Gliwice, 03.06.2018

****

**Laboratorium Programowania Komputerów**

Temat:

Biblioteka Elementów Cyfrowych

|  |  |
| --- | --- |
| autor | Bartłomiej Krasoń |
| prowadzący | dr inż. Jolanta Kawulok |
| rok akademicki | 2017/2018 |
| kierunek | informatyka |
| rodzaj studiów | SSI |
| semestr | 4 |
| termin laboratorium | wtorek, 13:45-15:15 |
| grupa | 6 |
| sekcja | 1 |
| data oddania sprawozdania | 12.06.2018 |
|  |  |

# 1. Temat

Napisać bibliotekę klas implementujących elementy układów cyfrowych oraz prostą aplikację testującą możliwość biblioteki.

# 2. Analiza zadania

## 2.1 Doprecyzowanie tematu

Głównym celem projektu jest zaimplementowanie biblioteki dynamicznie linkowanej (DLL), która dostarcza ustandaryzowany zestaw klas reprezentujących modele podstawowych elementów logicznych oraz narzędzia wspomagające pracę na tych elementach, np. tworzenie z nich złożonych układów cyfrowych. W bibliotece są zaimplementowane następujące elementy logiczne:

1. Bramki logiczne, w tym:
   1. AND
   2. OR
   3. XOR
   4. NAND
   5. NOR
2. Przerzutnik SR
3. Multiplekser
4. Demultiplekser
5. Stała logiczna

Dodatkowo dostarczana jest klasa umożliwiająca tworzenie połączeń między wszystkimi powyższymi elementami oraz klasa funkcji statycznych ułatwiających pracę na układach cyfrowych.

## 2.2 Struktura projektu

Rozwiązanie mojego zadania składa się z dwóch projektów:

1. LogicElementLibrary – to powyżej opisana biblioteka DLL

2. LogicElementClient – jest to prosta aplikacja konsolowa umożliwiająca przetestowanie tworzenia układu cyfrowego za pomocą stworzonej biblioteki, umożliwiająca dodawanie wszystkich powyższych elementów oraz tworzenie miedzy nimi połączeń, z jednoczesnym podglądaniem wartości na pinach\* wszystkich uprzednio dodanych elementów.

\*pin – przewód wejścia/wyjścia elementu układu cyfrowego

## 2.3 Uzasadnienie struktury klas

Założeniem działania biblioteki było ustandaryzowanie modeli elementów cyfrowych, tak aby akcje takie jak dodawanie połączenia danemu elementowi, czy zaktualizowanie jego stanu logicznego mogły odbywać się tak samo dla każdego elementu, czy to bramki, czy multipleksera itd. stąd wszystkie klasy które implementują element logiczny, dziedziczą po wspólnej abstrakcyjnej klasie *LogicElement*. Ponadto w bibliotece znajduje się klasa *Wire* umożliwiająca tworzenie połączeń miedzy elementami.

# 3. Specyfikacja zewnętrzna

## 3.1. Obsługa biblioteki

Poniższa specyfikacja zewnętrzna Biblioteki Elementów Cyfrowych ma za zadanie zobrazować ogólny sposób, w jaki programista może wykorzystać aspekty tej biblioteki. Ukazuje jak w swoim projekcie można realizować poszczególne operacje oraz przedstawia liczne przykłady praktycznego użycia elementów tej biblioteki. Szczegółowa specyfikacja kodu biblioteki znajduje się w rozdziale 4 – Specyfikacja Wewnętrzna.

Biblioteka Elementów Logicznych dostarcza pakiet ustandaryzowanych modeli podstawowych elementów logicznych układów cyfrowych, które mogą zostać wykorzystane w dowolnym projekcie. Istotna jest natomiast informacja jak elementy w bibliotece zostały zamodelowane oraz ustrukturyzowane, co obrazuje poniższa dokumentacja.

**I) PREZENCJA ELEMNTÓW W PROGRAMIE**

Każdy element składa się z linii wejść/wyjść które to mogą zawierać różną liczbę pinów (w zależności od rodzaju czy typu elementu). Metody klasy modelującej elementy logiczne dostarcza nam informację o nich w sposób zmapowany, tj. dostęp jak i odczyt z elementu odbywa się poprzez komunikację ze strukturą mapy, która zawiera nazwy linii wejść/wyjść danego elementu oraz tablice pinów tych linii:

map<string, vector<int>>

vector pinów danej linii

nazwa linii wejść/wyjść (klucz)

Słownik ustandaryzowanej terminologii:

NAZWY LINII wejść/wyjść (klucze):

input - zwykłe wejścia - zazwyczaj gdy element ma tylko jedną linię wejść

data\_input - wejścia danych

control\_input - wejścia sterujące

output - wyjścia

NUMEROWANIE (jeżeli istnieje kilka pinów na jednej linii)

rosnąco od 0:

- od lewej do prawej

- od góry do dołu

każda linia ma własne numerowanie (rozpoczynane od 0)

jeżeli jest jedno wejście/wyjście ma numer 0

Szczegółowa reprezentacja wszystkich modeli elementów cyfrowych: (kolory linii odpowiadają poszczególnym nazwą linii ze słownika)

|  |  |
| --- | --- |
| Stała logiczna | Bramka |
| stała.png | bramka.png |
| Multiplekser | Demultiplekser |
| mux_kolorowy.png | dmux_kolorowy.png |
| Przerzutnik |  |
| przerzutnik.png |  |

\*dla innych wymiarów indeksowanie analogicznie

Poszczególne elementy dostępne są w różnych wymiarach:

1. Bramki: – typ wyliczeniowy enum gateType
   1. dwu-wejściowe
   2. trzy-wejściowe
   3. cztero-wejściowe
2. Multipleksery/Demultipleksery – typ wyliczeniowy enum muxType

<wejść sterujących> x <wejść danych>

* 1. 1 x 2
  2. 2 x 4
  3. 3 x 8
  4. 4 x 16

**II) TWORZENIE NOWEGO ELEMENTU**

W zależności od tego jaki element chcemy utworzyć, z takiego konstruktora skorzystamy. Zaleca się aby do utworzonych obiektów odwoływać się jako do wskaźników klasy *LogicElement*. Poszczególne konstruktory:

1. bezargumentowe
   1. **Latch()**
2. z argumentem typu int
   1. **Constant(int)** – wartość argumentu określa wartość logiczną stałej
3. z argumentem typu muxType
   1. **MUX(muxType)** – typ argumentu określa rozmiary multipleksera
   2. **DMUX(muxType)** – typ argumentu określa rozmiary demultipleksera
4. z argumentem typu gateType
   1. **AND(gateType)** - typ argumentu określa ilość wejść bramki AND
   2. **OR(gateType)** - typ argumentu określa ilość wejść bramki OR
   3. **XOR(gateType)** - typ argumentu określa ilość wejść bramki XOR
   4. **NAND(gateType)** - typ argumentu określa ilość wejść bramki NAND
   5. **NOR(gateType)** - typ argumentu określa ilość wejść bramki NOR

**UWAGA!***Nie można korzystać bezpośrednio z konstruktora Gate()!*

Przykład kodu utworzenia nowej bramki XOR 3-wejściowej

LogicElement \*nElem = nullptr;

gateType gType = gateType::three\_input;

nElem = new XOR(gType);

**III) KOMUNIKACJA Z DANYM ELEMENTEM**

Komunikacja z danym elementem odbywa się za pomocą dwóch podstawowych metod klasy *LogicElement*:

**void setIN(int value, const string lineName = "input", const int i)** – która ustawia *i*-te wejście linii o podanej nazwie *lineName*, na wybraną wartość *value*,

**int getOUT(const int o)** – która zwraca wartość *o*-tego wyjścia z linii “output” (każdy element ma taką linię i zawsze jest tylko jedna linia wyjść),

**void RefreshLogic()** – która służy do odświeżenia wartości na wyjściach (względem wartości na wejściach).

Przykład kodu ustawienia wartość 1 na 5tym pinie wejść danych multipleksera i pobranie wyjścia z przerzutnika SR

LogicElement \*myMux = new MUX(muxType::\_3x8);

LogicElement \*myLatch = new Latch();

myMUX->setIN(1, ”data\_input”, 5);

myMUX->RefreshLogic();

int notQ=myLatch->getOUT(1);

**IV) POBIERANIE STANU/INFORMACJI O ELEMENCIE**

Klasa *LogicElement* dostarcza 3 metody podające następujące informacje:

**vector<int> ReturnAccessibleOutputs()** – zwraca vector indeksów dostępnych wyjść,

**map<string, vector<int>> ReturnAccessibleInputs()** – zwraca mapę linii wejść wraz z dostępnymi na nich pinami. Przez „dostępność” rozumie się to, że do jednego pinu wejścia, może być podłączony co najwyżej jeden przewód w danej chwili. Ta metoda zwraca wszystkie piny wejść, do których nie są podłączone przewody,

**map<string, vector<int>> ReturnIOState()** – zwraca mapę stanów na wszystkich pinach wejść/wyjść dowolnego elementu logicznego.

Powyższe metody mogą się okazać pomocne, przy tworzeniu połączeń, w przypadku gdy chcemy sprawdzić, czy możliwe jest podłączenie przewodu do jakiegoś pinu lub gdy chcemy pokazać użytkownikowi, jakie ma możliwości do wyboru na danym elemencie.

**V) TWORZENIE NOWEGO POŁĄCZENIA**

Do tworzenia połączeń miedzy elementami służy klasa *Wire*, której vector wskaźników zawiera każdy element logiczny układu. Jest to publiczne pole:

**vector<Wire\*> connectList** – które zawiera informację o wszystkich połączeniach wychodzących z danego elementu. Po każdym utworzeniu nowego połączenia, należy dodać je do tego vectora, używając metody Wire::Finalize()

**UWAGA!***Bardzo ważne jest aby każde utworzone połączenie sfinalizować metodą Finalize()!*

Proces tworzenia połączenia odbywa się następująco:

1. Tworzymy obiekt nowego połączenia poprzez wskaźnik
2. Sprawdzamy dostępność wyjścia elementu początkowego połączenia – metoda Wire::checkPinAccessibility(…) umożliwia nam taką akcję
3. Wybieramy sprawdzone wyjście metodą Wire::selectBegin(…)
4. Sprawdzamy dostępność wejścia elementu końcowego połączenia – jak wyżej
5. Wybieramy sprawdzone wejście metodą Wire::selectEnd(…)
6. WAŻNE! gdy zatwierdzamy prawidłowe połączenie używamy metody Wire::Finalize()
7. Gdy chcemy odrzucić połączenie używamy metody Wire::ResetEnd() oraz usuwamy utworzone wcześniej połączenie

Przykład kodu tworzenia połączenia:

Wire \*nWire = new Wire();

if (nWire->chechPinAccessibility(myLatch, ”output”, 1))

nWire->selectBegin(myLatch, 1);

else throw LEexception();

if (nWire->chechPinAccessibility(myMux, ”data\_input”, 5))

nWire->selectEnd(myMux, ”data\_input”, 1);

else throw LEexception();

nWire->Finalize(); //!!! Ważne

**VI) AKTUALIZOWANIE I SPRAWDZANIE UKŁADÓW POŁĄCZEŃ**

Metody biblioteki *Functions* służą do pracy na złożonych układach cyfrowych, znajdują się tam dwie metody do aktualizowania serii połączeń:

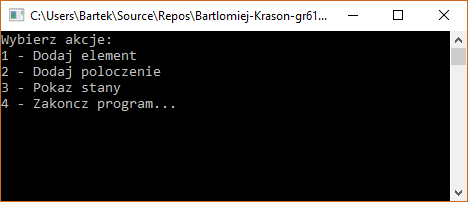
**static void CheckConnectedSeries\_start(LogicElement \*\_elem)** – która sprawdza, czy połączenia od danego elementu \_*elem* nie zapętlają się w układzie (jeśli tak, wyrzuca wyjątek typu RaceCondition)

**static void UpdateConnectedSeries(LogicElement \*\_elem)** – która aktualizuje stan wszystkich elementów tworzących serię połączeń wychodzących z elementu *\_elem*. Przed jej użyciem, należy zawsze wykonać funkcję CheckConnectedSeries\_start(…), aby zapobiec zapętleniu się programu.

## 3.2. Obsługa aplikacji testującej

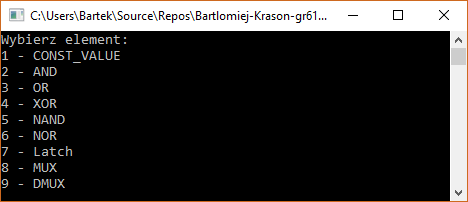
Wraz z biblioteką udostępniana jest prosta aplikacja umożliwiająca przetestowanie możliwości biblioteki. Aplikacja ta dostarcza nam następujące funkcje:

1. Dodawanie nowego elementu do układu
2. Dodawanie połączeń miedzy elementami układu
3. Sprawdzenie stanów logicznych na pinach wszystkich elementów układu

Tak prezentuje się interfejs aplikacji:

**I) DODAWANIE ELEMNTU**

Po wybraniu akcji dodania nowego elementu, wyskakują nam wszystkie dostępne elementy, które możemy umieść w naszym układzie. Wybór danego elementu zatwierdzamy wybraniem odpowiedniej liczby. Możliwości wyboru prezentuje poniższy screen:



**II) DODAWANIE POŁĄCZENIA**

Dodawanie połączeń można rozpocząć, dopiero gdy w naszym układzie znajdą się co najmniej 2 elementy. Po rozpoczęciu tej operacji na ekranie pojawi nam się lista wszystkich dostępnych elementów w naszym układzie. Operacja ta wykonywana jest w odpowiednich krokach:

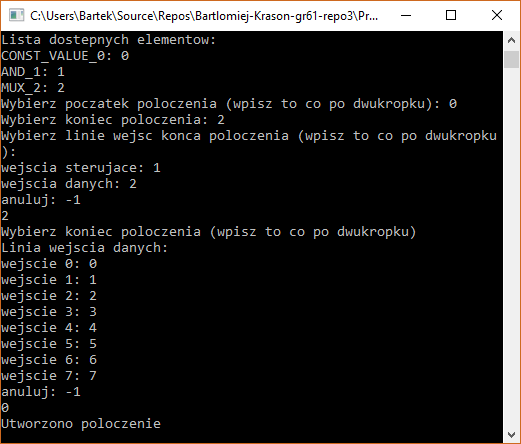
1. Wybieramy z listy element stanowiący początek nowego połączenia
2. Wybieramy z listy element stanowiący koniec naszego połączenia

**UWAGA!** *element końcowy* *musi być inny niż element początkowy!*

1. Wybieramy odpowiedni pin wyjścia elementu początkowego\*
2. Wybieramy linię wejść elementu końcowego połączenia\*
3. Wybieramy odpowiedni pin wejścia wybranej linii elementu końcowego\*

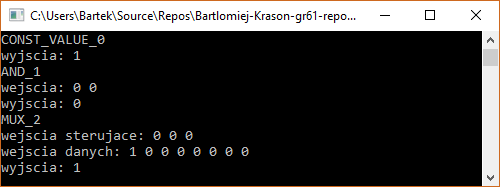
*\*-operacja nie występuje, gdy do wyboru jest tylko 1 pin/linia*

Dodawanie połączenia jest finalizowane odpowiednią informacją, czy połączenie zostało utworzone czy odrzucone, z podaną informacją dlaczego. W dowolnym momencie tworzenia połączenia, możemy je przerwać wybierając *”-1”*, wtedy tworzone połączenie jest anulowane oraz usuwane. Przykład tworzenia połączenia ukazuje poniższy screen:



**III) SPRAWDZANIE STANÓW**

W dowolnym momencie działania naszej aplikacji, możemy podglądnąć wartości na pinach wszystkich elementów naszego układu wybierając z menu aplikacji akcję *”3 – Pokaz stany”*. Realizowane jest to w następujący sposób:



indeksy/wagi bitów

0 1 2 3 4 5 6 7

0 1 2

**UWAGA!** *piny wszystkich elementów wyświetlane są w kolejności rosnącej, toteż najstarszy bit znajduje się zawsze po prawej stronie. Powyższy przykład to ukazuje.*

# 4. Specyfikacja wewnętrzna biblioteki

## 4.1 Klasy biblioteki

Klasa abstrakcyjna wszystkich elementów logicznych jest najistotniejszą klasą w tej bibliotece. To na jej wskaźnikach powinno się pracować na elementach generowanych w naszych projektach. Natomiast każda klasa dziedzicząca po tej klasie wymagała osobnej, autonomicznej implementacji 7 podstawowych metod wirtualnych, umożliwiających tworzenie zaawansowanych układów cyfrowych.

1) Klasa **LogicElement**:

1. chronione pola klasy:

**Element elemFlag** – flaga elementu logicznego

**int indx** – unikalny index elementu

**string name** – nazwa elementu

**int defaultValue** – domyślnie ustawione wartości logiczne na wejściach

1. publiczne pola klasy:

**vector<Wire\*> connectList** – vector wszystkich połączeń wychodzących z tego elementu

**static int globIndex** – globalny indeks gwarantujący unikalność elementów

1. wirtualne metody klasy:

**virtual void setIN(int value, const string lineName, const int i)** – ustawia *i*-te wejście linii *lineName* na wartość *value*, wyrzuca wyjątki:

- IO\_OutOfRange - gdy indeks pinu poza zakresem

- IO\_BadLineRequest - gdy nie posiada danej linii pinów

**virtual int getOUT(const int o)** – zwraca wartość *o*-tego wyjścia linii *”output”*, wyrzuca wyjątek:

- IO\_OutOfRange - gdy indeks pinu poza zakresem

**virtual void RefreshLogic()** – odświeża stan logiczny elementu, interpretując odpowiednio stan pinów wejść na stan pinów wyjść

**virtual void UpdateInputState(int i, string lineName, State s)** – aktualizuje stan dostępności *i*-tego wejścia linii *lineName* na stan *s* (*State::off* – wolne, *State::on* – zajęte), wyrzuca wyjątki jak metoda *setIN(…)*

**virtual vector<int> ReturnAccessibleOutputs()** – zwraca vector indexów dostępnych (o stanie *State::off*) wyjść linii *“output”*

**virtual map<string, vector<int>> ReturnAccessibleInputs**() – zwraca mapę dostępnych pinów wszystkich linii wejść elementu (jeśli w danej linii nie ma już wolnego pinu, nie jest ona dodawana do zwracanej mapy), wyrzuca wyjątek:

- LEexception - gdy nie ma dostępnych wejść

**virtual map<string, vector<int>> ReturnIOState()** – zwraca mapę stanów na wejściach/wyjściach dowolnego elementu logicznego

2) Klasa **Wire**:

1. prywatne pola klasy:

**LogicElement \*beginElement** – element początkowy przewodu (nadający sygnał)

**LogicElement \*endElement** – element końcowy przewodu (odbierający sygnał)

**int value** – wartość logiczna na przewodzie

**int beginPin** – zapamiętuje indeks pinu wyjścia podłączonego elementu początkowego (numerowanie *0* – *n-1*)

**int endPin** – zapamiętuje indeks pinu wejścia podłączonego elementu końcowego (numerowanie *0* – *n-1*)

**string endLineName** – zapamiętuje nazwę wybranej linii wejść podłaczonego elementu końcowego

1. publiczne metody klasy:

**bool checkPinAccessibility(LogicElement \*element, string lineName, int indx)** – sprawdza czy wybrane wejście/wyjście elementu jest dostępne

**void selectBegin(LogicElement \*\_IN, int i)** – wybiera wyjście elementu (które będzie początkiem przewodu)

**void selectEnd(LogicElement \*\_OUT, string lineName, int o)** – wybiera wejście elementu (które będzie końcem przewodu)

**void ResetEnd()** – resetuje stan elemntu końcowego do domyślnego (przed podłączeniem)

**void Finalize()** – finalizuje dodanie połączenia przypisując je do elementu początkowego, oraz aktualizująć dostępność wejscia elemntu końcowego

**void RefreshLogic()** – odświeża stan na przewodzie

**LogicElement \*getSuccessor()** – gdy istnieje zwraca wskaźnik elementu z którym jest połączony (*endElement*)

3) Klasa **Functions**:

1. publiczne metody statyczne klasy:

**static int ParseLogic(int x)** – gwarantuje zwrócenie wartości tylko 0 lub 1

**static int NegateLogic(int x)** – zwraca zanegowaną wartość logiczną

**static void CheckConnectedSeries(LogicElement \*elem, vector<int> vecLE)** – rekurencyjnie sprawdza stan serii połączeń pod względem wystąpienia wyścigu krytycznego (pętli w układzie), działanie opisane w dziale 4.4

**static void CheckConnectedSeries\_start(LogicElement \*\_elem)** – rozpoczyna rekurencyjne sprawdzenie stanu serii połączeń od elementu *\_elem*

**static void UpdateConnectedSeries(LogicElement \*\_elem)** – aktualizuje stany logiczne, elementów tworzących serie połączeń, działanie opisane w dziale 4.4

**static** **bool ElementExists(int indx, map<int, LogicElement\*> container)** – sprawdza czy element o danym indeksie istnieje w kontenerze

**static int BinAddrToDec(vector<int> control\_input)** – zwraca dziesiętną wartość binarnego adresu na danej linii wejść sterujących

4) Klasa **Gate**: (dziedziczy po **LogicElement**)

1. chronione pola klasy:

**GateIO IO** – struktura pinów wejść/wyjść bramki logicznej, wymiary parametrów uzależnione od typu podanego w konstruktorze

Struktura tworząca moduł wejść/wyjść bramki:

**struct GateIO** {

**GateIO(gateType \_type)** – konstruktor budujący moduł w zależności od podanego typu bramki (dwu-/trzy-/cztero-wejściowa), pierw określa wartość *input\_count*, następnie inicjuje o takim rozmiarze vectory *input* oraz *input\_status*

**vector<int> input** – vector wartości logicznych wejść bramki

**vector<State> input\_status** – vector stanów wejść bramki (*off*-wolne, *on*-zajęte)

**int output** – wartość logiczna wyjścia

**int input\_count** – liczba wejść

**~GateIO()** – destruktor modułu

};

1. publiczne metody klasy:

**Gate(gateType \_type)** – konstruktor, przekazujący wartość *\_type* do konstruktora *GateIO()* w celu zainicjalizowania atrybutu *IO*

**void setIN(int value, const string lineName, const int i)** – działa następująco:

- dla *lineName!=”input”*, wyrzuca wyjątek IO\_BadLineRequest

- *i >= IO.input\_count*, wyrzuca wyjątek IO\_OutOfRange

- ustawia *IO.input[i]=value*

- wywołuje metodę *RefreshLogic()*

**int getOUT(const int o)** – zawsze zwraca *IO.output* (nie wyrzuca wyjątku)

**void UpdateInputState(int i, string lineName, State s)** – działa następująco:

- wyrzuca wyjątki jak metoda *setIN(),*

- ustawia *IO.input\_status[i]=s*

**vector<int> ReturnAccessibleOutputs()** – zawsze zwraca vector jednoelementowy z indeksem 0 (wyjście bramki)

**map<string, vector<int>> ReturnAccessibleInputs**() – zwraca mapę zawierającą pary:

**-** ”input”, vector z indeksami wejść których *IO.input\_status==of f*

**-** gdy brak wolnych wejść wyrzuca wyjątek LEexception

**map<string, vector<int>> ReturnIOState()** – zwraca mapę zawierającą pary:

- ”output”, vector jednoelementowym (wartość wyjścia),

- ”input”, vector IO.input

5) Klasa **AND**: (dziedziczy po **Gate**)

1. publiczne metody klasy:

**AND(gateType \_type)** – przekazujący wartość *\_type* do konstruktora *Gate()*, *defaultValue* ustawione na 0, wypełnia *IO.input* wartościami *defaultValue*

**void RefreshLogic()** – ustawia wartość *IO.output=1* gdy wszystkie wartości vectora *IO.input==1*, w przeciwnym wypadku *IO.output=0*

6) Klasa **OR**: (dziedziczy po **Gate**)

1. publiczne metody klasy:

**OR(gateType \_type)** – przekazujący wartość *\_type* do konstruktora *Gate()*, *defaultValue* ustawione na 0, wypełnia *IO.input* wartościami *defaultValue*

**void RefreshLogic()** – ustawia wartość *IO.output=1* gdy co najmniej jedna wartość vectora *IO.input==1*, w przeciwnym wypadku *IO.output=0*

7) Klasa **XOR**: (dziedziczy po **Gate**)

1. publiczne metody klasy:

**XOR(gateType \_type)** – przekazujący wartość *\_type* do konstruktora *Gate(), defaultValue* ustawione na 0, wypełnia *IO.input* wartościami *defaultValue*

**void RefreshLogic()** – ustawia wartość *IO.output=1* gdy ilość wartości vectora dla których *IO.input==1* jest podzielna przez dwa, w przeciwnym wypadku *IO.output=0*

8) Klasa **NAND**: (dziedziczy po **Gate**)

1. publiczne metody klasy:

**NAND(gateType \_type)** – przekazujący wartość *\_type* do konstruktora *Gate(), defaultValue* ustawione na 1, wypełnia *IO.input* wartościami *defaultValue*

**void RefreshLogic()** – ustawia wartość *IO.output=0* gdy wszystkie wartości vectora *IO.input==1*, w przeciwnym wypadku *IO.output=1*

9) Klasa **NOR**: (dziedziczy po **Gate**)

1. publiczne metody klasy:

**NOR(gateType \_type)** – przekazujący wartość *\_type* do konstruktora *Gate(), defaultValue* ustawione na 1, wypełnia *IO.input* wartościami *defaultValue*

**void RefreshLogic()** – ustawia wartość *IO.output=0* gdy co najmniej jedna wartość vectora *IO.input==1*, w przeciwnym wypadku *IO.output=1*

10) Klasa **Latch**: (dziedziczy po **LogicElement**)

1. pola chronione klasy:

**LatchIO IO** – struktura pinów wejść/wyjść przerzutnika SR

Struktura tworząca moduł wejść/wyjść przerzutnika:

**struct LatchIO** {

**vector<State> input\_status** – vector stanów wejść S (indeks-0) i R (indeks-1) (*off*-wolne, *on*-zajęte)

**int S** – wartość logiczna wejścia ustawiającego

**int R** – wartość logiczna wejścia zerującego

**int Q** – wartość logiczna wyjścia prostego

**int notQ** – wartość logiczna wyjścia zanegowanego

};

1. publiczne metody klasy:

**Latch()** – konstruktor przerzutnika, wartości *IO.S* i *IO.R* oraz *defaultValue* ustawione na 0

**void setIN(int value, const string lineName, const int i)** – działa następująco:

- dla *lineName!=”input”*, wyrzuca wyjątek IO\_BadLineRequest

- dla *i==0* ustawia *IO.S=value*,

- dla *i==1* ustawia *IO.R=value*,

- dla innych *i* inaczej wyrzuca wyjątek IO\_OutOfRange

- wywołuje metodę *RefreshLogic()*

**int getOUT(const int o)** – dla:

- *o==0* zwraca *IO.Q*,

- *o==1* zwraca *IO.notQ*,

- dla innych *o* wyrzuca wyjątek IO\_OutOfRange

**void RefreshLogic()** – działa w następujący sposób:

- gdy *IO.R==1* ustawia *IO.Q=0*, *IO.notQ=1*

- gdy *IO.S==1* ustawia *IO.Q=1*, *IO.notQ=0*

- dla *IO.S==1* oraz *IO.R==1* wyższy priorytet ma *R*

- dla *IO.S==0* oraz *IO.R==0* pozostaje poprzedni stan

**void UpdateInputState(int i, string lineName, State s)** – działa następująco:

- wyrzuca wyjątki jak metoda *setIN(),*

- ustawia *IO.input\_status[i]=s*

**vector<int> ReturnAccessibleOutputs()** – zawsze zwraca vector dwuelementowy z indeksami:

- 0 - wyjście *IO.Q*

- 1 - wyjście *IO.notQ*

**map<string, vector<int>> ReturnAccessibleInputs**() – zwraca mapę zawierającą pary:

**-** ”control\_input”, vector z indeksami wejść których *IO.input\_status==off*

**-** gdy brak wolnych wejść wyrzuca wyjątek LEexception

**map<string, vector<int>> ReturnIOState()** – zwraca mapę zawierającą pary:

- ”control\_input”, vector dwuelementowym (0 - *IO.S*, 1 - *IO.R*),

- ”output”, vector dwuelementowym (0 - *IO.Q*, 1 - *IO.notQ*)

11) Klasa **MUX**: (dziedziczy po **LogicElement**)

1. chronione pola klasy:

**muxIO IO** – struktura pinów wejść/wyjść multipleksera, wymiary parametrów uzależnione od typu podanego w konstruktorze

Struktura tworząca moduł wejść/wyjść multipleksera:

**struct muxIO** {

**muxIO(muxType \_type)** – konstruktor budujący moduł w zależności od podanego typu multipleksera (1x2/2x4/3x8/4x16), pierw określa wartość *data\_input\_count* oraz *control\_input\_count*, następnie inicjuje o takim rozmiarze vectory *data\_input*, *data\_input\_status, control\_input* oraz *control\_input\_status*

**vector<int> data\_input** – vector wartości logicznych linii wejść danych

**vector<int> control\_input** – vector wartości logicznych linii wejść sterujących

**vector<State> data\_input\_status** – vector stanów wejść danych (*off*-wolne, *on*-zajęte)

**vector<State> control\_input\_status** – vector stanów wejść sterujących (*off*-wolne, *on*-zajęte)

**int output** – wartość logiczna wyjścia

**int data\_input\_count** – liczba pinów linii wejść danych

**int control\_input\_count** – liczba pinów linii wejść sterujących

**~muxIO()** – destruktor modułu

};

1. publiczne metody klasy:

**mux(muxType \_type)** – konstruktor, przekazujący wartość *\_type* do konstruktora *muxIO()* w celu zainicjalizowania atrybutu *IO*, ustawia *defaultValue* na 0 oraz wypełnia tą wartością vectory *IO.data\_input* oraz *IO.control\_input*

**void setIN(int value, const string lineName, const int i)** – działa następująco:

- dla *lineName=”data\_input”* oraz *i<data\_input\_count* ustawia *IO.data\_input[i]=value*

- dla *lineName=”control\_input”* oraz *i<control\_input\_count* ustawia *IO.control\_input[i]=value*

- dla błędnego argumentu *lineName* wyrzuca wyjątek IO\_BadLineRequest

- dla błędnego argumentu *i* wyrzuca wyjątek IO\_OutOfRange

- wywołuje metodę *RefreshLogic()*

**int getOUT(const int o)** – zawsze zwraca *IO.output* (nie wyrzuca wyjątku)

**void RefreshLogic()** – ustawia *IO.output* na wartość z wejścia *IO.data\_input[addr]*, gdzie *addr* to indeks wskazywany przez bity wektora *IO.control\_input*

**void UpdateInputState(int i, string lineName, State s)** – działa następująco:

- wyrzuca wyjątki jak metoda *setIN(),*

- dla *lineName=”data\_input”* ustawia *IO.data\_input\_status[i]=s*

- dla *lineName=”control\_input”* ustawia *IO.control\_input\_status[i]=s*

**vector<int> ReturnAccessibleOutputs()** – zawsze zwraca vector jednoelementowy z indeksem 0 (wyjście bramki)

**map<string, vector<int>> ReturnAccessibleInputs**() – zwraca mapę zawierającą pary:

- ”data\_input”, vector z indeksami wejść danych których *IO.data\_input\_status==off*

- ”control\_input”, vector z indeksami wejść sterujących których *IO.control\_input\_status==off*

- gdy w danej linii nie ma wolnych wejść, nie dodaje jej do mapy

- gdy całkowity brak wolnych wejść wyrzuca wyjątek LEexception

**map<string, vector<int>> ReturnIOState()** – zwraca mapę zawierającą pary:

- ”data\_input”, vector IO.data\_input,

- ”control\_input”, vector IO.control\_input,

- ”output”, vector jednoelementowym (wartość wyjścia)

12) Klasa **DMUX**: (dziedziczy po **LogicElement**)

1. chronione pola klasy:

**dmuxIO IO** – struktura pinów wejść/wyjść demultipleksera, wymiary parametrów uzależnione od typu podanego w konstruktorze

Struktura tworząca moduł wejść/wyjść demultipleksera:

**struct dmuxIO** {

**dmuxIO(muxType \_type)** – konstruktor budujący moduł w zależności od podanego typu demultipleksera (1x2/2x4/3x8/4x16), pierw określa wartość *output\_count* oraz *control\_input\_count*, następnie inicjuje o takim rozmiarze vectory *output, control\_input* oraz *control\_input\_status*

**int input** – wartości logiczna wejścia informacyjnego

**State input\_status** – stan wejścia informacyjnego (*off*-wolne, *on*-zajęte)

**vector<int> control\_input** – vector wartości logicznych linii wejść sterujących

**vector<State> control\_input\_status** – vector stanów wejść sterujących (*off*-wolne, *on*-zajęte)

**vector<int> output** – vector wartości logicznych linii wyjść

**int control\_input\_count** – liczba pinów linii wejść sterujących

**int output\_count** – liczba pinów linii wyjść

**~dmuxIO()** – destruktor modułu

};

1. publiczne metody klasy:

**dmux(muxType \_type)** – konstruktor, przekazujący wartość *\_type* do konstruktora *dmuxIO()* w celu zainicjalizowania atrybutu *IO*, ustawia *defaultValue* na 0 oraz wypełnia tą wartością vector *IO.control\_input* oraz pole *IO.\_input,* wywołuje metodę *RefreshLogic()*

**void setIN(int value, const string lineName, const int i)** – działa następująco:

- dla *lineName=”input”* oraz *i==0* ustawia *IO.input=value*

- dla *lineName=”control\_input”* oraz *i<control\_input\_count* ustawia *IO.control\_input[i]=value*

- dla błędnego argumentu *lineName* wyrzuca wyjątek IO\_BadLineRequest

- dla błędnego argumentu *i* wyrzuca wyjątek IO\_OutOfRange

- wywołuje metodę *RefreshLogic()*

**int getOUT(const int o)** – dla *o<IO.output\_count* zwraca wartość *IO.output[o]* inaczej wyrzuca wyjątek IO\_OutOfRange

**void RefreshLogic()** – ustawia wszystkie wartości vectora *IO.output* na wartość 1, następnie zanegowaną wartość z wejścia *IO.input*, przekazuje na wyjście *IO.output[addr]* gdzie *addr* to indeks wskazywany przez bity wektora *IO.control\_input*

**void UpdateInputState(int i, string lineName, State s)** – działa następująco:

- wyrzuca wyjątki jak metoda *setIN(),*

- dla *lineName=” input”* ustawia *IO.input\_status=s*

- dla *lineName=”control\_input”* ustawia *IO.control\_input\_status[i]=s*

**vector<int> ReturnAccessibleOutputs()** – zawsze zwraca vector o indeksach *0* – *IO.output\_count-1*

**map<string, vector<int>> ReturnAccessibleInputs**() – zwraca mapę zawierającą pary:

**-** ”input”, vector jednoelementowy z indeksem 0 (wejście informacyjne), gdy *IO.input\_status==off*

**-** ”control\_input”, vector z indeksami wejść sterujących których *IO.control\_input\_status==off*

- gdy w danej linii nie ma wolnych wejść, nie dodaje jej do mapy

- gdy całkowity brak wolnych wejść wyrzuca wyjątek LEexception

**map<string, vector<int>> ReturnIOState()** – zwraca mapę zawierającą pary:

- ”input”, vector jednoelementowym (wartość wejścia informacyjnego),

- ”control\_input”, vector IO.control\_input,

- ”output”, vector IO.output

13) Klasa **Constant**: (dziedziczy po **LogicElement**)

1. prywatne pola klasy:

**int value** – wartość logiczna stałej nadawana w konstruktorze, nie może zostać zmieniona

1. publiczne metody klasy:

**Constant(int IN)** – konstruktor, ustawia sparsowaną wartość *IN* jako wartość stałej *value*

**void setIN(int value, const string lineName, const int i)** – *metoda ignorowana*, nic nie może zmienić wartości stałej

**int getOUT(const int o)** – zawsze zwraca *value* (wartość stałej) (nie wyrzuca wyjątku)

**void RefreshLogic()** – *metoda ignorowana*, nic nie może zmienić wartości stałej

**void UpdateInputState(int i, string lineName, State s)** – *metoda ignorowana*, nic nie może zmienić wartości stałej

**vector<int> ReturnAccessibleOutputs()** – zawsze zwraca vector jednoelementowy z indeksem 0 (wyjście stałej)

**map<string, vector<int>> ReturnAccessibleInputs**() – zawsze wyrzuca wyjątek LEexception, nie można podłączać przewodów do stałej

**map<string, vector<int>> ReturnIOState()** – zwraca mapę zawierającą pary:

- ”output”, vector jednoelementowym (wartość stałej)

## 4.2 Wykorzystane techniki obiektowe

W moim projekcie skorzystałem z następujących technik obiektowych:

1. **Wyjątki** – plik *LEexceptions.h* jest plikiem zawierającym wszystkie klasy wyjątków występujące w Bibliotece Elementów Cyfrowych, oto ich lista:
   1. Klasa **LEexception** – główna klasa wyjątków zapewniająca interfejs ich obsługi:

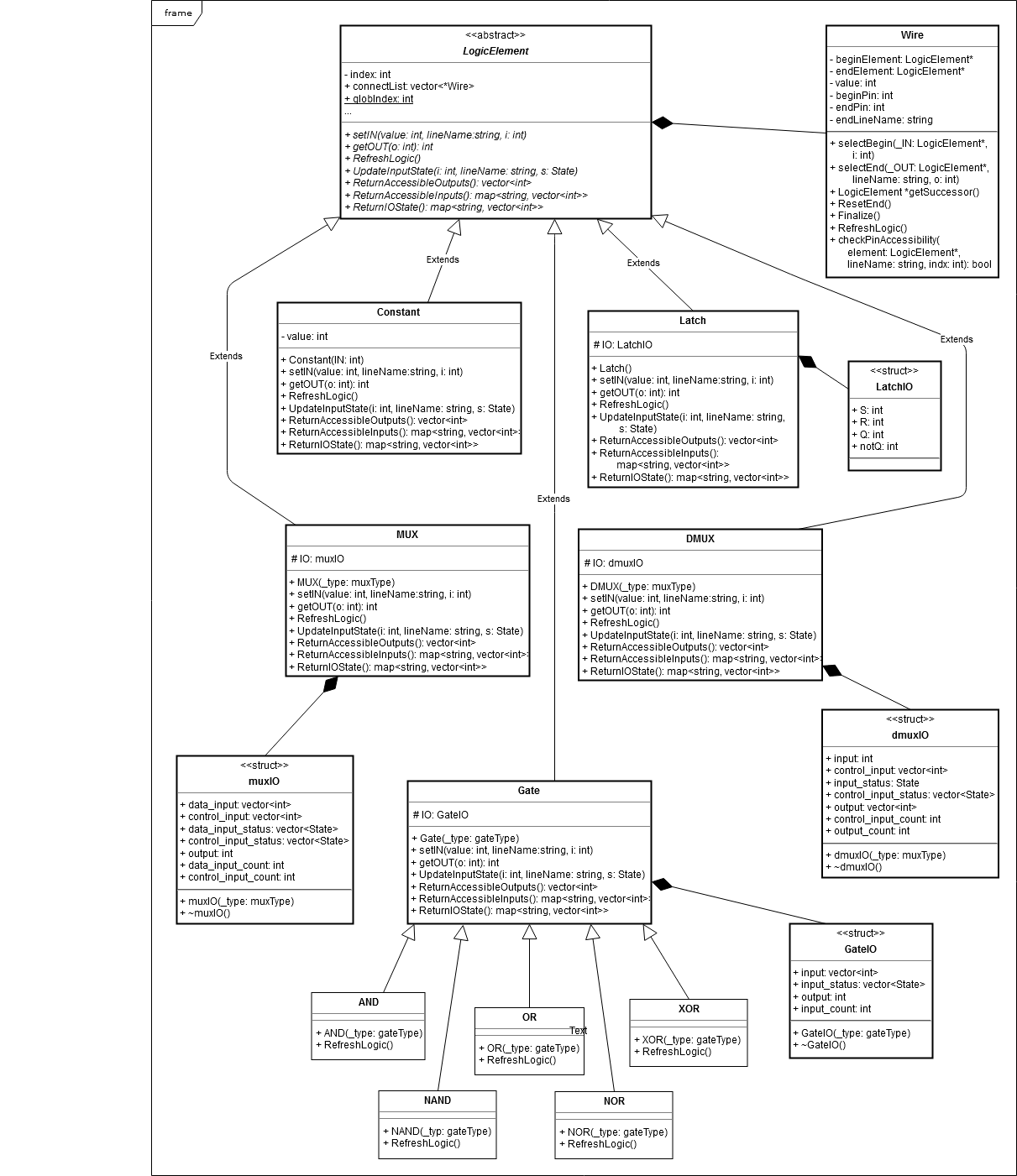
- konstruktor argumentowy **LEexception(string msg)** – zapewnia możliwość edytowania komunikatu wyjątku

- metoda **GetMessage()** – zwraca komunikat wyjątku jako string

* 1. Klasa **CancelEx** – (dziedziczy po LEexception) – wyjątki tego typu sygnalizują rezygnację użytkownika z jakiegoś wyboru
  2. Klasa **RaceCondition** – (dziedziczy po LEexception) – wyjątki tego typu sygnalizują wykrycie wyścigu krytycznego w układzie
  3. Klasa **WrongConstructorParameter** – (dziedziczy po LEexception) – wyjątki tego typu sygnalizują użycie złego parametru przy tworzeniu nowych elementów, tam gdzie wymagane jest podanie parametru w konstruktorze
  4. Klasa **IO\_OutOfRange** – (dziedziczy po LEexception) – wyjątki tego typu sygnalizują odwołanie się do indeksu pinu, spoza zakresu na danej linii
  5. Klasa **IO\_BadLineRequest** – (dziedziczy po LEexception) – wyjątki tego typu sygnalizują odwołanie się do błędnej linii pinów danego elementu

1. **Struktury STL** – w moim projekcie licznie występują:
   1. wektory – które stanowią listy połączeń każdego elementu oraz linie pinów, tam gdzie ich liczba jest większa od 1,
   2. mapy – dzięki którym odbywa się komunikacja o aktualnych stanach elementów, oraz która jest głównym kontenerem elementów układu generowanego w aplikacji
2. **Algorytmy STL** – wykorzystywane algorytmy to m. in. count\_if() używane w metodach *RefreshLogic()* wszystkich bramek logicznych. W wielu aspektach wykorzystałem również iteratory, głównie w celu przeglądania map.
3. **RTTI** – wykorzystałem m. in. do sprawdzenia czy dany element w zbiorze elementów *LogicElement* jest bramką logiczną.

## 4.3 Diagram hierarchii klas



## 4.4 Algorytmy

Zastosowanie dedykowanego algorytmu wymagał problem aktualizacji całych serii połączeń elementów w układzie cyfrowym, bo jak wiemy, zmiana stanu na wyjściu jednego elementu zazwyczaj powoduje zmianę stanu elementów do niego podłączonych. Natrafiamy tutaj na problem synchronizacji. W moim rozwiązaniu zastosowałem algorytm rekurencyjny, którego lista kroków prezentuje się następująco:

*K01:* current\_elem=begin\_elem *//pobierz pierwszy element serii połączeń*

*K02:* **Na** current\_elem **wykonaj** *K03…K07*

*K03:* current\_elem.RefreshLogic() *//odśwież logicznie aktualny element*

*K04:* **Dla każdego** next\_elem **w** current\_elem.ConnectedElements **wykonaj** *K05…K07*

*K05:* value=current\_elem.getOUT() *//pobierz wartość wyjścia*

*K06:* next\_elem.setIN(value) *//ustaw wartość wejścia*

*K07:*  **Wykonaj** *K02* **dla** next\_elem

*K08:* **Zakończ**

Jednakże takie podejście do tego problemu generuje kolejny problem, a mianowicie możliwość zapętlenia się układu. Jest to powszechny problem występujących przy projektowaniu układów cyfrowych nazywany mianem wyścigu krytycznego. Aby nie dopuścić do takiej sytuacji zastosowałem kolejny algorytm, sprawdzający poprawność połączenia:

*//V - vector indeksów elementów podłączonych do serii*

*K01:* V = NULL *// inicjalizacja vectora indeksów*

*K02:* current\_elem=begin\_elem *//pobierz pierwszy element serii połączeń*

*K03:* **Na** current\_elem **wykonaj** *K04…K09*

*K04:* **Dla każdego** index **w** V **wykonaj** *K05*

*K05:*  **Jeśli** index==current\_elem.getIndex() **to**

**Zakończ z porażką**

*K06:* **Dla każdego** next\_elem **w** current\_elem.ConnectedElements **wykonaj** *K07…K09*

*K07:* V.push\_back(current\_elem.getIndex()) *//umieść indeks w vectorze*

*K08:*  **Wykonaj** *K03* **dla** next\_elem

*K09:* V.pop\_back() *//zdejmij indeks z vectora*

*K10:* **Zakończ z sukcesem**

# 5. Testowanie

Program został dogłębnie przetestowany na wiele różnych sposobów. Posłużyła do tego napisana prosta aplikacja *LogicElementClient*, która umożliwia tworzenie układów z licznych i różnorodnych elementów. W ramach testowania napotkano kilka, błędów, lecz zostały one wychwycone i naprawione. Kłopotliwe okazało się rozwiązanie problemu występowania zjawiska wyścigu krytycznego w układzie, gdyż wymagało one utworzenia dedykowanego algorytmu rozwiązującego ten problem. Program został przetestowany pod względem wycieków pamięci biblioteką *<vld>*, która nie wykryła, żadnych negatywnych zachowań.

# 6. Wnioski

Przy realizacji zaprezentowanego projektu wiele się nauczyłem. Pierwszy raz napisałem bibliotekę dll, a nie aplikację, co okazuje się być dobrą nauką, gdyż zastosowanie bibliotek umożliwia wykorzystanie jednego kodu w wielu projektach, co jest niezmiernie przydatne, gdyż może to zaoszczędzić sporo czasu. Głównym założeniem tworzenie tego projektu było szlifowanie umiejętności programowania obiektowego. Myślę, że wypełniłem to założenie, gdyż sporo w moim programie jest powiązań miedzy klasami, czy to na zasadzie dziedziczenia, realizacji czy agregacji. Sporo jest także abstrakcji i polimorfizmu, które ukazują potęgę podejścia obiektowego do programowania. Wykorzystanie technik programowania obiektowego przedstawionych na laboratorium oraz wykładzie również okazało się przydatne, gdyż pozwoliło mi się rozwinąć pod tym względem. Mogę stwierdzić, że poradziłbym sobie bez tych technik, aczkolwiek zajęłoby mi napisanie tego projektu o wiele więcej czasu i byłoby to nierozsądne, gdyż jeżeli mamy do dyspozycji narzędzia, które mają nam ułatwić pracę, to należy z nich korzystać.