فاز اول

فاز اول از دو بخش کلی تشکیل شده است که در بخش اول ابتدا هدف تبدیل فایل با فرمت isc. به فایل با فرمت bench. است و بعد خواند فایل مقادیر ورودی اولیه به ورودی‌های مدار و سپس محاسبه‌ی خروجی‌ها و گره‌های میانی است.

تبدیل فایل isc به فایل bench

ابتدا فایل isc را به صورت خط به خط می‌خوانیم و در یک لیست پایتون (آرایه) ذخیره می‌کنیم به طوری که هر عنصر آن یک خط باشد که شامل یک خط دستوری است و از اضافه کردن خط‌های کامنت به لیست جلوگیری می‌کنیم.

```
# Read the file line by line and ignore the comments
with open(input_file) as file:
    for line in file:
        if not (line.__contains__("*")):
            lines.append(line.strip())
```

حال روتین زیر را برای تمامی خط‌ها اجرا می‌کنیم.

ابتدا چک کنیم که می‌خواهیم روی آن تبدیل انجام دهیم تماما عددی نباشد و یعنی یک دستور باشد و برای این کار یک تابع طراحی می‌کنیم که یک خط را از روی فواصل از هم جدا می‌کند و اگر حتی یکی از آنها در آن حرفی به کار رفته باشید false و در غیر این صورت true باز می‌گرداند و برای تمامی خطوطی که حالت تماما عدد را ندارند باید پارس کردن را انجام دهیم.

تابع نوشته شده برای چک کردن این که در یک خط آیا تمامی کلمات عددی هستند یا خیر به حالت زیر است.

```
# Functions used for MAKING BENCH FILE
# Takes a string which is a line of file and returns TRUE if it's only numbers (only used when giving a list of fan_ins)
def check_all_number(string):
    string = string.split()
    flag = True
    for i in string:
        if not i.isnumeric():
            flag = False
            break
    return flag
```

حال بر روی هر که تماما عدد نیست کلمات که با فاصله از هم جدا شده اند را بدست می‌آوریم. از آنجایی که در خط‌های شاخه‌های fanout برای نشان دادن ساقه از آدرس آن استفاده می‌کنیم و نه شماره سیم، یک دیکشنری مورد نیاز است که بدانیم هر هر آدرس نماد چه سیمی است و هر خط دستوری را که می‌خوانیم باید اول آنرا به دیکشنری آدرس – نام سیم اضافه کنیم.

از آنجایی که خط دستوری است چند حالت مختلف ممکن است پیش آید که با توجه به هرکدام باید کاری کنیم و داده را برای فایل bench آماده کنیم.

اگر قسمت مربوط به تعداد fanin‌ها بیش از صفر باید یعنی دقیقا اولین خط بعد از آن لیستی از ورودی‌ها است و یک گیت دستوری است و به قسمت بدنه مدار یک خط اضافه می‌کنیم که سیم خروجی برابر گیت است و ورودی‌های آنرا از خط بعدی می‌خوانیم و اضافه می‌کنیم و یک عنصر از بدنه مدار بدست آمده.

```
# A line that is all number is just a list of fan_ins to a gate
if not check_all_number(lines[i]):

    # data: [output_wire, address, gate_type, number_of_fan_outs, number_of_fan_ins, ...sa faults...]
    data = lines[i].split()
    name_dict[data[1]] = data[0]
    try:

        # Has fan_ins
        if int(data[4]) > 0:
            i += 1
            fan_ins = lines[i].split()
            tmp = data[0] + " = " + data[2].upper() + "("
            gate_counter[data[2].upper()] = gate_counter.get(data[2].upper()) + 1
            for j in range(0, len(fan_ins)):
                tmp = tmp + fanout_dict.get(fan_ins[j], fan_ins[j]) + ", "

            # At the end tmp has an extra ", " and these two chars aren't needed just needs ")" at the end
            final[2].append(tmp[:-2] + ")")
```

در صورتی که قسمت مربوط به fanin‌ها برابر صفر باشد یعنی خط در حال تعریف یک ورودی اولیه است و آنرا قسمت مربوط به ورودی‌های فایل خروجی اضافه می‌کنیم.

```
# No fan_ins (PI wire)
else:
    if data[2] == 'inpt':
        final[0].append("INPUT(" + data[0] + ")")
```

حال آن خط را با توجه به تعداد fanout‌ها مورد بررسی قرار می‌دهیم.

اگر تعداد fanout‌ها برابر صفر باشد یعنی آن خط مربوط به خروجی اصلی مدار است و آنرا به قسمت خروجی‌ها اضافه می‌کنیم. با توجه به تعریف موجود در [۱] خروجی‌ها باید تعداد fanout برابر ۰ داشته باشند و هر تعداد آنها برابر صفر مربوط به خروجی است.

```
# No fan_outs (PO wire)
if int(data[3]) == 0:
    final[1].append("OUTPUT(" + data[0] + ")")
```

در صورتی که تعداد fanout بیش از ۱ باشد به آن تعداد خط خروجی‌های fanout است که با توجه به آن تعداد خط بعدی بدست می‌آیند. وجود fanout موردی به فایل bench اضافه نمی‌کند و فقط از آنجایی که در فایل bench شماره شاخه‌های fanout وجود ندارد و آنها با شماره ساقه عوض می‌شوند به یک دیکشنری نیاز داریم که بدانیم هر شماره شاخه از چه شماره ساقه‌ای مقدار می‌گیرد و از آنجایی که شماره شاخه مقداری برابر آدرس ساقه دارد آدرس ساقه را با دیکشنری آدرس – نام سیم مطابقت می‌دهیم.

```
# Multiple fan_outs
elif int(data[3]) > 1:
    for j in range(0, int(data[3])):
        i += 1

        # fan_out_Line_data = [output_wire, address, from, fan_in_address, ...sa faults...] and only
        # adds to the fan_out dict cause the steam and the branch have same values and are names by
        # same wire in bench
        fan_out_Line_data = lines[i].split()
        fanout_dict[fan_out_Line_data[0]] = name_dict.get(fan_out_Line_data[3])
```

در صورتی که تعداد fanoutها برابر ۱ باشد خطی دیگر در فایل isc نیست و آن fanout به صورت ورودی در خط دیگری قابل مشاهده است و آنجا بدست می‌آید.

در پایان اجرای این قسمت برای c17 دو دیکشنری به حالت زیر هستند:

```
branch: stem {'8': '3', '9': '3', '14': '11', '15': '11', '20': '16', '21': '16'}
address: name {'1gat': '1', '2gat': '2', '3gat': '3', '8fan': '8', '9fan': '9', '6gat':
```

دیکشنری دوم ادامه دارد و برای تمام خط‌ها که فقط عدد نیستند، یک ورودی به آن اضافه می‌کند و در ساخت دیکشنری بالا استفاده می‌شود.

برای خروجی نوشتن یک لیست داریم که از سه زیر لیست ساخته شده است، قسمت اول مربوط به ورودی‌ها، قسمت دوم مربوط به خط‌های خروجی و قسمت سوم مربوط به بدنه مدار است و با توجه به این ترتیب در بالا به زیر لیست‌های مختلف که final[0] و final[1] و final[2] هستند رشته مورد نیاز خروجی را اضافه می‌کنیم.

برای قسمت کامنت‌های بالایی برای تعداد output و input از تعداد عناصر زیر لیست مربوط به آنها استفاده می‌کنیم و برای شمردن تعداد گیت‌ها از یک دیکشنری استفاده می‌کنیم که موارد آن نام گیت‌ها هستند و هر لحظه خطی خواستیم به زیر لیست مربوط به بدنه اضافه کنیم، شمارنده مربوط به آن عنصر را در دیکشنری زیاد می‌کنیم. بعد از نوشتن کامنت‌ها در فایل خروجی، سه قسمت final را در فایل خروجی می‌نویسیم که هر عنصر در یک خط و بین هر ئو زیر لیست نیز یک خط خالی وجود دارید که به حالت زیر پیاده شده است.

```
# Writing data to bench file
file_out = open(output_file, "w")
file_out.write("#" + output_file.split(".")[0] + "\n")
file_out.write("#" + str(len(final[0])) + " inputs\n")
file_out.write("#" + str(len(final[1])) + " outputs\n")
counter = 0
for gate in gate_counter.keys():
    counter += gate_counter[gate]
file_out.write("#" + str(gate_counter['NOT']) + " inverters\n")
file_out.write("#" + str(counter) + " gates (")
for gate in gate_counter.keys():
    if gate_counter[gate] > 0:
        file_out.write(" " + str(gate_counter[gate]) + " " + gate + "s")
file_out.write(" )\n\n")
for i in range(0, 3):
    for j in final[i]:
        file_out.write(j + "\n")
    file_out.write("\n")
file_out.close()
```

در نهایت کدهای مربوط قسمت تبدیل isc به bench به طور کامل و با تابع مورد نیاز آن در زیر آورده شده است:

```
# Functions used for MAKING BENCH FILE
# Takes a string which is a line of file and returns TRUE if it's only
numbers (only used when giving a list of fan_ins)
def check_all_number(string):
    string = string.split()
    flag = True
    for i in string:
        if not i.isnumeric():
            flag = False
            break
    return flag
# Takes an isc format file and outputs the bench file only needs the path of
isc file and path for output
def isc_to_bench(input_file, output_file):
    # Dict of fanout_branch :  fanout_steam
    fanout_dict = {}

    # Dict of address : name
    name_dict = {}

    # First one is inputs, second one is output, last one is body
    final = [[], [], []]

    # Lines in the input file
    lines = []
    gate_counter = {'AND': 0, 'NAND': 0, 'OR': 0, 'NOR': 0, 'XOR': 0, 'XNOR':
```

```
0, 'NOT': 0, 'BUFF': 0}

    # Read the file line by line and ignore the comments
    with open(input_file) as file:
        for line in file:
            if not (line.__contains__("*")):
                lines.append(line.strip())
    for i in range(0, len(lines)):

        # A line that is all number is just a list of fan_ins to a gate
        if not check_all_number(lines[i]):

            # data: [output_wire, address, gate_type, number_of_fan_outs,
number_of_fan_ins, ...sa faults...]
            data = lines[i].split()
            name_dict[data[1]] = data[0]
            try:

                # Has fan_ins
                if int(data[4]) > 0:
                    i += 1
                    fan_ins = lines[i].split()
                    tmp = data[0] + " = " + data[2].upper() + "("
                    gate_counter[data[2].upper()] =
gate_counter.get(data[2].upper()) + 1
                    for j in range(0, len(fan_ins)):
                        tmp = tmp + fanout_dict.get(fan_ins[j], fan_ins[j]) +
", "

                    # At the end tmp has an extra ", " and these two chars
aren't needed just needs ")" at the end
                    final[2].append(tmp[:-2] + ")")

                # No fan_ins (PI wire)
                else:
                    if data[2] == 'inpt':
                        final[0].append("INPUT(" + data[0] + ")")

                # No fan_outs (PO wire)
                if int(data[3]) == 0:
                    final[1].append("OUTPUT(" + data[0] + ")")

                # Multiple fan_outs
                elif int(data[3]) > 1:
                    for j in range(0, int(data[3])):
                        i += 1

                        # fan_out_Line_data = [output_wire, address, from,
fan_in_address, ...sa faults...] and only
                        # adds to the fan_out dict cause the steam and the
branch have same values and are names by
                        # same wire in bench
                        fan_out_Line_data = lines[i].split()
                        fanout_dict[fan_out_Line_data[0]] =
name_dict.get(fan_out_Line_data[3])
            except:
                pass
```

```
    # Writing data to bench file
    file_out = open(output_file, "w")
    file_out.write("#" + output_file.split(".")[0] + "\n")
    file_out.write("#" + str(len(final[0])) + " inputs\n")
    file_out.write("#" + str(len(final[1])) + " outputs\n")
    counter = 0
    for gate in gate_counter.keys():
        counter += gate_counter[gate]
    file_out.write("#" + str(gate_counter['NOT']) + " inverters\n")
    file_out.write("#" + str(counter) + " gates (")
    for gate in gate_counter.keys():
        if gate_counter[gate] > 0:
            file_out.write(" " + str(gate_counter[gate]) + " " + gate + "s")
    file_out.write(" )\n\n")
    for i in range(0, 3):
        for j in final[i]:
            file_out.write(j + "\n")
        file_out.write("\n")
    file_out.close()
```

ورودی دادن به مدار و محاسبه مقادیر روی سیم‌ها

در این قسمت مورد اصلی یک دیکشنری است که ترتیب دارد ورودی‌ها را حفظ می‌کند و اطلاعات داخل آنها سیم‌ها و مقادیر روی آنها است.

در این قسمت ابتدا فایل مربوط به ورودی‌ها را می‌خوانیم و خط اول که اسم سیم‌ها است و خط دوم مقادیر مربوط به آنها است و آنها را به دیکشنری اضافه می‌کنیم.

```
with open(input_data) as file:

    # input_wires: PI1 PI2 PI3 PI4 ...
    input_wires = file.readline().strip().split()

    # input_wire_values: 0/1/U 0/1/U 0/1/U 0/1/U ...
    input_wire_values = file.readline().strip().split()
for i in range(len(input_wires)):
    wire_value[input_wires[i]] = input_wire_values[i]
```

حال فایل مربوط به bench را خط به خط می‌خوانیم و خط‌هایی که کامنت نیستند و خالی نیستند را مورد بررسی قرار می‌دهیم.

از آنجایی که مقادیر مربوط به ورودی‌ها در قطعه کد بالا مورد بررسی قرار داده‌ایم دیگر خط‌های مربوط به ورودی را کاری نداریم و از خط‌های مربوط به تعریف خروجی شروع می‌کنیم و بار اول که خروجی را به دیکشنری اضافه می‌کنیم به آنها مقدار اولیه U را می‌دهیم.

```
for line in lines:
    if line.__contains__("OUTPUT"):
        tmp = get_wires(line, "OUTPUT")
        for PO in tmp:
            wire_value[PO] = "U"
```

حال تمامی خط‌هایی که شامل = هستند یعنی بدنه مدار هستند را مورد بررسی قرار می‌دهیم و محاسبات را انجام می‌دهیم.

از آنجایی که با رشته‌ها در حال کار هستیم و در پایتون وقتی بخواهیم زیررشته را چک کنیم باید به وجود رشته OR در NOR و NOR در XNOR بر می‌خوریم، از else ifها استفاده می‌کنیم و باید این ترتیب اول زیر رشته جامع‌تر و بعد جزئی تر را رعایت کنیم.

ابتدا NAND را مورد برریس قرار می‌دهیم.

از آنجایی که ممکن است تعداد ورودی‌های هر گیت بیش از ۲ باشد و یک لیست از سیم‌ها موجود باشد و برای هر خط باید این لیست بدست آید، تابع زیر را پیاده سازی می‌کنیم که ورودی آن سمت راست = است و به ما لیستی از سیم‌های ورودی را می‌دهد.

```
# Functions used for SIMULATION
# Takes right part of "=" and gives back the list wires of fan_ins used to feed that gate
def get_wires(string, gate):
    string = string.replace(gate, "")
    string = string.replace("(", "")
    string = string.replace(")", "")
    wires = string.split(",")
    return wires
```

حال با توجه به این که در bench ما شاخه‌های fanout را از ساقه جدا نمی‌کنیم باید روشی برای تشخیص وجود fanout بدست آوریم و بدانیم چند شاخه است. برای این موضوع اگر یک سیم در سمت راست تساوی بیش از یک بار باشد به آن تعداد fanout از آن ساقه وجود دارد.

برای این موضوع یک دیکشنری می‌سازیم که که کلید آن شماره سیم و مقدار آن تعداد استفاده از آن در سمت راست‌ها است و به صورت یک شمارنده است که در هر خط افزایش می‌یابد. پیاده سازی این تابع به حالت زیر است.

```
# Takes the dictionary of number of each wire used as input for each gate which is later used to know which wires
# represent fan_out stems so the output would list the branches of fan_outs as well
def wire_counter_right(wire_list, fanout_dict):
    for item in wire_list:
        fanout_dict[item] = fanout_dict.get(item, 0) + 1
```

حال با توجه به تعریف NAND که تعدادی ورودی است که آنها را با هم AND می‌کنیم و در نهایت NOT می‌کنیم و با توجه به ۳ منطق ۳ مقدار دو تابع AND و NOT را پیاده می‌کنیم.

در تابع AND اگر یکی از دو مقدار ۰ باشد، خروجی ۱ و در غیر این صورت اگر هر دو ۱ باشند خروجی ۱ می‌شود و در غیر این حالات خروجی U است. خط اول ۵ حالت 00، 01، 0U، 10، U0 را شامل می‌شود، خط بعد ۱ حالت و در نهایت ۳ حالت 1U، U1، UU در خط آخر بدست می‌آیند.

```
def my_and(a, b):
    if a == '0' or b == '0':
        return '0'
    elif a == '1' and b == '1':
        return '1'
    else:
        return 'U'
```

برای پیاده سازی NOT هم ۱ به ۰ و برعکس آن وجود دارد ولی نقیض U همان U است.

```
def my_not(a):
    if a == '1':
        return '0'
    elif a == '0':
        return '1'
    else:
        return 'U'
```

حال با کمک این توابع مقدار حاصل را حساب می‌کنیم و اگر بیش از ۲ ورودی باشد ابتدا دوتای اول را AND می‌کنیم و بعد دانه دانه با بعدی‌ها AND می‌کنیم و در نهایت NOT می‌کنیم.

و در نهایت سیم سمت چپ با مقدار بدست آمده را به دیکشنری اضافه می‌کنیم. پیاده سازی گیت AND نیز به همین ترتیب است و فقط قسمت NOT آخر را ندارد.

```
    if tmp[1].__contains__("NAND"):
        wires = get_wires(tmp[1], "NAND")
        wire_counter_right(wires, fanout_check)
        tmp_val = my_and(wire_value[wires[0]], wire_value[wires[1]])
        for i in range(2, len(wires)):
            tmp_val = my_and(tmp_val, wire_value[wires[i]])
        tmp_val = my_not(tmp_val)
        wire_value[tmp[0]] = tmp_val
    elif tmp[1].__contains__("AND"):
        wires = get_wires(tmp[1], "AND")
        wire_counter_right(wires, fanout_check)
        tmp_val = my_and(wire_value[wires[0]], wire_value[wires[1]])
        for i in range(2, len(wires)):
            tmp_val = my_and(tmp_val, wire_value[wires[i]])
        wire_value[tmp[0]] = tmp_val
```

حال برای پیاده سازی XOR و XNOR با توجه به تعریف به تابع پایه‌ای XOR نیاز است که اگر حتی یکی از ورودی‌ها U باشد، خروجی هم U است زیرا به میزان آن ربط دارد. در حالتی که هیچکدام U نباشد، اگر ورودی اول ۱ باشد، خروجی NOT ورودی دیگر و اگر ۰ باشد خروجی همان ورودی دیگر است.

```
def my_xor(a, b):
    if a == 'U' or b == 'U':
        return 'U'
    elif a == '1':
        return my_not(b)
    else:
        return b
```

با همان روش بالا هم XNOR و XOR را هم پیاده می‌کنیم.

```
    elif tmp[1].__contains__("XNOR"):
        wires = get_wires(tmp[1], "XNOR")
        wire_counter_right(wires, fanout_check)
        tmp_val = my_xor(wire_value[wires[0]], wire_value[wires[1]])
        for i in range(2, len(wires)):
            tmp_val = my_xor(tmp_val, wire_value[wires[i]])
        tmp_val = my_not(tmp_val)
        wire_value[tmp[0]] = tmp_val
    elif tmp[1].__contains__("XOR"):
        wires = get_wires(tmp[1], "XOR")
        wire_counter_right(wires, fanout_check)
        tmp_val = my_xor(wire_value[wires[0]], wire_value[wires[1]])
        for i in range(2, len(wires)):
            tmp_val = my_xor(tmp_val, wire_value[wires[i]])
        wire_value[tmp[0]] = tmp_val
```

حال سر پیاده سازی OR و NOR می‌رویم و تابع پایه‌ای OR را ابتدا به حالت زیر پیاده می‌کنیم که اگر یکی از ورودی‌های آن ۱ باشد خروجی ۱ و فقط در صورتی ۰ است که هر دو ورودی ۰ باشند و در غیر این دو U است.

```
def my_or(a, b):
    if a == '1' or b == '1':
        return '1'
    elif a == '0' and b == '0':
        return '0'
    else:
        return 'U'
```

با همان منطق قبلی NOR و OR را هم پیاده می‌کنیم.

```
elif tmp[1].__contains__("NOR"):
    wires = get_wires(tmp[1], "NOR")
    wire_counter_right(wires, fanout_check)
    tmp_val = my_or(wire_value[wires[0]], wire_value[wires[1]])
    for i in range(2, len(wires)):
        tmp_val = my_or(tmp, wire_value[wires[i]])
    tmp_val = my_not(tmp_val)
    wire_value[tmp[0]] = tmp_val
elif tmp[1].__contains__("OR"):
    wires = get_wires(tmp[1], "OR")
    wire_counter_right(wires, fanout_check)
    tmp_val = my_or(wire_value[wires[0]], wire_value[wires[1]])
    for i in range(2, len(wires)):
        tmp_val = my_or(tmp_val, wire_value[wires[i]])
    wire_value[tmp[0]] = tmp_val
```

از آنجایی که منطق NOT را از قبل اضافه کرده‌ایم پیاده سازی آن با استفاده از همان تابع به حالت زیر انجام می‌دهیم.

```
elif tmp[1].__contains__("NOT"):
    wires = get_wires(tmp[1], "NOT")
    wire_counter_right(wires, fanout_check)
    wire_value[tmp[0]] = my_not(wire_value[wires[0]])
```

برای این که روند کد به همان شکل باشد، یک تابع برای بافر هم پیاده می‌کنیم که همان ورودی را به خروجی می‌دهد با این حال که می‌توان این کار انجام نداد و مستقیم همان جا خروجی را محاسبه کرد.

```
def my_buff(a):
    return a
```

```
elif tmp[1].__contains__("BUFF"):
    wires = get_wires(tmp[1], "BUFF")
    wire_counter_right(wires, fanout_check)
    wire_value[tmp[0]] = my_buff(wire_value[wires[0]])
```

در پیاده سازی NOT و BUFF از آنجایی که فقط یک ورودی وجود دارد دیگر حلقه روی لیست ورودی‌ها وجود ندارد ولی از آنجایی که آنها ورودی آنها می‌تواند از یک ساقه fanout آمده باشد و در خروجی روی خطوط می‌خواهیم آنرا نیز جدا مشاهده کنیم، باید آپدیت مقادیر داخل دیکشنری تعداد سیم‌های سمت راست تساوی انجام شود.

حال در پایان حلقه تمامی خطوط مدار بدست آمده است و می‌دانیم کدام خطوط fanout هستند و مقادر را آماده داریم و باید فقط فرمت خروجی را آماده کنیم.

برای این کار روی دیکشنری سیم – مقدار پیمایش می‌کنیم و اسم سیم با مقدار آنرا می‌نویسم و بعد چک می‌کنیم اگر آن مقدار یک ساقه fanout هست یا نه و اگر باشد، به همان تعداد موجود در دیکشنری تعداد سیم‌های سمت راست تساوی با اندیس‌های ۰ تا یکی کمتر از تعداد آن، در داخل خروجی می‌نویسیم و بعد برای نوشتن سیم بعدی اقدام می‌کنیم.

پیاده سازی موارد بالا به حالت زیر انجام شده است.

```
for wire in wire_value.keys():
    final.append(wire + ": " + wire_value[wire])
    if wire in fanout_check.keys() and fanout_check[wire] > 1:
        for i in range(0, fanout_check.get(wire)):
            final.append(wire + "_" + str(i) + ": " + wire_value[wire])
file_out = open(output_file, "w")
for i in final:
    file_out.write(i + "\n")
file_out.close()
```

برای c17 و با تمامی ورودی‌ها برابر ۱، دیکشنری‌های سیم – مقدار و تعداد سیم در سمت راست به حالت زیر است.

```
wire value: OrderedDict([('1', '1'), ('2', '1'), ('3', '1'), ('6', '1'), ('7', '1'), ('22', '1'), ('23', '0'), ('10', '0'), ('11', '0'), ('16', '1'), ('19', '1')])
fan out check: {'1': 1, '3': 2, '6': 1, '2': 1, '11': 2, '7': 1, '10': 1, '16': 2, '19': 1}
```

و فایل خروجی بدست آمده به حالت زیر است.

```
1: 1
2: 1
3: 1
3_0: 1
3_1: 1
6: 1
7: 1
22: 1
23: 0
10: 0
11: 0
11_0: 0
11_1: 0
16: 1
16_0: 1
16_1: 1
19: 1
```

همان طور که مشاهده می‌شود مقادیر شاخه‌های fanout را هم جز ساقه به حالت جدا در فایل برای ما ذخیره کرده است.

در نهایت کدهای مربوط به خروجی گرفتن از مدار در زیر آمده است

```
# Functions used for SIMULATION
# Takes right part of "=" and gives back the list wires of fan_ins used to
feed that gate
def get_wires(string, gate):
    string = string.replace(gate, "")
    string = string.replace("(", "")
    string = string.replace(")", "")
    wires = string.split(",")
    return wires


# Takes the dictionary of number of each wire used as input for each gate
which is later used to know which wires
# represent fan_out stems so the output would list the branches of fan_outs
as well
def wire_counter_right(wire_list, fanout_dict):
    for item in wire_list:
        fanout_dict[item] = fanout_dict.get(item, 0) + 1


def my_and(a, b):
    if a == '0' or b == '0':
        return '0'
    elif a == '1' and b == '1':
        return '1'
    else:
```

```
        return 'U'


def my_or(a, b):
    if a == '1' or b == '1':
        return '1'
    elif a == '0' and b == '0':
        return '0'
    else:
        return 'U'


def my_xor(a, b):
    if a == 'U' or b == 'U':
        return 'U'
    elif a == '1':
        return my_not(b)
    else:
        return b


def my_not(a):
    if a == '1':
        return '0'
    elif a == '0':
        return '1'
    else:
        return 'U'


def my_buff(a):
    return a
# Takes the path of a bench file description and the path of a PI file and
the path to produce an output file containing
# all the wires and their values
def run_bench(input_bench, input_data, output_file):
    # Dict of wire : value which will have all the wires in the circuit and
their values and gets written to output file
    wire_value = collections.OrderedDict()

    # Dict of wire : number_of_times_in_input_of_gates so the output could
know which stems have how many branches and
    # write the value of branch to output file as well
    fanout_check = {}

    # Lines of bench file
    lines = []

    final = list()
    with open(input_data) as file:

        # input_wires: PI1 PI2 PI3 PI4 ...
        input_wires = file.readline().strip().split()

        # input_wire_values: 0/1/U 0/1/U 0/1/U 0/1/U ...
        input_wire_values = file.readline().strip().split()
    for i in range(len(input_wires)):
```

```
        wire_value[input_wires[i]] = input_wire_values[i]
    with open(input_bench) as file:
        for line in file:
            if not line.__contains__("#") and line != "\n":
                lines.append(line.strip())
    for line in lines:
        if line.__contains__("OUTPUT"):
            tmp = get_wires(line, "OUTPUT")
            for PO in tmp:
                wire_value[PO] = "U"
        if line.__contains__("="):

            # tmp: output_wire=GATE(Wire1, Wire2, ...)
            tmp = line.replace(" ", "").split("=")
            if tmp[1].__contains__("NAND"):
                wires = get_wires(tmp[1], "NAND")
                wire_counter_right(wires, fanout_check)
                tmp_val = my_and(wire_value[wires[0]], wire_value[wires[1]])
                for i in range(2, len(wires)):
                    tmp_val = my_and(tmp_val, wire_value[wires[i]])
                tmp_val = my_not(tmp_val)
                wire_value[tmp[0]] = tmp_val
            elif tmp[1].__contains__("AND"):
                wires = get_wires(tmp[1], "AND")
                wire_counter_right(wires, fanout_check)
                tmp_val = my_and(wire_value[wires[0]], wire_value[wires[1]])
                for i in range(2, len(wires)):
                    tmp_val = my_and(tmp_val, wire_value[wires[i]])
                wire_value[tmp[0]] = tmp_val
            elif tmp[1].__contains__("XNOR"):
                wires = get_wires(tmp[1], "XNOR")
                wire_counter_right(wires, fanout_check)
                tmp_val = my_xor(wire_value[wires[0]], wire_value[wires[1]])
                for i in range(2, len(wires)):
                    tmp_val = my_xor(tmp_val, wire_value[wires[i]])
                tmp_val = my_not(tmp_val)
                wire_value[tmp[0]] = tmp_val
            elif tmp[1].__contains__("XOR"):
                wires = get_wires(tmp[1], "XOR")
                wire_counter_right(wires, fanout_check)
                tmp_val = my_xor(wire_value[wires[0]], wire_value[wires[1]])
                for i in range(2, len(wires)):
                    tmp_val = my_xor(tmp_val, wire_value[wires[i]])
                wire_value[tmp[0]] = tmp_val
            elif tmp[1].__contains__("NOR"):
                wires = get_wires(tmp[1], "NOR")
                wire_counter_right(wires, fanout_check)
                tmp_val = my_or(wire_value[wires[0]], wire_value[wires[1]])
                for i in range(2, len(wires)):
                    tmp_val = my_or(tmp, wire_value[wires[i]])
                tmp_val = my_not(tmp_val)
                wire_value[tmp[0]] = tmp_val
            elif tmp[1].__contains__("OR"):
                wires = get_wires(tmp[1], "OR")
                wire_counter_right(wires, fanout_check)
                tmp_val = my_or(wire_value[wires[0]], wire_value[wires[1]])
                for i in range(2, len(wires)):
```

```
                tmp_val = my_or(tmp_val, wire_value[wires[i]])
            wire_value[tmp[0]] = tmp_val
        elif tmp[1].__contains__("NOT"):
            wires = get_wires(tmp[1], "NOT")
            wire_counter_right(wires, fanout_check)
            wire_value[tmp[0]] = my_not(wire_value[wires[0]])
        elif tmp[1].__contains__("BUFF"):
            wires = get_wires(tmp[1], "BUFF")
            wire_counter_right(wires, fanout_check)
            wire_value[tmp[0]] = my_buff(wire_value[wires[0]])
for wire in wire_value.keys():
    final.append(wire + ": " + wire_value[wire])
    if wire in fanout_check.keys() and fanout_check[wire] > 1:
        for i in range(0, fanout_check.get(wire)):
            final.append(wire + "_" + str(i) + ": " + wire_value[wire])
file_out = open(output_file, "w")
for i in final:
    file_out.write(i + "\n")
file_out.close()
```

در نهایت در برای اجرای برنامه این دو تابع را صدا می‌کنیم و برای تبدیل isc به bench فقط دو ورودی آدرس فایل‌ها مورد نیاز است و برای ورودی مشخص به مدار دادن و گرفتن مقدار تمامی خطوط سه ورودی آدرس فایل bench، آدرس فایل ورودی‌ها و آدرس خروجی مورد نیاز است.

```
import phaseOne

if __name__ == '__main__':
    isc_file_in = "c17.isc"
    bench_file_out = "c17.bench"
    input_wire_in = "c17.pi"
    output_wire_out = "c17.log"

    phaseOne.isc_to_bench(isc_file_in, bench_file_out)
    phaseOne.run_bench(bench_file_out, input_wire_in, output_wire_out)
```

منابع

[۱] Bryan, D., 1985. The ISCAS'85 benchmark circuits and netlist format. North Carolina State University, 25, p.39.