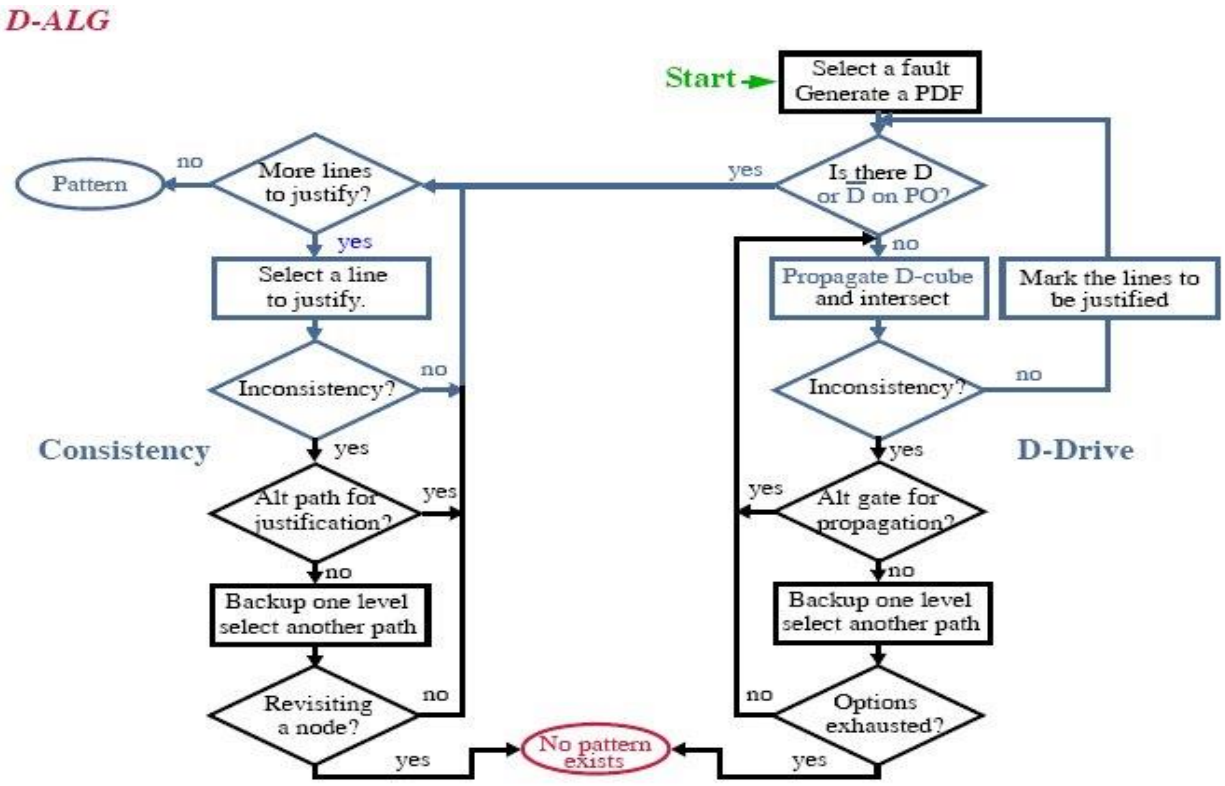
قسمت اول) تعریف الگوریتم D-algorithm

اولین الگوریتم کامل ATPG، دی-الگوریتم است. این الگوریتم، به عنوان ورودی، یک نقطه از مدار را می گیرد، که مشخص شده که stuck at چه فالتی است.

در مرحله ی اول، این فالت در فرایند propagate به خروجی منتقل می شود. سپس در فرایند justify، مقادیر داده شده به سیگنال ها در مدار منتشر می‌شود. اگر در این فرایند به inconsistency یا تناقض در مقدار یک سیگنال برخوردیم، فرایند backtrack انجام می شود.

در فرایند backtrack، به یک پشته که در طول فرایند justify ایجاد شده مراجعه می شود، آخرین عضو این پشته که مقدار آن قابل تغییر کردن است، انتخاب می‌شود و مقدار آن عوض می‌شود. سپس با این مقدار تغییر یافته، فرایند justify دوباره طی می شود. این مراحل تا جایی ادامه پیدا می‌کنند که مقادیر تمامی PI ها مشخص شود.

فلوچارت این الگوریتم در ادامه آمده است:



قسمت دوم) نحوه تعریف توابع و شرح کد پروژه

این پروژه به زبان C++ و در پلتفرم QT نوشته شده است. به این صورت که فایل ورودی با فرمت وریلاگ انتخاب شده، و خروجی برنامه در پنجره قابل مشاهده است.

تابع propagate به این صورت تعریف شده است:

۱- ورودی: یک مقدار سیگنال فالتی، که قرار است به PO منتشر شود
۲- در صورتی که سیگنال داده شده خودش PO است، return کن.
۳- اگر این سیگنال PO نیست، پس یا Fan out یک PO است و یا ورودی یک گیت دیگر.
۴- در صورت وجود Fan out برای سیگنال، آن را در نظر بگیر.
۵- برای ورودی های دیگر گیت مقابل سیگنال، مقدار non-controlling را در نظر بگیر.
۶- تابع propagate را برای حالت های بالا صدا کن.

تابع justify هم به این صورت تعریف شده است:

۱- ورودی: یک سیگنال و مقدار آن، که قرار است justify بر اساس آن انجام شود.
۲- ابتدا این مقدار را به این سیگنال اختصاص بده.
۳- آیا این سیگنال یک PI است؟ اگر بله return کن. اگر نه:
۴- اگر سیگنال PI نیست، پس این سیگنال خروجی یک گیت دیگر است.
۵- اگر مقدار سیگنال، با مقدار non-controlling گیت برابر بود، همه ورودی های گیت پست سر سیگنال نیز باید این مقدار را بگیرند.
۶- اگر مقدار سیگنال برابر با مقدار controlling گیت بود، یکی از ورودی های گیت پشت سر آن را انتخاب کن و آن را برابر با مقدار controlling قرار بده.
۷- تابع justify را برای دو حالت بالا صدا کن.

در تابع تولید تست، ابتدا همه مقادیر x قرار داده می شوند، سپس تک تک برای همه سیگنال ها فالت های SA1 و SA0 در نظر گرفته می شوند، و برای آن ها الگوی تست تولید می شود.

در ابتدا در یک فایل همه تایپ های مورد نیاز روند برنامه مثل مقادیری که یک سیگنال می تواند بگیرد، انواع گیت ها و ... تعریف شده است. مثلاً یک سیگنال می‌تواند ورودی، خروجی و یا هردو باشد. یا می‌تواند ویژگی‌هایی مثل PO بودن، PI بودن یا سیگنال میانی بودن را داشته باشد. می‌تواند یک اسم و یک آی دی داشته باشد.

می‌توان برای سیگنال ها کلاس تعریف نمود تا با صدا زدن تابع constructor مربوط به آن، بتوان به این ویژگی‌ها مقدار اولیه داد.

سپس یک پارسر اجرا می شود، تا برنامه بتواند درکی از زبان وریلاگ و دستورات تعریف شده در آن داشته باشد. سپس به کمک این درک از زبان وریلاگ، مدار به صورت یک گراف با گره های مختلف برای برنامه تعریف می شود.

برای هریک از سیگنال های برنامه، فالت های SA1 و SA0 به صورت جداگانه به تابع تولید تست داده می شوند، تابع تولید تست،به ترتیب توابع propagate و justify را اجرا می‌کند و جلو می برد. و سپس تست وکتور ورودی مربوطه ایجاد می شود.

جهت تشکیل گراف مدار و همچنین کامپایل کردن زبان وریلاگ، از کتابخانه ی boost در زبان C++ استفاده شده است. جهت اجرای برنامه، باید کدهای آن و فایل اصلی پروژه در نرم افزار QT باز شده و دکمه ی اجرا زده شود. در پنجره ی باز شده، دکمه ی ورودی را زده و فایل وریلاگ ورودی را انتخاب می کنیم. برنامه شروع به اجرا شدن کرده و پس از اتمام اجرا، می توان با فشردن دکمه ی خروجی، خروجی برنامه را مشاهده نمود.

برنامه دارای پنج قسمت است که قسمت اصلی main window است. فایل lineparser مسیول کامپایل کردن کد Verilog است.

قسمت سوم) مدار C17 و خروجی پروژه برای این مدار:

Input:

```
module c17 (N1,N2,N3,N6,N7,N22,N23);

input N1,N2,N3,N6,N7;

output N22,N23;

wire N10,N11,N16,N19;

nand NAND2_1 (N10, N1, N3);
nand NAND2_2 (N11, N3, N6);
nand NAND2_3 (N16, N2, N11);
nand NAND2_4 (N19, N11, N7);
nand NAND2_5 (N22, N10, N16);
nand NAND2_6 (N23, N16, N19);
```

endmodule

Output:

#	FaultLine	Type	input_N7	input_N6	input_N3	input_N2	input_N1
1	NAND2_6 _N19	SA-1	1	X	0	0	X
2	NAND2_6 _N19	SA-0	X	1	1	0	X
3	NAND2_6 _N16	SA-1	0	X	0	1	X
4	NAND2_6 _N16	SA-0	X	1	1	0	X
5	NAND2_6 _N23	SA-1	0	1	1	X	X
6	NAND2_6 _N23	SA-0	1	X	0	X	X
7	NAND2_5 _N16	SA-1	X	X	0	1	X
8	NAND2_5 _N16	SA-0	X	X	0	0	X
9	NAND2_5 _N10	SA-1	X	X	1	0	1
10	NAND2_5 _N10	SA-0	X	X	0	0	X
11	NAND2_5 _N22	SA-1	X	X	X	0	0
12	NAND2_5 _N22	SA-0	X	X	1	X	1
13	NAND2_4 _N7	SA-1	0	0	X	0	X
14	NAND2_4 _N7	SA-0	1	X	0	0	X
15	NAND2_4 _N11	SA-1	1	1	1	0	X
16	NAND2_4 _N11	SA-0	1	X	0	0	X
17	NAND2_4 _N19	SA-1	1	1	0	0	X
18	NAND2_4 _N19	SA-0	X	1	1	X	X
19	NAND2_3 _N11	SA-1	0	1	1	1	X
20	NAND2_3 _N11	SA-0	X	X	0	1	0

```
 21 |         NAND2_3 _N2 | SA-1 |         0 |         0 |         X |         0 |         X |
-----------------------------------------------------------------------------------------------
 22 |         NAND2_3 _N2 | SA-0 |         X |         X |         0 |         1 |         X |
-----------------------------------------------------------------------------------------------
 23 |        NAND2_3 _N16 | SA-1 |         X |         X |         0 |         1 |         0 |
-----------------------------------------------------------------------------------------------
 24 |        NAND2_3 _N16 | SA-0 |         X |         1 |         1 |         0 |         X |
-----------------------------------------------------------------------------------------------
 25 |         NAND2_2 _N6 | SA-1 |         X |         0 |         1 |         1 |         0 |
-----------------------------------------------------------------------------------------------
 26 |         NAND2_2 _N6 | SA-0 |         X |         1 |         1 |         1 |         X |
-----------------------------------------------------------------------------------------------
 27 |         NAND2_2 _N3 | SA-1 |         1 |         1 |         0 |         0 |         X |
-----------------------------------------------------------------------------------------------
 28 |         NAND2_2 _N3 | SA-0 |         1 |         1 |         1 |         0 |         X |
-----------------------------------------------------------------------------------------------
 29 |        NAND2_2 _N11 | SA-1 |         1 |         1 |         1 |         0 |         X |
-----------------------------------------------------------------------------------------------
 30 |        NAND2_2 _N11 | SA-0 |         1 |         X |         0 |         0 |         X |
-----------------------------------------------------------------------------------------------
 31 |         NAND2_1 _N3 | SA-1 |         X |         X |         0 |         0 |         1 |
-----------------------------------------------------------------------------------------------
 32 |         NAND2_1 _N3 | SA-0 |         X |         X |         1 |         0 |         1 |
-----------------------------------------------------------------------------------------------
 33 |         NAND2_1 _N1 | SA-1 |         X |         1 |         1 |         X |         0 |
-----------------------------------------------------------------------------------------------
 34 |         NAND2_1 _N1 | SA-0 |         X |         1 |         1 |         X |         1 |
-----------------------------------------------------------------------------------------------
 35 |        NAND2_1 _N10 | SA-1 |         X |         X |         1 |         0 |         1 |
-----------------------------------------------------------------------------------------------
 36 |        NAND2_1 _N10 | SA-0 |         X |         X |         X |         0 |         0 |
```

هدف از انجام پروژه پیاده سازی منطق D-Algorithm است.

این منطق گویای این موضوع است که در ابتدا گیت های الگوریتم را مشخص کرده سپس باید از ورودی به خروجی برسیم و اینکه در طول برنامه پیش رفته و بر روی سیگنال مورد نظر جایی که stuck خورده را مشخص کند باید بتواند آن را در طول مدار منتشر کند به صورتی که در خروجی دیده شود. انتشار تاخیرها باید در خروجی به صورت D یا D NOT مشاهده شود.

دیدن این تاخیر ها مستلزم این است که اگر stuck at 1 دیدیم D و در صورتی که stuck at 0 را دیدیم D NOT را انتشار دهیم. البته در گیت های مختلف این طریقه انتشار متفاوت است. برای مثال در گیت NOT به یک صورت دیده میشود و یا در FAN OUT به صورت دیگری شاهد این انتشار هستیم.

برای انتشار این تاخیر ها باید NON CONTROLLING هر گیت را در نظر بگیریم.

مثلا برای گیت NOR و OR باید NON COTROLLING را 0 و برای گیت NAND و AND باید NON CONTROLLING را 1 در نظر بگیریم. همچنین باید در نظر داشته باشیم که این منطق برای گیت های XOR , XNOR بی تاثیر است و در گیت‌هایی با حالات مذکور باید سیم دیگر را به صورتی در نظر بگیریم که دارای مقداری باشد به طوری که تاخیر منتشر شود.

برای تعریف این منطق در برنامه مستلزم این هستیم که عمل هر گیت را به صورت اختصاصی در زیر برنامه‌ای جدا تعریف کنیم. چون به صورت دقیق در تعریف پروژه گفته نشده که در کدام گیت STUCK داریم.

همگی سیم های بین این گیت ها به صورت سیگنال تعریف شده این سیگنال ها یکبار تعریف شده و برای اینکه هر بار که باید برنامه را از ابتدا برای یک FAULT اجرا کنیم باید از تابعی مانند ....استفاده کنیم که بارها و بارها این فراخوانی را برایمان انجام دهد و برای تمامی BENCH MARK ها خروجی مناسب تولید کند.

توضیح اجرایی برنامه:

برنامه باید بتواند ورودی Verilog گرفته و خروجی صحیح ایجاد کند. Bench formatای که برایش تعریف میکنیم که برای این موضوع تعریف میکنیم به این صورت تعریف میکنیم که هر گیت خروجی کدام گیت را به عنوان ورودی گرفته و قرار است چه فالتی را منتشر کند.

تعداد ورودی های اصلی در برنامه محدود و سپس تمام ورودی های دیگر برنامه که به صورت داخلی اجرا می‌شوند را به صورت سیگنال تعریف میکنیم. تمام تغییرات حاصل شده بر روی این سیگنال ها را در نظر می‌گیریم همچنین بررسی میکنیم که در سیگنال ها کدام یک دچار fault شده اند.

برنامه دارای ۵ قسمت می باشد به نام های atpg.h, build graph.h, fault line.h, global.h, Verilognode.h, veriloglineparser.h همچنین دارای یک قسمت اصلی بنام main window می‌باشد. به این دلیل که برنامه باید ورودی های VHDL گرفته و روی آن شبیه سازی شود پس باید شبیه به کدهای VHDL تعریف شوند.

تعریف کلاس های Verilog log :

از دستور enum استفاده کردیم. enum یک نوع داده ای تعریف شده توسط برنامه نویس را که به آن نوع داده شمارش می گویند .این دستور به ترتیب در ثابت 1 ، ثابت 2 و ... و ثابت n اعداد صحیح متوالی 0 تا n را قرار می دهد . به صورت پیش فرض مقداردهی متغیرها در این دستور از صفر شروع می شود.

دستور به ثابت false ، عدد صفر و به ثابت true عدد 1 را تخصیص می دهد. حال اگر بخواهیم مقداردهی از عددی غیر از صفر شروع شود باید عدد مورد نظر را مشخص کنیم .در برنامه ما تمام end listها , وmajol ها و end majol وجود دارد.

حالا در برنامه ما خطوط برنامه را میخواند واینکه یک کلاس سازنده دارد به اینصورت که هر nod ای را میخواند میبیند که آیا جزو دسته های تعریف شده هست یا نه! اگر بود که دستور مربوطه را انجام می‌دهد در غیر اینصورت با خطای error روبه رو میشویم!

مقدارهای خروجی را در یک پشته می‌ریزیم. در برنامه دستور کار یکی از متغیرها به این صورت است که node ها را می سازد. (level 1)

حالا باید از چه دستوری استفاده کنیم برای اینکه بتوانیم به کد بفهمانیم که خط ورودی که الان تعریف شده مربوط به چه گیتی است (and – or- nand و یا حتی ورودی و یا ... است) به این منظور از دستور Verilog lineparser استفاده میکنیم یعنی خط به خط را بخوان و ... .

این قسمت شامل دو زیر مجموعه است به نام های public و private ! در قسمت public باید دستورات قابل خواندن و بررسی های کلی را تعریف کرده و در قسمت private باید بررسی کردن زیر مجموعه ها که هرکدام شامل یک گیت است را بررسی کنیم. این دستور در قسمت public نقش component و در قسمت private از نقش port map کردن استفاده میکنیم. اینجا باید تمام کتابخانه هایی که لازم است استفاده کنیم را تعریف کنیم. همچنین باید کلاسی با نام این تابع تعریف کنیم.

این تابع شامل 3 بخش زیر مجموعه است که در هر کدام از آنها باید خط به خط هر string و خط به خط هر node را بخواند. از دستورات زیر استفاده میکنیم تا برنامه متوجه شود که string آن به چه صورت باشد :

حالا وارد دستور خود string میشویم که شامل دستورات if –else میشود. عمل این دستور اینطور است که بررسی میکند ورودی text line را میگیرد بعد با تابع که در بالا تعریف کردیم مقایسه میکند اگر باهم برابر بود که جوابش را به عنوان در تابعی که مربوط به آن دستور است و فراخوانی میکنیم قرار بده.

دستور global.h شامل تعاریف اولیه میشود مثل خروجی هر خط که در برنامه استفاده میکنیم. در این قسمت از برنامه تعریف میکنیم که لاین ها چه فرمتی دارند و چه کاری را انجام می‌دهند .در این قسمت برنامه تمام دستورات Verilog قرار میگیرد و به طور واضح تحلیل میشود که برنامه شامل چه ورودی چه خروجی و یا حتی چه دستوری است. در این قسمت هم در برنامه مشخص میشود که هم میتوان ورودی فایل verilog گرفت و هم می‌توان فایلی که مشخص میکند جه برنامه هایی را در انتها نیاز است نمایش دهیم را به عنوان خروجی تعریف کنیم. در این قسمت مشخص میکنیم که لزوما نباید خروجی عدد باشد یا به صورت یک رشته بلکه میتوان حتی توابع مورد نیاز را در خروجی نمایش داد، ولی این توابع خروجی حقیقی نبوده و نقش سیگنال را به عهده دارند. خروجی این توابع مانند بافر عمل کرده و ورودی گیت یا دستور بعدی را فراهم میکنند.

```
for (i=0; i<linelist_size;i++) {
   LineType curr_line = linelist.front();
   linelist.pop_front();
   if (curr_line.direction == IN) continue;
   if (curr_line.type == PI) continue;
```

```
cout <<"        [Line:"<<curr_line.name;
cout <<" Val:"<<printValue(curr_line);
cout <<" Dir:"<<printDir(curr_line);
cout <<" Visited: "<<curr_line.isVisited;
```

دستور بعدی مربوط به کلاس build graph است .یعنی به نوعی شبیه سازی خود الگوریتم برنامه است که شامل تمام گیت ها و تمام سیگنال های میانی میشود. تمام ورودی ها و خروجی های برنامه و رابطه آنها باهم در آن مشخص شده همچنین دستور انتشار تاخیر در این کلاس تعریف میشود که اصل کار این الگوریتم میباشد. حالا این گراف در یک کلاس ساخته میشود و وقتی ساخته شد نقطه ای را به نام نقطه stuck مشخص میکند که باید بتواند D را تشخیص داده و از آن مسیرآن را انتشار دهد.

در این قسمت از برنامه باید تمامی دستورات مربوط به تابع verilognode.h را وارد کنیم تا تمامی گیت‌ها و سیگنال ها را شناسایی کرده و همه خطاهای احتمالی روی هر سیگنال را اعمال کند. سپس شکل الگوریتم شبیه‌سازی میشود در اینجا باید تمامی خطاها روی هر سیگنال تا انتها propagate شده و خروجی با stuck مورد نظر نمایش داده شود. به همین منظور برای تمامی قسمت ها خروجی به طور جداگانه تعریف شده است.

```
LineList linelist_tmp;
list<LineType>::size_type i,j;
const list<LineType>::size_type linelist_size=linelist.size();
LineType curr_line_i, curr_line_j;

//putLine (linelist_i);
for (i=0; i<linelist_size; i++) {
   curr_line_i = linelist_i.front();
   linelist_i.pop_front();
   if (curr_line_i.type == PI) continue;
   if (curr_line_i.direction == IN) continue; // Only OUT lines

   //cout <<"here"<<endl;
   //putLine(curr_line_i);
   map<LineType,int>::iterator p = line_to_node_map.find(curr_li
   if (p == line_to_node_map.end()) cout<<" [ERROR: NO MATCHING
   else {
  int nodeID = p->second;
  list<string> input_terms_list = rnode[nodeID].input_list;
  list<string>::size_type i_terms=0, no_of_terms = input_terms_l
  linelist_tmp.mlines.clear();
  //cout <<"i_terms="<<i_terms<<" no_of_terms="<<no_of_terms<<en
  for (i_terms=0; i_terms<no_of_terms; i_terms++) {
     //cout <<"here"<<endl;
     string input_line_name = rnode[nodeID].component_name+"_"+
        input_terms_list.front();
     //cout <<input_line_name<<endl;
     input_terms_list.pop_front();
     //cout <<input_line_name<<endl;

     // find input_line_name in linelist and take tht line.
     linelist_j = linelist; // Initialize linelist_j again.
     for (j=0; j<linelist_size; j++) {
        LineType curr_line_j = linelist_j.front();
        linelist_j.pop_front();
```

حالا باید دستور ATPG را بررسی کنیم. در حال حاضر node که در آن اشکال وجود دارد مشخص شده است.

فقط باید مقادیر مختلف را برایش امتحان کنیم مثل ”0“ یا "D" یا " D NOT " یا "U" و یا "X" !

در این قسمت از برنامه از تابع FAULT LINE استفاده کرده و در این مجموعه فراخوانی میکنیم باید خروجی ها را طبق خروجی این برنامه به ما نشان بدهد. در این قسمت با چند تابع جدید کار داریم یکی برای update کردن چون باید هر دفعه مقدار جدید را به ورودی ها بدهیم تا بررسی شود تابعی بنام update change ورودی هایش از نوع line list و current line است.

کارهای آتی:

باید برنامه برای مدارهای بیشتر گسترش داده شود تا باگ و خطاهای احتمالی مشخص شود. لازم است تست الگوهایی برای مدار های ISCAS نیز تولید شود. کارهای گفته شده در این فاز فعلی انجام نشده است زیرا یک جواب صحیح برای مقایسه با آن وجود نداشت. مدار ISCAS c880a ckt تقریبا 60 ورودی اولیه دارد و غیر ممکن است که جواب بصورت دستی چک کنیم ما به یک شبیه ساز fault برای تصدیق تست الگوهای تولید شده نیاز داریم. کل کد باید دوباره ساخته شود و وویطگی‌ها باید قدرتمندتر شده و کلاس‌های بیشتری مورد استفاده قرارگیرد.

درحال حاضر کارهای زیادی برای رسیدن به این مرحله فعلی انجام شده است و باید بطور مناسب برنامه ریزی شده و برنامه فعلی را باز نویسی کنیم . باید جایگزین و الگوریتم‌های بهتری امتحان شود و بهینگی و کارایی را مقایسه کنیم.