود. البته باید توجه داشت که در نهایت لیست اشکال خود سیم خروجی را نیز به لیست اشکال های آن سیم اضافه کنیم.

۳. **گیت OR و NOR** : برای این گیت ها در صورتی که مقدار خود گیت ۰ (۱ برای (NOR باشد، هر کدام از ورودی های گیت تغییر کنند، خروجی تغییر می کند. لذا لیست اشکال نهایی به صورت اجتماع اشکال های ورودی تعریف می شود. ولی اگر مقدار خود گیت ۱ (۰ برای (NOR باشد، برای تغییر خروجی، نیاز است تمام ورودی های یک به صفر تبدیل شوند به شرطی که این تغییر روی سایر ورودی های صفر تاثیری نداشته باشد. لذا لیست اشکال خروجی نهایی به صورت اشتراک لیست اشکال ورودی های یک منهای لیست اشکال ورودی های صفر تعریف می شود. البته همانند حالت قبل ، باید توجه داشت که در نهایت لیست اشکال خود سیم خروجی را نیز به لیست اشکال های آن سیم اضافه کنیم.

۴. **گیت XOR و XNOR** : لیست اشکال این نوع گیت ها به صورت دیگری برخلاف حالات قبل، تعریف می شود. لیست اشکال این گیت ها به این صورت که تعریف می شود که اگر تعداد فردی از ورودی ها تغییر کند، سبب تغییر خروجی می شود. لذا نیاز است تعداد حالات فرد ورودی را اشتراک و در ادامه با سایر ورودی های صفر که اجتماع گرفته ایم ، از هم کم کنیم. البته همانند حالت قبل ، باید توجه داشت که در نهایت لیست اشکال خود سیم خروجی را نیز به لیست اشکال های آن سیم اضافه کنیم.

در ادامه کد مربوط به **Deductive Fault Simulation** ، آورده شده است:

```
1  class FaultSimulationDeductiveOperation(Operation):
2      class ValidationFaultSimulationDeductiveOperation(Operation):
3          @classmethod
4          def __value_validation(cls, gate: Gate) -> None:
5              assert gate.value in LogicValueBinaryEnum.list_values()
6
7              for input_wire in gate.input_wires:
```

```
8                 assert input_wire.value in LogicValueBinaryEnum.
                      list_values()
9
10            for output_wire in gate.output_wires:
11                assert output_wire.value in LogicValueBinaryEnum.
                      list_values()
12
13        @classmethod
14        def input_operation(cls, gate: InputGate, all_fault_dict: dict[
              Wire, set[str]]) -> None:
15            cls.__value_validation(
16                gate=gate
17            )
18
19            assert len(gate.input_wires) == 0
20            assert len(gate.output_wires) == 1
21
22            assert gate.output_wires[0] not in all_fault_dict
23
24        @classmethod
25        def output_operation(cls, gate: OutputGate, all_fault_dict: dict[
              Wire, set[str]]) -> None:
26            cls.__value_validation(
27                gate=gate
28            )
29
30            assert len(gate.input_wires) == 1
31            assert len(gate.output_wires) == 0
32
33            assert gate.input_wires[0] in all_fault_dict
34
35        @classmethod
36        def buffer_operation(cls, gate: BufferGate, all_fault_dict: dict[
              Wire, set[str]]) -> None:
37            cls.__value_validation(
38                gate=gate
39            )
40
41            assert len(gate.input_wires) == 1
42            assert len(gate.output_wires) == 1
43
```

```
44              assert gate.input_wires[0] in all_fault_dict
45              assert gate.output_wires[0] not in all_fault_dict
46
47          @classmethod
48          def not_operation(cls, gate: NotGate, all_fault_dict: dict[Wire,
                set[str]]) -> None:
49              cls.__value_validation(
50                  gate=gate
51              )
52
53              assert len(gate.input_wires) == 1
54              assert len(gate.output_wires) == 1
55
56              assert gate.input_wires[0] in all_fault_dict
57              assert gate.output_wires[0] not in all_fault_dict
58
59          @classmethod
60          def and_operation(cls, gate: AndGate, all_fault_dict: dict[Wire,
                set[str]]) -> None:
61              cls.__value_validation(
62                  gate=gate
63              )
64
65              assert len(gate.input_wires) >= 2
66              assert len(gate.output_wires) == 1
67
68              for input_wire in gate.input_wires:
69                  assert input_wire in all_fault_dict
70
71              assert gate.output_wires[0] not in all_fault_dict
72
73          @classmethod
74          def nand_operation(cls, gate: NandGate, all_fault_dict: dict[Wire
                , set[str]]) -> None:
75              cls.__value_validation(
76                  gate=gate
77              )
78
79              assert len(gate.input_wires) >= 2
80              assert len(gate.output_wires) == 1
81
```

```
82              for input_wire in gate.input_wires:
83                  assert input_wire in all_fault_dict
84
85              assert gate.output_wires[0] not in all_fault_dict
86
87          @classmethod
88          def or_operation(cls, gate: OrGate, all_fault_dict: dict[Wire,
                set[str]]) -> None:
89              cls.__value_validation(
90                  gate=gate
91              )
92
93              assert len(gate.input_wires) >= 2
94              assert len(gate.output_wires) == 1
95
96              for input_wire in gate.input_wires:
97                  assert input_wire in all_fault_dict
98
99              assert gate.output_wires[0] not in all_fault_dict
100
101         @classmethod
102         def nor_operation(cls, gate: NorGate, all_fault_dict: dict[Wire,
                set[str]]) -> None:
103             cls.__value_validation(
104                 gate=gate
105             )
106
107             assert len(gate.input_wires) >= 2
108             assert len(gate.output_wires) == 1
109
110             for input_wire in gate.input_wires:
111                 assert input_wire in all_fault_dict
112
113             assert gate.output_wires[0] not in all_fault_dict
114
115         @classmethod
116         def xor_operation(cls, gate: XorGate, all_fault_dict: dict[Wire,
                set[str]]) -> None:
117             cls.__value_validation(
118                 gate=gate
119             )
```

```
120
121             assert len(gate.input_wires) >= 2
122             assert len(gate.output_wires) == 1
123
124             for input_wire in gate.input_wires:
125                 assert input_wire in all_fault_dict
126
127             assert gate.output_wires[0] not in all_fault_dict
128
129         @classmethod
130         def xnor_operation(cls, gate: XnorGate, all_fault_dict: dict[Wire
                , set[str]]) -> None:
131             cls.__value_validation(
132                 gate=gate
133             )
134
135             assert len(gate.input_wires) >= 2
136             assert len(gate.output_wires) == 1
137
138             for input_wire in gate.input_wires:
139                 assert input_wire in all_fault_dict
140
141             assert gate.output_wires[0] not in all_fault_dict
142
143         @classmethod
144         def fanout_operation(cls, gate: XorGate, all_fault_dict: dict[
               Wire, set[str]]) -> None:
145             cls.__value_validation(
146                 gate=gate
147             )
148
149             assert len(gate.input_wires) == 1
150             assert len(gate.output_wires) >= 2
151
152             assert gate.input_wires[0] in all_fault_dict
153
154             for output_wire in gate.output_wires:
155                 assert output_wire not in all_fault_dict
156
157     @classmethod
158     def input_operation(cls, gate: InputGate, all_fault_dict: dict[Wire,
```

```
        set[str]]) -> None:
159         cls.ValidationFaultSimulationDeductiveOperation.input_operation(
160             gate=gate,
161             all_fault_dict=all_fault_dict
162         )
163 
164         all_fault_dict[gate.output_wires[0]] = set()
165 
166         all_fault_dict[gate.output_wires[0]].add(
167             f'{gate.output_wires[0].id}_s-a-{LogicValueBinaryEnum.ONE.
                value if gate.output_wires[0].value ==
                LogicValueBinaryEnum.ZERO.value else LogicValueBinaryEnum.
                ZERO.value}'
168         )
169 
170     @classmethod
171     def output_operation(cls, gate: OutputGate, all_fault_dict: dict[Wire
        , set[str]]) -> None:
172         cls.ValidationFaultSimulationDeductiveOperation.output_operation(
173             gate=gate,
174             all_fault_dict=all_fault_dict
175         )
176         pass
177 
178     @classmethod
179     def buffer_operation(cls, gate: BufferGate, all_fault_dict: dict[Wire
        , set[str]]) -> None:
180         cls.ValidationFaultSimulationDeductiveOperation.buffer_operation(
181             gate=gate,
182             all_fault_dict=all_fault_dict
183         )
184 
185         all_fault_dict[gate.output_wires[0]] = set()
186 
187         all_fault_dict[gate.output_wires[0]] = all_fault_dict[gate.
            output_wires[0]].union(
188             all_fault_dict[gate.input_wires[0]]
189         )
190 
191         all_fault_dict[gate.output_wires[0]].add(
192             f'{gate.output_wires[0].id}_s-a-{LogicValueBinaryEnum.ONE.
```

```
                value if gate.output_wires[0].value ==
                LogicValueBinaryEnum.ZERO.value else LogicValueBinaryEnum.
                ZERO.value}'
193         )
194
195     @classmethod
196     def not_operation(cls, gate: NotGate, all_fault_dict: dict[Wire, set[
            str]]) -> None:
197         cls.ValidationFaultSimulationDeductiveOperation.not_operation(
198             gate=gate,
199             all_fault_dict=all_fault_dict
200         )
201
202         all_fault_dict[gate.output_wires[0]] = set()
203
204         all_fault_dict[gate.output_wires[0]] = all_fault_dict[gate.
                output_wires[0]].union(
205             all_fault_dict[gate.input_wires[0]]
206         )
207
208         all_fault_dict[gate.output_wires[0]].add(
209             f'{gate.output_wires[0].id}_s-a-{LogicValueBinaryEnum.ONE.
                    value if gate.output_wires[0].value ==
                    LogicValueBinaryEnum.ZERO.value else LogicValueBinaryEnum.
                    ZERO.value}'
210         )
211
212     @classmethod
213     def and_operation(cls, gate: AndGate, all_fault_dict: dict[Wire, set[
            str]]) -> None:
214         cls.ValidationFaultSimulationDeductiveOperation.and_operation(
215             gate=gate,
216             all_fault_dict=all_fault_dict
217         )
218
219         all_fault_dict[gate.output_wires[0]] = set()
220
221         if gate.value == LogicValueBinaryEnum.ONE.value:
222             for input_wire in gate.input_wires:
223                 all_fault_dict[gate.output_wires[0]] = all_fault_dict[
224                     gate.output_wires[0]
```

```
225                 ].union(
226                     all_fault_dict[input_wire]
227                 )
228
229         else:
230             temp: set[Wire] = set()
231
232             for input_wire in gate.input_wires:
233                 if input_wire.value == LogicValueBinaryEnum.ZERO.value:
234                     if len(all_fault_dict[gate.output_wires[0]]) == 0:
235                         all_fault_dict[gate.output_wires[0]] =
                                all_fault_dict[gate.output_wires[0]].union(
236                                 all_fault_dict[input_wire]
237                             )
238                     else:
239                         all_fault_dict[gate.output_wires[0]] =
                                all_fault_dict[gate.output_wires[0]].
                                intersection(
240                                 all_fault_dict[input_wire]
241                             )
242                 else:
243                     temp.union(
244                         all_fault_dict[input_wire]
245                     )
246
247             all_fault_dict[gate.output_wires[0]
248                         ] = all_fault_dict[gate.output_wires[0]] -
                               temp
249
250         all_fault_dict[gate.output_wires[0]].add(
251             f'{gate.output_wires[0].id}_s-a-{LogicValueBinaryEnum.ONE.
                value if gate.output_wires[0].value == 
                LogicValueBinaryEnum.ZERO.value else LogicValueBinaryEnum.
                ZERO.value}'
252         )
253
254     @classmethod
255     def nand_operation(cls, gate: NandGate, all_fault_dict: dict[Wire,
            set[str]]) -> None:
256         cls.ValidationFaultSimulationDeductiveOperation.nand_operation(
257             gate=gate,
```

```
258             all_fault_dict=all_fault_dict
259         )
260
261         all_fault_dict[gate.output_wires[0]] = set()
262
263         if gate.value == LogicValueBinaryEnum.ZERO.value:
264             for input_wire in gate.input_wires:
265                 all_fault_dict[gate.output_wires[0]] = all_fault_dict[
266                     gate.output_wires[0]
267                 ].union(
268                     all_fault_dict[input_wire]
269                 )
270
271         else:
272             temp: set[Wire] = set()
273
274             for input_wire in gate.input_wires:
275                 if input_wire.value == LogicValueBinaryEnum.ZERO.value:
276                     if len(all_fault_dict[gate.output_wires[0]]) == 0:
277                         all_fault_dict[gate.output_wires[0]] =
                                all_fault_dict[gate.output_wires[0]].union(
278                                 all_fault_dict[input_wire]
279                         )
280                     else:
281                         all_fault_dict[gate.output_wires[0]] =
                                all_fault_dict[gate.output_wires[0]].
                                intersection(
282                                 all_fault_dict[input_wire]
283                         )
284                 else:
285                     temp.union(
286                         all_fault_dict[input_wire]
287                     )
288
289             all_fault_dict[gate.output_wires[0]
290                            ] = all_fault_dict[gate.output_wires[0]] -
                                   temp
291
292         all_fault_dict[gate.output_wires[0]].add(
293             f'{gate.output_wires[0].id}_s-a-{LogicValueBinaryEnum.ONE.
                    value if gate.output_wires[0].value ==
```

```
                LogicValueBinaryEnum.ZERO.value else LogicValueBinaryEnum.
                ZERO.value}'
294         )
295
296     @classmethod
297     def or_operation(cls, gate: OrGate, all_fault_dict: dict[Wire, set[
            str]]) -> None:
298         cls.ValidationFaultSimulationDeductiveOperation.or_operation(
299             gate=gate,
300             all_fault_dict=all_fault_dict
301         )
302
303         all_fault_dict[gate.output_wires[0]] = set()
304
305         if gate.value == LogicValueBinaryEnum.ZERO.value:
306             for input_wire in gate.input_wires:
307                 all_fault_dict[gate.output_wires[0]] = all_fault_dict[
308                     gate.output_wires[0]
309                 ].union(
310                     all_fault_dict[input_wire]
311                 )
312
313         else:
314             temp: set[Wire] = set()
315
316             for input_wire in gate.input_wires:
317                 if input_wire.value == LogicValueBinaryEnum.ONE.value:
318                     if len(all_fault_dict[gate.output_wires[0]]) == 0:
319                         all_fault_dict[gate.output_wires[0]] =
                                all_fault_dict[gate.output_wires[0]].union(
320                                 all_fault_dict[input_wire]
321                         )
322                     else:
323                         all_fault_dict[gate.output_wires[0]] =
                                all_fault_dict[gate.output_wires[0]].
                                intersection(
324                                 all_fault_dict[input_wire]
325                         )
326                 else:
327                     temp.union(
328                         all_fault_dict[input_wire]
```

```
329                     )
330
331             all_fault_dict[gate.output_wires[0]
332                         ] = all_fault_dict[gate.output_wires[0]] -
                                temp
333
334         all_fault_dict[gate.output_wires[0]].add(
335             f'{gate.output_wires[0].id}_s-a-{LogicValueBinaryEnum.ONE.
                value if gate.output_wires[0].value == 
                LogicValueBinaryEnum.ZERO.value else LogicValueBinaryEnum.
                ZERO.value}'
336         )
337
338     @classmethod
339     def nor_operation(cls, gate: NorGate, all_fault_dict: dict[Wire, set[
            str]]) -> None:
340         cls.ValidationFaultSimulationDeductiveOperation.nor_operation(
341             gate=gate,
342             all_fault_dict=all_fault_dict
343         )
344
345         all_fault_dict[gate.output_wires[0]] = set()
346
347         if gate.value == LogicValueBinaryEnum.ONE.value:
348             for input_wire in gate.input_wires:
349                 all_fault_dict[gate.output_wires[0]] = all_fault_dict[
350                     gate.output_wires[0]
351                 ].union(
352                     all_fault_dict[input_wire]
353                 )
354
355         else:
356             temp: set[Wire] = set()
357
358             for input_wire in gate.input_wires:
359                 if input_wire.value == LogicValueBinaryEnum.ONE.value:
360                     if len(all_fault_dict[gate.output_wires[0]]) == 0:
361                         all_fault_dict[gate.output_wires[0]] =
                                all_fault_dict[gate.output_wires[0]].union(
362                                 all_fault_dict[input_wire]
363                         )
```

```
364                     else:
365                         all_fault_dict[gate.output_wires[0]] =
                                all_fault_dict[gate.output_wires[0]].
                                intersection(
366                                 all_fault_dict[input_wire]
367                             )
368                 else:
369                     temp.union(
370                         all_fault_dict[input_wire]
371                     )
372
373             all_fault_dict[gate.output_wires[0]
374                             ] = all_fault_dict[gate.output_wires[0]] -
                                    temp
375
376         all_fault_dict[gate.output_wires[0]].add(
377             f'{gate.output_wires[0].id}_s-a-{LogicValueBinaryEnum.ONE.
                value if gate.output_wires[0].value == 
                LogicValueBinaryEnum.ZERO.value else LogicValueBinaryEnum.
                ZERO.value}'
378         )
379
380     @classmethod
381     def xor_operation(cls, gate: XorGate, all_fault_dict: dict[Wire, set[
            str]]) -> None:
382         cls.ValidationFaultSimulationDeductiveOperation.xor_operation(
383             gate=gate,
384             all_fault_dict=all_fault_dict
385         )
386
387         all_fault_dict[gate.output_wires[0]] = set()
388
389         for index in range(1, 2**len(gate.input_wires)):
390             binary_index: str = bin(index)[2:]
391
392             if binary_index.count('1') % 2 == 1:
393                 binary_index = binary_index.zfill(len(gate.input_wires))
394                 include_wires: set[Wire] = set()
395                 exclude_wires: set[Wire] = set()
396
397                 for i in range(len(gate.input_wires)):
```

```
398                     if binary_index[i] == LogicValueBinaryEnum.ZERO.value
                            :
399                         exclude_wires = exclude_wires.union(
400                             all_fault_dict[gate.input_wires[i]]
401                         )
402                     else:
403                         if len(include_wires) == 0:
404                             include_wires = include_wires.union(
405                                 all_fault_dict[gate.input_wires[i]]
406                             )
407
408                         else:
409                             include_wires = include_wires.intersection(
410                                 all_fault_dict[gate.input_wires[i]]
411                             )
412
413                 all_fault_dict[gate.output_wires[0]] = all_fault_dict[
                        gate.output_wires[0]].union(
414                     include_wires - exclude_wires
415                 )
416
417         all_fault_dict[gate.output_wires[0]].add(
418             f'{gate.output_wires[0].id}_s-a-{LogicValueBinaryEnum.ONE.
                    value if gate.output_wires[0].value ==
                    LogicValueBinaryEnum.ZERO.value else LogicValueBinaryEnum.
                    ZERO.value}'
419         )
420
421     @classmethod
422     def xnor_operation(cls, gate: XnorGate, all_fault_dict: dict[Wire,
            set[str]]) -> None:
423         cls.ValidationFaultSimulationDeductiveOperation.xnor_operation(
424             gate=gate,
425             all_fault_dict=all_fault_dict
426         )
427
428         all_fault_dict[gate.output_wires[0]] = set()
429
430         for index in range(1, 2**len(gate.input_wires)):
431             binary_index: str = bin(index)[2:]
432
```

```
433             if binary_index.count('1') % 2 == 1:
434                 binary_index = binary_index.zfill(len(gate.input_wires))
435                 include_wires: set[Wire] = set()
436                 exclude_wires: set[Wire] = set()
437 
438                 for i in range(len(gate.input_wires)):
439                     if binary_index[i] == LogicValueBinaryEnum.ZERO.value
                        :
440                         exclude_wires = exclude_wires.union(
441                             all_fault_dict[gate.input_wires[i]]
442                         )
443                     else:
444                         if len(include_wires) == 0:
445                             include_wires = include_wires.union(
446                                 all_fault_dict[gate.input_wires[i]]
447                             )
448 
449                         else:
450                             include_wires = include_wires.intersection(
451                                 all_fault_dict[gate.input_wires[i]]
452                             )
453 
454                 all_fault_dict[gate.output_wires[0]] = all_fault_dict[
                        gate.output_wires[0]].union(
455                     include_wires - exclude_wires
456                 )
457 
458         all_fault_dict[gate.output_wires[0]].add(
459             f'{gate.output_wires[0].id}_s-a-{LogicValueBinaryEnum.ONE.
                    value if gate.output_wires[0].value ==
                    LogicValueBinaryEnum.ZERO.value else LogicValueBinaryEnum.
                    ZERO.value}'
460         )
461 
462     @classmethod
463     def fanout_operation(cls, gate: FanoutGate, all_fault_dict: dict[Wire
            , set[str]]) -> None:
464         cls.ValidationFaultSimulationDeductiveOperation.fanout_operation(
465             gate=gate,
466             all_fault_dict=all_fault_dict
467         )
```

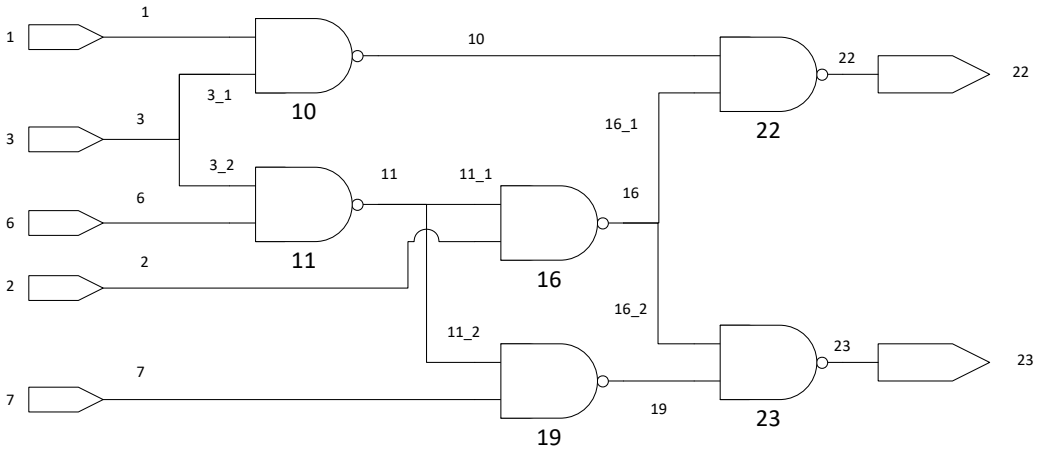
```
468
469             for output_wire in gate.output_wires:
470                 all_fault_dict[output_wire] = set()
471
472                 all_fault_dict[output_wire] = all_fault_dict[output_wire].
                        union(
473                     all_fault_dict[gate.input_wires[0]]
474                 )
475
476                 all_fault_dict[output_wire].add(
477                     f'{output_wire.id}_s-a-{LogicValueBinaryEnum.ONE.value if
                            output_wire.value==LogicValueBinaryEnum.ZERO.value
                            else LogicValueBinaryEnum.ZERO.value}'
478                 )
```

۵ تحلیل مدار c17

در این قسمت قصد داریم به تحلیل مدار c۱۷ بپردازیم. ابتدا مدار را به وسیله دو ورودی که حاوی چهار منطق است، بررسی کرده و شبیه سازی true-value را انجام می دهیم. در ادامه مدار را به وسیله دو ورودی که حاوی دو منطق است، بررسی می کنیم و شبیه سازی اشکال استنتاجی یا Deductive را انجام می دهیم.

۱.۵ شبیه سازی true-value

شبیه سازی true-value به این صورت است که زمانی شبکه مدار های منطقی شکل گرفت، از لایه صفر یعنی پایانه های ورودی، مقادیر را به مدار تزریق کنیم و خروجی هر گیت را محاسبه و بر روی سیم خروجی گیت قرار دهیم و این روند را از تا زمانی به پیانه های خروجی نرسیده ایم، انجام دهیم.



شکل ۱: نمای مدار منطقی مربوط به فایل c17 پس از خواندن فایل به همراه سیم های متصل کننده

۱.۱.۵ تلاش اول

در اولین ورودی که قصد داریم به مدار c17 اعمال کنیم به شکل زیر است:

$$1 : Z, 3 : 0, 6 : 1, 2 : U, 7 : 0$$

که این ورودی پس از اعمال بر روی شبکه ، خروجی های هر سیم و همچنین خروجی های نهایی، به شکل زیر خواهد بود:

```
1  Wire:1, Value:Z
2  Wire:2, Value:U
3  Wire:3, Value:0
4  Wire:6, Value:1
5  Wire:7, Value:0
6  Wire:3_1, Value:0
7  Wire:3_2, Value:0
8  Wire:10, Value:1
9  Wire:11, Value:1
10 Wire:11_1, Value:1
11 Wire:11_2, Value:1
12 Wire:16, Value:U
13 Wire:19, Value:1
14 Wire:16_1, Value:U
15 Wire:16_2, Value:U
16 Wire:22, Value:U
17 Wire:23, Value:U
18 ----------------
19 Output Gate:22, Value:U
20 Output Gate:23, Value:U
```

۲.۱.۵ تلاش دوم

در دومین ورودی که قصد داریم به مدار c17 اعمال کنیم به شکل زیر است:

$$1 : 0, 3 : 1, 6 : Z, 2 : 1, 7 : U$$

که این ورودی پس از اعمال بر روی شبکه ، خروجی های هر سیم و همچنین خروجی های نهایی، به شکل زیر خواهد بود:

```
1  Wire:1, Value:0
2  Wire:2, Value:1
3  Wire:3, Value:1
4  Wire:6, Value:Z
```

```
5  Wire:7, Value:U
6  Wire:3_1, Value:1
7  Wire:3_2, Value:1
8  Wire:10, Value:1
9  Wire:11, Value:Z
10 Wire:11_1, Value:Z
11 Wire:11_2, Value:Z
12 Wire:16, Value:Z
13 Wire:19, Value:U
14 Wire:16_1, Value:Z
15 Wire:16_2, Value:Z
16 Wire:22, Value:Z
17 Wire:23, Value:U
18 ----------------
19 Output Gate:22, Value:Z
20 Output Gate:23, Value:U
```

۲.۵ شبیه سازی اشکال Deductive

الگوریتم و کد الگوریتم شبیه سازی اشکال Deductive در قسمت های قبلی ، توضیح داده شد. در این قسمت قصد داریم که دو ورودی تست ، با منطق دو مقداره به مدار اعمال کنیم و اشکالات هر سیم و همچنین اشکالاتی که در خروجی نمایان می شوند را مشاهده کنیم.

۱.۲.۵ تلاش اول

در اولین بردار تستی که قصد داریم که به مدار c17 اعمال کنیم، به شکل زیر است:

$$1:1,3:0,6:1,2:0,7:0$$

که این بردار تست پس از اعمال بر روی شبکه، اشکالات هر سیم و همچنین اشکالاتی که در خروجی نهایی نمایان می شود، به شکل زیر خواهد بود:

```
1  Wire:1, Value:1, Discovered faults:{'1_s-a-0'}
2  Wire:2, Value:0, Discovered faults:{'2_s-a-1'}
3  Wire:3, Value:0, Discovered faults:{'3_s-a-1'}
4  Wire:6, Value:1, Discovered faults:{'6_s-a-0'}
5  Wire:7, Value:0, Discovered faults:{'7_s-a-1'}
6  Wire:3_1, Value:0, Discovered faults:{'3_s-a-1', '3_1_s-a-1'}
7  Wire:3_2, Value:0, Discovered faults:{'3_s-a-1', '3_2_s-a-1'}
8  Wire:10, Value:1, Discovered faults:{'3_s-a-1', '10_s-a-0', '3_1_s-a-1'}
9  Wire:11, Value:1, Discovered faults:{'3_s-a-1', '11_s-a-0', '3_2_s-a-1'}
10 Wire:11_1, Value:1, Discovered faults:{'3_s-a-1', '11_s-a-0', '11_1_s-a
    -0', '3_2_s-a-1'}
```

```
11 Wire:11_2, Value:1, Discovered faults:{'3_s-a-1', '11_s-a-0', '11_2_s-a
    -0', '3_2_s-a-1'}
12 Wire:16, Value:1, Discovered faults:{'16_s-a-0', '2_s-a-1'}
13 Wire:19, Value:1, Discovered faults:{'19_s-a-0', '7_s-a-1'}
14 Wire:16_1, Value:1, Discovered faults:{'16_s-a-0', '2_s-a-1', '16_1_s-a
    -0'}
15 Wire:16_2, Value:1, Discovered faults:{'16_s-a-0', '2_s-a-1', '16_2_s-a
    -0'}
16 Wire:22, Value:0, Discovered faults:{'10_s-a-0', '16_s-a-0', '2_s-a-1',
    '16_1_s-a-0', '22_s-a-1', '3_1_s-a-1', '3_s-a-1'}
17 Wire:23, Value:0, Discovered faults:{'16_s-a-0', '2_s-a-1', '16_2_s-a-0',
     '19_s-a-0', '23_s-a-1', '7_s-a-1'}
18 ------------------
19 OutputGate: 22, Value:0, Wire:22, Discoverd faults:{'10_s-a-0', '16_s-a
    -0', '2_s-a-1', '16_1_s-a-0', '22_s-a-1', '3_1_s-a-1', '3_s-a-1'}
20 OutputGate: 23, Value:0, Wire:23, Discoverd faults:{'16_s-a-0', '2_s-a
    -1', '16_2_s-a-0', '19_s-a-0', '23_s-a-1', '7_s-a-1'}
```

۲.۲.۵ تلاش دوم

در دومین بردار تستی که قصد داریم که به مدار c17 اعمال کنیم، به شکل زیر است:

$$1:1,3:0,6:0,2:1,7:1$$

```
 1 Wire:1, Value:1, Discovered faults:{'1_s-a-0'}
 2 Wire:2, Value:1, Discovered faults:{'2_s-a-0'}
 3 Wire:3, Value:0, Discovered faults:{'3_s-a-1'}
 4 Wire:6, Value:0, Discovered faults:{'6_s-a-1'}
 5 Wire:7, Value:1, Discovered faults:{'7_s-a-0'}
 6 Wire:3_1, Value:0, Discovered faults:{'3_1_s-a-1', '3_s-a-1'}
 7 Wire:3_2, Value:0, Discovered faults:{'3_2_s-a-1', '3_s-a-1'}
 8 Wire:10, Value:1, Discovered faults:{'3_1_s-a-1', '3_s-a-1', '10_s-a-0'}
 9 Wire:11, Value:1, Discovered faults:{'11_s-a-0'}
10 Wire:11_1, Value:1, Discovered faults:{'11_1_s-a-0', '11_s-a-0'}
11 Wire:11_2, Value:1, Discovered faults:{'11_2_s-a-0', '11_s-a-0'}
12 Wire:16, Value:0, Discovered faults:{'11_1_s-a-0', '16_s-a-1', '2_s-a-0',
     '11_s-a-0'}
13 Wire:19, Value:0, Discovered faults:{'11_2_s-a-0', '19_s-a-1', '11_s-a
    -0', '7_s-a-0'}
14 Wire:16_1, Value:0, Discovered faults:{'2_s-a-0', '11_s-a-0', '11_1_s-a
    -0', '16_s-a-1', '16_1_s-a-1'}
```

```
15 Wire:16_2, Value:0, Discovered faults:{'2_s-a-0', '11_s-a-0', '16_2_s-a
    -1', '11_1_s-a-0', '16_s-a-1'}
16 Wire:22, Value:1, Discovered faults:{'22_s-a-0', '2_s-a-0', '11_1_s-a-0',
     '16_s-a-1', '16_1_s-a-1', '11_s-a-0'}
17 Wire:23, Value:1, Discovered faults:{'23_s-a-0', '11_s-a-0'}
18 ------------------
19 OutputGate: 22, Value:1, Wire:22, Discoverd faults:{'22_s-a-0', '2_s-a
    -0', '11_1_s-a-0', '16_s-a-1', '16_1_s-a-1', '11_s-a-0'}
20 OutputGate: 23, Value:1, Wire:23, Discoverd faults:{'23_s-a-0', '11_s-a
    -0'}
```