Aufgabe 4: Nandu

Team-ID: 00099

Team-Name: Cryptellum

Bearbeiter/-innen dieser Aufgabe:
Selahaddin Inan

10. November 2023

Inhaltsverzeichnis

1 Lösungsidee

Die Aufgabe „Nandu“ (Die Name der Aufgabe kommt wahrscheinlich von den NAND-Gattern) besteht im Wesentlichen daraus, sogenannte Logikgatter zu implementieren. Hierbei sind die weißen Bausteine NAND-Gatter, die blauen OR-Gatter und die roten NOT-Gatter. Nach der Implementierung der einzelnen Gatter mit deren logischen Operationen und nachdem wir die Eingabe gelesen haben, können wir eine Simulation durchführen, um zu sehen, welche Lampen am Ende tatsächlich leuchten.

2 Umsetzung

2.1 Testfälle generieren

Da jede Quelle entweder an (1) oder aus (0) sein kann, müssen wir zuerst einmal (vor der Simulation) die einzelnen Testfälle für jede mögliche Konfiguration von Quellen generieren, hierzu können wir uns die Arbeit sparen, indem wir eine Python-Library namens itertools benutzen. Hierbei verwenden wir die Funktion itertools.product(*iterables, repeat=int) und geben bei den iterables [0, 1] an, da die Quellen nur an oder aus sein können und geben bei Repeat die Anzahl der Quellen an. Somit haben wir eine Liste aus alle n möglichen Testfällen erzeugt.

2.2 Logikgatter Implementation

Nachdem wir alle möglichen Testfälle generiert haben, können wir in eine Funktion die drei Logikgatter (NAND, NOT, OR) implementieren. Dabei gehen wir bei jedem Gatter so vor: 1. Farbe bestimmen (also Gattertyp) 2. Schauen, ob und welche Eingaben (am Baustein) an bzw. aus sind und implementieren dementsprechend für jede Möglichkeit von Eingaben die Ausgaben bei dem jeweiligen Baustein.

2.3 Simulation

Jetzt iterieren wir durch jede Beispieldatei und erzeugen am Anfang eine Wahrheits-Liste, diese Liste ist so lang, wie die Zeile selbst und beinhaltet nur Boolean-Werte für jeden einzelnen Baustein-Teil. Die Liste wird benutzt, um zu speichern, welche Teile von welchen Bausteinen an bzw. aus sind und ist nutzhaft, um am Ende zeigen zu können, an welchen Stellen das „Licht“ leuchtet (ist besonders wichtig, wenn man die Zusatz-Aufgabe bearbeiten möchte). Danach iterieren wir durch jedes einzelnes Element in einer Zeile und gehen dabei so vor: 1. Schauen, ob wir auf einem Baustein gelandet sind, was nicht schon „benutzt“ wurde (also zu einem anderen Baustein gehört) 2. Schauen, wo der andere Teil des Bausteins sich befindet3. Schauen, ob dieser Teil auch schon benutzt worden ist 4. Falls nicht, führen wir unsere Logik-Gatter Funktion durch und setzen dabei in der Wahrheits-Liste bei den beiden Positionen des Bausteins die Werte auf True, also benutzt.

3 Beispiele

nandu1

Q1	Q2	L1	L2
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

nandu2

Q1	Q2	L1	L2
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

nandu3

Q1	Q2	Q3	L1	L2	L3	L4
0	0	0	1	0	0	1
0	0	1	1	0	0	0
0	1	0	1	0	1	1
0	1	1	1	0	1	0
1	0	0	0	1	0	1
1	0	1	0	1	0	0
1	1	0	0	1	1	1
1	1	1	0	1	1	0

nandu4

Q1	Q2	Q3	Q4	L1	L2
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	1
0	1	1	1	1	0

nandu4

Q1	Q2	Q3	Q4	L1	L2
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	1
1	1	1	1	0	0

nandu5

Q1	Q2	Q3	Q4	Q5	Q6	L1	L2	L3	L4	L5
0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	0	1	1	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	0	1	0	1	0	0	1	0	0
0	0	0	1	1	0	0	0	0	1	1
0	0	0	1	1	1	0	0	0	1	1
0	0	1	0	0	0	0	0	0	1	0
0	0	1	0	0	1	0	0	0	1	0
0	0	1	0	1	0	0	0	0	1	1
0	0	1	0	1	1	0	0	0	1	1
0	0	1	1	0	0	0	0	1	0	0
0	0	1	1	0	1	0	0	1	0	0
0	0	1	1	1	0	0	0	0	1	1
0	0	1	1	1	1	0	0	0	1	1
0	1	0	0	0	0	0	0	0	1	0
0	1	0	0	0	1	0	0	0	1	0
0	1	0	0	1	0	0	0	0	1	1
0	1	0	0	1	1	0	0	0	1	1
0	1	0	1	0	0	0	0	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	0	1	1	0	0	0	0	1	1
0	1	0	1	1	1	0	0	0	1	1
0	1	1	0	0	0	0	0	0	1	0
0	1	1	0	0	1	0	0	0	1	0
0	1	1	0	1	0	0	0	0	1	1
0	1	1	0	1	1	0	0	0	1	1
0	1	1	1	0	0	0	0	1	0	0
0	1	1	1	0	1	0	0	1	0	0
0	1	1	1	1	0	0	0	0	1	1
0	1	1	1	1	1	0	0	0	1	1
1	0	0	0	0	0	1	0	0	1	0
1	0	0	0	0	1	1	0	0	1	0
1	0	0	0	1	0	1	0	0	1	1
1	0	0	0	1	1	1	0	0	1	1
1	0	0	1	0	0	1	0	1	0	0
1	0	0	1	0	1	1	0	1	0	0
1	0	0	1	1	0	1	0	0	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	0	0	0	1	0	0	1	0
1	0	1	0	0	1	1	0	0	1	0
1	0	1	0	1	0	1	0	0	1	1

nandu5

Q1	Q2	Q3	Q4	Q5	Q6	L1	L2	L3	L4	L5
1	0	1	0	1	1	1	0	0	1	1
1	0	1	1	0	0	1	0	1	0	0
1	0	1	1	0	1	1	0	1	0	0
1	0	1	1	1	0	1	0	0	1	1
1	0	1	1	1	1	1	0	0	1	1
1	1	0	0	0	0	1	0	0	1	0
1	1	0	0	0	1	1	0	0	1	0
1	1	0	0	1	0	1	0	0	1	1
1	1	0	0	1	1	1	0	0	1	1
1	1	0	1	0	0	1	0	1	0	0
1	1	0	1	0	1	1	0	1	0	0
1	1	0	1	1	0	1	0	0	1	1
1	1	0	1	1	1	1	0	0	1	1
1	1	1	0	0	0	1	0	0	1	0
1	1	1	0	0	1	1	0	0	1	0
1	1	1	0	1	0	1	0	0	1	1
1	1	1	0	1	1	1	0	0	1	1
1	1	1	1	0	0	1	0	1	0	0
1	1	1	1	0	1	1	0	1	0	0
1	1	1	1	1	0	1	0	0	1	1
1	1	1	1	1	1	1	0	0	1	1

Im folgenden Beispiel (nandu6) sehen wir, dass das Licht, was auf nicht auf den richtigen Eingang eines Roten-Gatters trifft, sozusagen "blockiert" wird (sprich, es wird nicht weitergeleitet).

nandu6 (eigenes Beispiel)

Q1	Q2	L1	L2
0	0	0	0
0	1	0	0
1	0	1	1
1	1	1	1

In solchen Faellen wie im folgenden Beispiel (nandu7) nehmen wir an, dass "ungueltige Gatter" einfach das Licht weiterleiten, da Sie eigentlich gar nicht existieren duerfen (sonst muesste man ja die Bausteine auseinander bauen)

nandu7 (eigenes Beispiel)

Q1	Q2	Q3	Q4	Q5	L1	L2	L3	L4	L5
0	0	0	0	0	1	0	0	0	0
0	0	0	0	1	1	0	0	0	1
0	0	0	1	0	1	0	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	0	0	1	0	1	0	0
0	0	1	0	1	1	0	1	0	1
0	0	1	1	0	1	0	1	1	0
0	0	1	1	1	1	0	1	1	1
0	1	0	0	0	1	1	0	0	0
0	1	0	0	1	1	1	0	0	1

nandu7 (eigenes Beispiel)

Q1	Q2	Q3	Q4	Q5	L1	L2	L3	L4	L5
0	1	0	1	0	1	1	0	1	0
0	1	0	1	1	1	1	0	1	1
0	1	1	0	0	1	1	1	0	0
0	1	1	0	1	1	1	1	0	1
0	1	1	1	0	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	0
1	0	0	1	1	0	0	0	1	1
1	0	1	0	0	0	0	1	0	0
1	0	1	0	1	0	0	1	0	1
1	0	1	1	0	0	0	1	1	0
1	0	1	1	1	0	0	1	1	1
1	1	0	0	0	0	1	0	0	0
1	1	0	0	1	0	1	0	0	1
1	1	0	1	0	0	1	0	1	0
1	1	0	1	1	0	1	0	1	1
1	1	1	0	0	0	1	1	0	0
1	1	1	0	1	0	1	1	0	1
1	1	1	1	0	0	1	1	1	0
1	1	1	1	1	0	1	1	1	1

4 Quellcode

```
# Der Verstaendlichkeit halber ist der folgende Code gekuerzt

# Funktion zur Generierung aller moeglichen Testfaelle basierend auf den gegebenen Indizes der Quellen
def test_case_generation(indeces):
    possible_test_cases = list(
        map(list, itertools.product([0, 1], repeat=len(indeces))))
    return possible_test_cases

# Funktion zur Logikgatter-Implementierung, leftRight gibt an, ob das Gatter-Teil, was mit dem aktuellen Gatter verbunden ist, links oder rechts ist. Hier in der Doku ist nur ein Teil angegeben
def gate_logic(k, arr, id, leftRight):
# NOT LOGIC GATE
    if id == "R":
        if leftRight == "r":
            if arr[k] == 0:
                arr[k] = 1
                arr[k+1] = 1
                return arr

            elif arr[k] == 1:
                arr[k] = 0
                arr[k+1] = 0
                return arr

        if leftRight == "l":
            if arr[k] == 0:
                arr[k] = 1
                arr[k-1] = 1
                return arr

            elif arr[k] == 1:
                arr[k] = 0
                arr[k-1] = 0
```

```
33                return arr
```

Testfällen Generation und Logikgatter-Implementation

```
for j in range(2, m+1):
    print(crnt_arr, case)
    # zuruecksetzen der verwendeten Gatter fuer jede Zeile
    usedGates = [False for _ in range(n)]

    # aufteilen der Zeile basierend auf den Zeichen
    zeile = [line for line in lines[j].split("␣") if line != ""]
    # vorherige Zeile fuer die ueberpruefung, ob ein "X" ueber dem Gatter ist
    vorherige_zeile = [_ for _ in lines[j-1].split("␣") if _ != ""]
    # Iteration ueber die Zeile
    for k in range(0, len(zeile)):
        # wenn das Zeichen kein X ist und das Gatter noch nicht verwendet wurde
        if zeile[k] != "X":
            # bedingung um zu schauen, ob ein "X" ueber dem Gatter ist, falls ja,
    vernachluessigen wir diesen Teil des Gatters

            if usedGates[k] == False:
                # NAND LOGIC GATE
                if zeile[k] == "W":
                    # schauen, wo der andere Eingang des Gatters ist
                    if k - 1 >= 0:
                        if zeile[k-1] == "W" and usedGates[k-1] == False:
                            gate_logic(k=k, arr=crnt_arr,
                                       id="W", leftRight="l")

                            usedGates[k] = True
                            usedGates[k-1] = True

                    if k + 1 <= len(zeile):
                        if zeile[k+1] == "W" and usedGates[k+1] == False:
                            gate_logic(k=k, arr=crnt_arr,
                                       id="W", leftRight="r")
                            usedGates[k] = True
                            usedGates[k+1] = True
```

Simulation