Házi feladat

Dokumentáció

Programozás alapjai 2.

Pálinkás Lőrinc Mihály - XB0SMF

Tartalom

1. Feladat

Digitális áramkör

Készítsen egyszerű objektummodellt digitális áramkör szimulálására! A modell minimálisan tartalmazza a következő elemeket:

- NOR kapu
- vezérelhető forrás
- összekötő vezeték
- csomópont

A modell felhasználásával szimulálja egy olyan 5 bemenetű kombinációs hálózat működését, amely akkor ad a kimenetén hamis értéket, ha bementén előálló kombináció 5!

Demonstrálja a működést külön modulként fordított tesztprogrammal! A megoldáshoz ne használjon STL tárolót!

2. Feladatspecifikáció

Feladat általános leírása

A program lehetőséget ad digitális áramkörök szimulálására. A felhasználó áramköröket képes betölteni szöveges file-okból, beállítani a bemeneti jelkombinációt és a kapcsolók állapotát és ez alapján kiolvasni a kimeneti jeleket.

Megvalósított áramköri elemek

A következő elemeket képes szimulálni az áramkör:

- Forrás: állítható LOW és HIGH kimeneti jelekkel, kiolvasható az értéke
- Vezeték: két részt köt össze az áramkörben
- Csomópont: 1 bemeneti jelet több kimeneti irányba tud továbbítani
- Kapu: Több bemenetből képes pontosan 1 kimenetet produkálni. Megvalósított kapuk:
  - AND, OR, NOT
  - NAND, NOR
  - XOR, XNOR
- Lámpa: tárolja a kapott jelet, kiolvasható az értéke
- Kapcsoló: továbbítja a jelet, amennyiben zárt, egyébként LOW jelszintet ad ki

A bonyolultabb elemeket (pl. funkcionális elemek) egyelőre nem implementáljuk, mert könnyen felépíthető ezekből szimuláció során, de ha marad idő, akkor ezeket is megvalósíthatjuk.

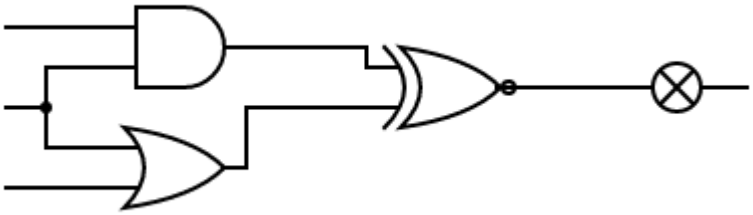
Bemenet formátuma

Az áramkörök felkonfigurálása szöveges file alapján történik. Ebben a felhasználó felsorolja a komponenseket, megadva, hogy hogyan kapcsolódnak. A kapcsolódás megadásához meg kell adni, hogy az adott lába az elemnek melyik csompópontra kapcsolódik. A csomópontokat számok jelölik megadáskor, azonos szám azonos csomópontot jelent. Tehát a konfigurációs file körülbelül így néz ki:

test.txt

```
SOURCE: (1) (2) (3)
AND: (1,2,4)[(…,…,…) …] <- ha több van
OR: (2,3,5) …
XNOR: (4,5,6) …
LAMP: (6)
```

Például erről az ábráról azt tudjuk leolvasni, hogy 3db forrás van jelen, ezek az 1-es, 2-es és 3-as csomópontokra küldik a jeleiket. Emellett van az 1 és 2-es csomópontra kapcsolódó ÉS kapu, mely a 4-es csomópontra küldi a jelét. Hasonlóan kell értelmezni a többit. Ez alapján az alábbi digitális áramkör szimulálható:



Fontos megjegyzés: A szimuláció során az összekötő vezetékeket is csomópontnak tekintünk, így tudjuk könnyen megadni formátumosan a kapcsolódásokat.

A példa azt is mutatja hogy milyen egy egyszerű kapu megadásának például általános formátuma:

GATE_NAME: (IN1, IN2, OUT1) …

Kimenet opciók

A felhasználó képes lekérdezni több információt az áramkörből:

- A lámpák státusza: minden lámpának ki tudjuk olvasni az állapotát, hogy világít-e vagy nem.
- A források státusza: minden forrásnak meg tudjuk adni és ki tudjuk olvasni a jelszintjét.
- A kapcsolók státusza: minden kapcsolónak meg tudjuk adni és ki tudjuk olvasni, hogy zárva van-e vagy sem.

Az áramkör kimenetének megadható, hogy melyik file-ba irányítjuk át a szimuláció kimenetét.

3. Pontosított specifikáció (kiegészítés)

Áramköri elemek I/O pin száma

Az alábbi táblázat mutatja az egyes elemekhez tartozó ki- és bemeneti pin-ek számát, amivel létre lehet hozni:

Áramköri elem típus	Bemeneti pin-ek száma	Kimeneti pin-ek száma
Forrás	0 (csak jelet ad ki)	1 (egy jelet ad ki)
Csomópont	1 (ahonnan kapja a jelet)	≥1 (tetszőlegesen sok helyre küldhet jelet)
Vezeték	1 (csomópont speciális esete)	1 (csomópont speciális esete, amikor 1 kimenet van)
Kapu	≥1 (minden kapu legalább egy bemenetből állít elő kimenetet, támogatva lesz több mint 2 bemenetű AND, OR, stb.)	1 (minden kapu 1 logika jelet állít elő)
Lámpa	1 (1 helyről fogad jelet)	0 (nem ad ki jelet, csak eredmény tárolásra van)
Kapcsoló	1 (1 helyről fogad jelet)	1 (1 helyre továbbít jelet)

Felhasználói felület

A felhasználó számára van biztosítva egy egyszerű menü, melyben a következő műveleteket tudja elvégezni:

- Áramkör betöltése: képes megadni egy file nevét, és innen betölteni egy áramkört
- Bemeneti adatok beállítása: meg tudja adni a források bemeneti jeleit illetve a kapcsolók állását
- Kimeneti file beállítása: meg tudja adni hogy melyik file-ba irányítsa át a kimenetet, alapvetően a std::cout-ra küldi a szimuláció kimenetét
- Szimuláció: végrehajtja a szimuláció lefuttatását
- Kilépés: leállítja a programot

4. Terv

Információ áramlása

A digitális áramkör szimulálása során az információ áramlását fogjuk modellezni. Mielőtt a tervezett objektummodell be lesz mutatva, azelőtt elengedhetetlennek tűnt, hogy előbb az információ áramlásának modelljét jellemezzem, mert ennek jelentős kihatásai lesznek az egyes osztályok tervezésére, meghatározó hogy hogyan is kommunikálnak egymással az objektumok.

A fő ötlet és inspiráció a tervezéskor a Számítógépes architektúrák tárgy keretében megismert adatáramlásos modell volt. Ha egy áramkör szimulálását vesszük figyelembe, akkor jön a gondolat, hogyan is tudjuk, hogy honnan kell kezdeni a jelek kiértékelését?

Kezdetben csak a források jele adott, a többi áramköri elemnek nem tudhatjuk, mert korábbi elemek jelére is építhetnek. Emiatt először ezek jeleit ismerve tudjuk elindítani az információ áramlását, hiszen azon elemek, amelyeknek minden lába forrásra kapcsolódik, rögtön kiértékelhetőek, majd ezek után az ezekre kapcsolt elemek, és így tovább.

Ez a viselkedés nagyon szoros párhuzamot mutatott az adatáramlásos adatfeldolgozási modellel, így erre alapozva fejlesztettem ki az adatok feldolgozásának menetét. Az áramköri kapuk, kapcsolók, stb. a precedenciagráfnak az egyes csúcsai, melyek kiértékelnek a bemeneti jel alapján egy kimeneti jelet, amit aztán tovább küldenek a következő csúcsoknak, jelen esetben egy másik áramköri elemnek.

A működése nagy vonalakban a következő: a szimuláció során mindig számon tartunk egy „aktív” FIFO-t. Ebben a FIFO-ban mindig azon elemeket tartjuk, amelyeknek minden jele meg van, tehát kiértékelhetőek. Amikor egy ilyen elemet kiértékelünk, akkor minden kapcsolódó áramköri elemnek jelezzük, hogy eggyel nőtt a „kész” bemenetek száma. Ha ez eléri a bemenetek számát, akkor meg van minden szükséges bemenete, tehát be tudjuk rakni az aktív FIFO-ba, ahol aztán ki lesz értékelve.

Így sorjában minden áramköri elemre kiértékeli és beállítja a megfelelő jelértéket, amíg van ilyen elem.

Adatáramlásos működés problémái és megoldások

Tervezés során előjött több probléma is, ami elkerülhetetlen az adatáramlásos modellből fakadóan, bár ezeknek egy része főleg az áramkör megadásának kiszámíthatatlanságából adódik. A következőekben ezekere adok megoldásokat.

1. Elszigetelet, kiértékeletlen elemek:

Tegyük fel hogy az alábbi file-t kapjuk felkonfiguráláskor:

```
SOURCE: (1)
LAMP: (2)
```

Ezen az egyszerű látni hogy mi a baj: az 1-es csomópontra kapcsolódó forrásból sosem fog eljutni a 2-es csomópontra kapcsolódó lámpába a jel. Ez azt is jelenti, hogy a kimeneti értéke nem lesz értelmes, a lámpa nem mér valós értéket.

Ez alapvetően nem is probléma, mert (mint később látjuk) minden pin alapvetően LOW jelet kap, ami egyezik azzal, ami a valóságban lenne, hogy nincs rákötve tápra = LOW jel. Ez azonban akkor baj, ha mondjuk 2 LOW jelből mondjuk egy NAND HIGH jelet kell képezzen, de ezt nem teszi meg, mert sose lesz kiértékelve.

Ha valóságban elképzeljük, akkor ez gyakorlatilag egy „levegőben lebegő”, áramkörtől független lábat jelent valamilyen áramköri elemre nézve.

Megoldás: felkonfiguráláskor futtatunk egy próba szimulációt, mely során figyeljük, hogy le lett-e szimulálva minden elem. Amennyiben ez teljesül, akkor minden rendben, az áramkör biztosan helyesen kiértékel minden elemet. Amennyiben van olyan elem, ami nem lesz kiértékelve, az jelzi, hogy ez a baj van valahol, ezt jelezzük a felhasználó felé, és az áramkör el lesz utasítva.

2. Visszacsatolás

Másik szembetűnő problémát az adatáramlásos modellel az alábbi kapcsolások szemlélteti:

1: Önhivatkozásos visszacsatolás:

```
SOURCE: (1)
AND: (1, 2, 2)
LAMP: (2)
```

A fenti áramkör szemlélteti ezt a problémát, mert itt egy kapu bemenete függ a kimenetétől, emiatt hiába is van minden rendesen összekötve, nem fog kiértékelődni.

Megoldás: Ez a probléma elsőre nehezen kezelhetőnek tűnhet, de a megoldás már létezik. A fő probléma, hogy nem kiértékelhető, hiszen saját magára alapszik. Azonban ezt az előző rész tesztje ezt is el fogja ugyanúgy kapni, hiszen sosem lesz kiértékelve a 2-es csomópontra kapcsolódó lába az elemnek.

2: Stabil visszacsatolás

```
SOURCE: (1)
NOT: (1, 2) (2, 3)
LAMP: (3)
```

A jelenlegi példában gyakorlatilag egy D flip-flopot valósítunk meg. Jogosan merül fel a kérdés, hogy mégis hogyan lenne lehetséges itt kezelni a visszacsatolást. Jelenlegihez hasonló esetekben nincsen baj a visszacsatolásból, mert ugyanazt a stabil jelet küldi vissza, mint amit kapott.

Megoldás: Amennyiben a visszacsatolás azonos jelt küld vissza, akkor nincs probléma, az elem nem lesz újra kiértékelve, hiszen nem változtat semmilyen szempontból a kapcsoláson.

3: Instabil visszacsatolás

```
SOURCE: (1)
NOT: (1, 2) (2, 3) (3, 1)
LAMP: (3)
```

Hogyan értékeljünk ki egy ilyen áramkört? Több kérdés is felmerülhet, hiszen, ebben a visszacsatolás ellentétes jelet küldi vissza, mint amit kapott. Bár elsőre ez a helyzet problémásnak tűnhet, hiszen sok szélső eset is lehetséges, de a feloldásához egy észrevétel kell.

Vegyünk egy tetszőleges ilyen áramkört. Ekkor fel tudjuk osztani stabil és instabil kapcsolású részlegekre. Forrás mindig stabil lesz, mert nincs kimenete. Ez viszont vagy direkt kapcsolódik az instabil részre, vagy stabil kapcsolású részeken keresztül, amig hasonlóan funckionálnak jelen esetben a forráshoz (stabil a kimenet, nincs instabil visszacsatolás). Ez azonban azt jelenti, hogy instabil visszacsatolás kezdetekor egy stabil jelforrást kapó csomópontra küldünk vissza ellentétes jelet (mert instabil).

Valóságban elképzelve ez hasonlóan funckionál mint ha a földet összekötjük a táppal, rövidzárat alkotva. Ezek alapján a következő megoldásra jutottam.

Megoldás: Ha egy visszacsatolás ellentétes jelet küld vissza, akkor biztosan keletkezne rövidzár az áramkörben, tehát amennyiben ez az eset következik be, akkor jelezzük a felhasználó fele, hogy rövidzár történt (megadva a csomópontot), és szimuláció nem ad információt a kimenetről, mert értelmetlen lenne.

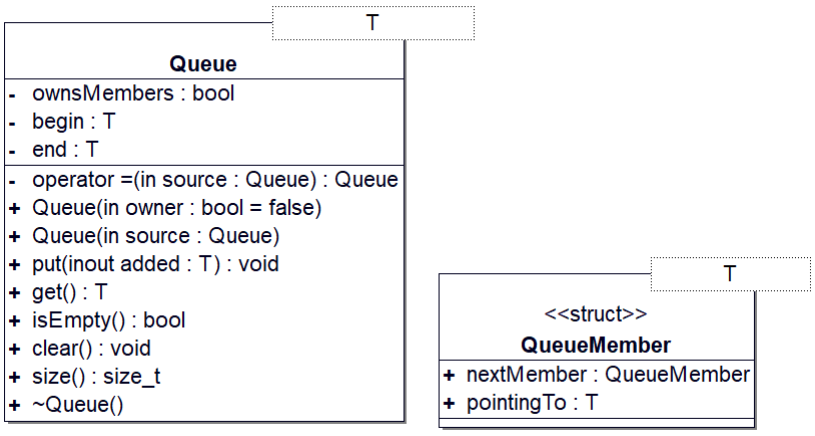
Objektummodell

Az áramköri elemek modellezése során adódott hogy az egyes áramköri elemek egymással kommunikáló objektumokként viselkednek. Ezek alapján dolgoztam ki a modellt, szemléletesség érdekében ez most bottum-up módon szeretném bemutatni, kezdve legalulról, lépésekben felépítve az elemek modelljét.

Queue osztály

Mielőtt a konkrét áramkör elemek modellezéséről beszélnék, fontosnak tartom, hogy előbb egy általánosabb szerkezetet mutassak be, amelyet több osztály is fel fog használni.

Ahogyan az információáramlás modelljének leírásában mondtam, egy aktív FIFO-ban kell tárolnunk a kiértékelendő elemeket. Ebből egyértelmű lett, hogy egy láncolt listában érdemes ezeket tárolnunk. Azonban a tervezés során kiderült, hogy nem csak erre az egy funkcióra kell egy láncolt list szerkezet, emiatt úgy döntöttem hogy egy generikus Queue osztályt hozok létre, melyet több helyen is újra fogunk használni:



Eltérés azonban van egy sima láncolt listához képest. Először is a Queue osztály alapvetően pointereket tárol objektumokra, ezek mindig dinamikusan foglaltak lesznek. Emiatt be kellett vezetnem egy „tulajdonos” attribútomot. Ennek a fő oka, hogy nem egyszer fogunk több Queue-ból ugyanarra az objektumra hivatkozni (más pointereken

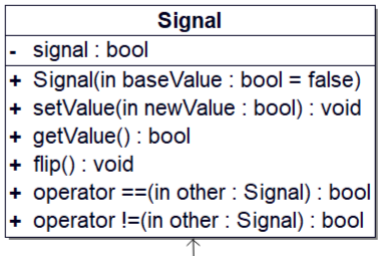
keresztül, máshogyan kezelve), és mivel ennek a Queue-nak felelőssége lehet törölni a memóriát, ezért létrehozáskor tudnia kell, hogy felelős-e az elemeiért.

Ezen túl a legtöbb művelet a szokásos, mint egy láncolt listában, a get() kiveszi az elejéről az elemet, a put() berak egyet a végére. Folyamatosan számon tartja a méretét, illetve lekérdezhető hogy üres-e.

Fontos!: A másoló konstruktor egy olyan queue-t hoz létre, ami ugyanazon elemeket tartalmazz, viszont NEM SZABADÍTJA fel őket, automatán, nem tulajdonos! (Ennek oka, hogy többször is kell kiszedni és valami végezni ezeken az objektumokon)
Assign operátor külső használat elkerülésére le van tiltva, ezért privát.

Signal osztály

A digitális áramkörökben jelszinteket mérünk le, emiatt döntöttem, hogy érdemes lenne egy saját osztályként működjön maga a logikai jel, ennek az eredménye lett a Signal, a digitális jelet modellező osztály:



Funkcionalitás szempontjából elég egyszerű osztály, létre tudunk hozni vele jelet, beállítani és kiolvasni, megfordítani és összehasonlítani. A jeleket boolean értékként tároljuk, mert azonos viselkedésű a digitális jelértékekkel. (true ~ HIGH (1), false ~ LOW (0))

Pin osztályok

A tervezés során következő felmerülő osztály a Pin volt, ezen keresztül tudnak kommunikálni az áramköri elemek. Két fő funkciót látnak el: egyrész jelet tárolnak, melyet ki lehet olvani, másrészt jelet adnak át a másik pin-nek.

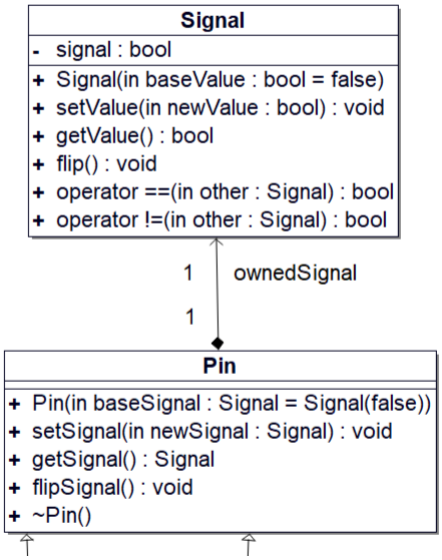
Tervezés elején az tűnt célszerűnek, hogy egyfajta Pin osztály létezzen, viszont hamar egyértelmű lett, hogy nem elég, az információ áramlásának modellje miatt szükséges volt két részre bontani.

Ennek fő oka, mint ahogy a specifikációban is látszik, az hogy alapvetően két szerepet tölthet be egy láb: információt fogad, azaz bemenetként viselkedik, illetve információt továbbít, azaz

kimenetként viselkedik. Ezek funkcionalitásokhoz szükséges műveletek teljesen másak, más információt szükséges tárolni.

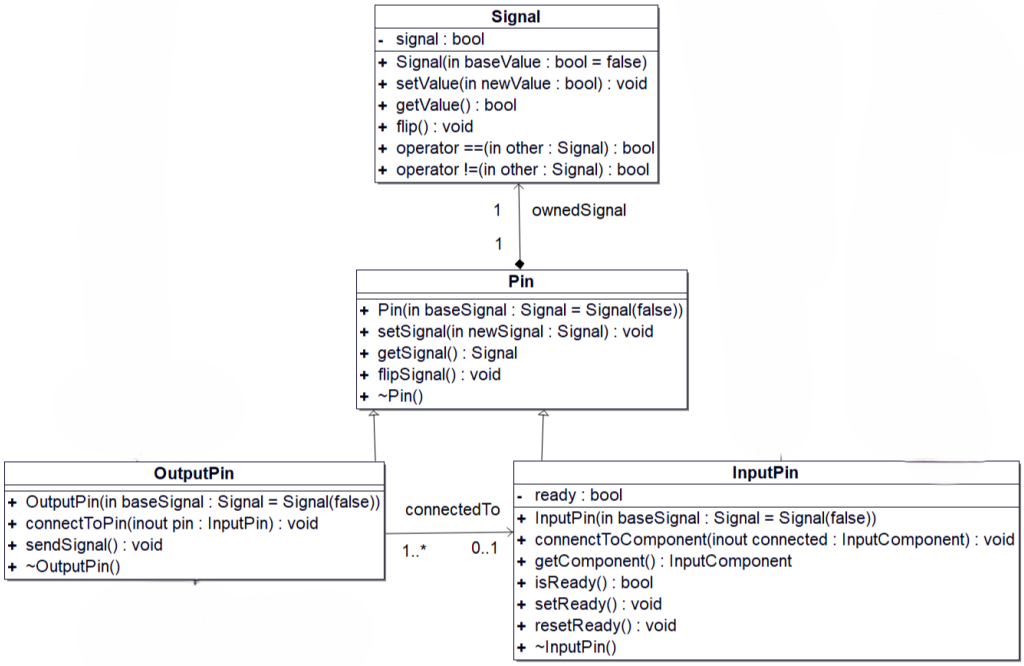
Emiatt döntöttem úgy, hogy bár fog létezni egy közös ősosztály, melyben a közös funckionalitás van megvalósítva, azonban két külön osztályként célszerűbb megvalósítani őket.

A sima Pin ősosztály tartalmaz minden azonos viselkedését egy elem lábának:



Van egy jele, amit tárol, alapértelmezetten ez LOW. Ezt lehet állítani és olvasni, illetve megfordítani (öröklés miatt van csak destruktor, memóriát nem kell felszabadítania, ez a többi osztályban is hasonló okok miatt, hogy a további öröklés esetén ne legyen baj vele).

Az InputPin és OutputPin osztályok végzik el a konkrét kimeneti és bemeneti szerep megvalósítását:



Az OutputPin osztály fő bővítése, hogy képes kapcsolódni InputPin-hez és neki jelet küldeni, egyébként ugyanolyan mint a sima Pin.

Az InputPin ezzel szemben nem másik Pin-hez, hanem egy áramköri elemhez kapcsolódik (ld. később), ennek jelzi, hogy kapott jelet egy OutputPin-től, ami ennek hatására ellenőrzi, hogy készen áll-e.

Minden InputPin tárolja a ready változóban, hogy készen áll-e információfeldolgozásra, ezt lehet beállítani és resetelni, illetve az állapotát lekérdezni (az elem részéről fogjuk). A resetelés lehetősége újraszimuláláskor lesz fontos, hiszen ekkor minden áramköri elem bemeneti lábainak készenlétét resetelni kell, hogy ne tévesen kerüljön be az aktív FIFO-ba (illetve a visszacsatolást is ezzel lehet ellenőrizhetni).

Component osztályok

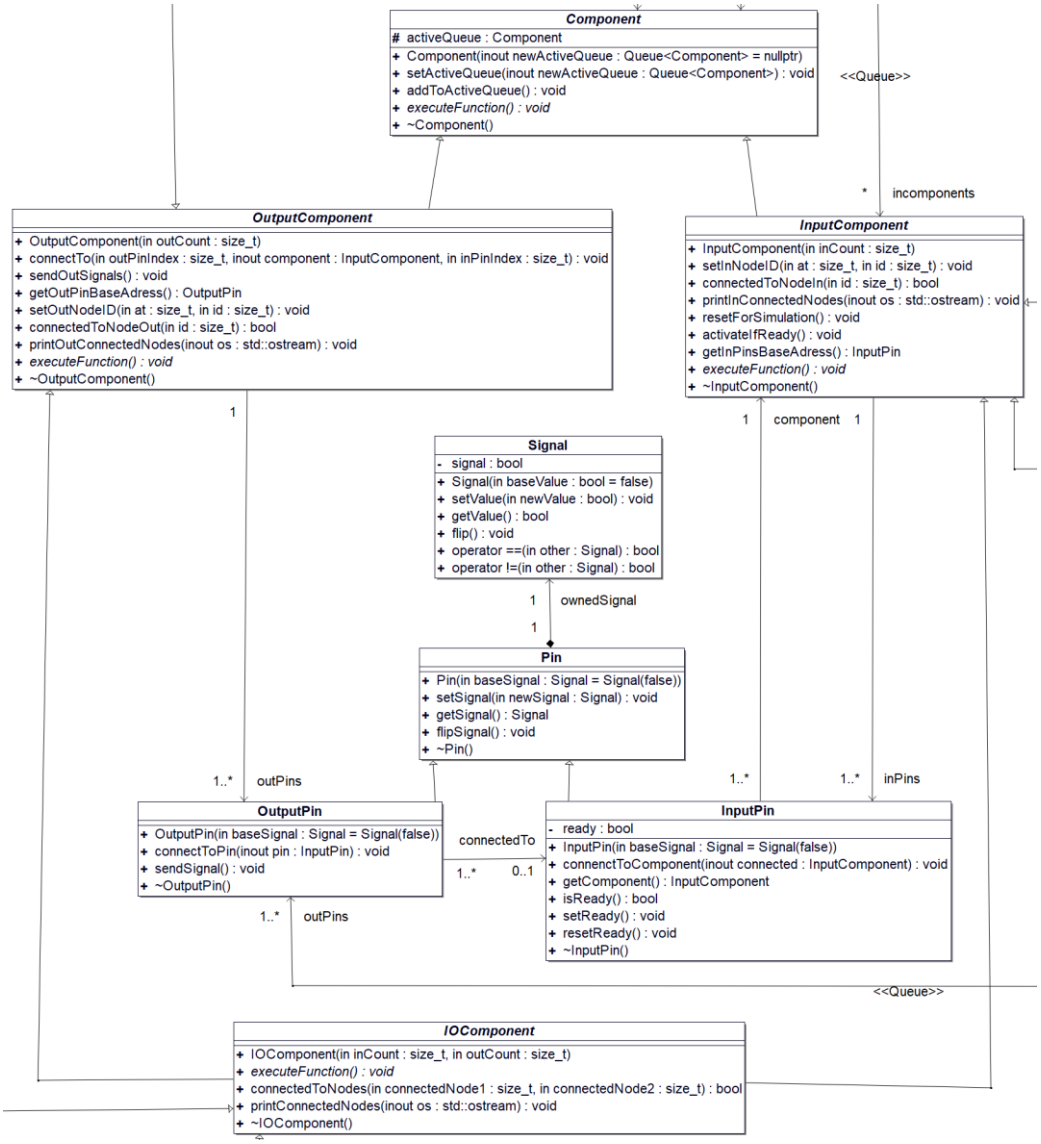
Az áramköri elemek tervezés során a legcélszerűbb egy közös ősosztály volt, melyen keresztül egy heterogén kollekcióban tudjuk majd tárolni őket. A másik fő ok, hogy minden elemre ugyanazokat az általános műveleteket végezzük el: hozzáadjuk az aktív FIFO-hoz, ha készen áll kiértékelésre, illetve kiértékeljük és elvégezzük a jelkiküldést, amennyiben van kimenete.

Azonban viszonylag hamar szembetűnt, hogy ez az egyetlen közös osztály nem lesz elég. Ahogyan a Pin-ek leírásában is láthattuk, szükség volt két külön célú pin osztályra, attól függően hogy milyen szerepet töltenek be. Elsőre célszerű lehet, hogy minden elemnek legyen ki és bemenete, és vegyük 0-nak, ha nincs ilyen. Csak ezzel a problémába ütközünk, hogy sok elemre értelmetlen műveletek lesznek értelmezve, jobbnak tűnt, ha minden elemre választhatjuk hogy melyik funkciót tölti be, ha esetleg mindkettőt, akkor megkapja mindkét funkcionalitást.

Ezáltal hoztam létre az InputComponent és OutputComponent interfész osztályokat, melyeken keresztül egy áramköri elem megkaphatja vagy egyik vagy másik funckiót, esetleg mindkettőt, de akár ha valamelyiket bizonyos okok miatt máshogy kell implementálnunk (ld. később Node osztály) akkor nem okoz gondot, egyszerűen máshonnan kapja az egyik interfészt.

Gyakran azonban mindkettő interfészt közösen használjuk, emiatt döntöttem amellett, hogy egy közbenső teljesen absztrakt, mindösszeg kódírást megkönnyítő harmadik, IOComponent osztály is létrehoztam.

Ezen döntések eredményeként kaptuk meg az alábbi osztályokat:



Az ábrán olvashatunk le néhány függvényt, ami csak a konfiguráláskor és kiíráskot használatos, a többi, alapvetőbb fontosságú funckióra érdemes most koncentrálnunk (ezek változhatnak még fejlesztés alatt).

A Component osztály gyakorlatilag semmiben nem mutat újat a fent leírtakhoz képest, egyedül annyiban csak, hogy az aktív FIFO címét tároljuk, amihez aztán majd hozzá kell adnunk.

Mind az InputComponent, mind az OutputComponent osztályok tömbként tárolják a Pin-jeiket. Ezt a döntést az befolyásolta, hogy létrehozás után szinte minden elemnek konstans a lábszáma, emiatt ez tűnt a legyegyszerűbbnek. Plusz az indexeléssel könnyen lehet azonosítani a bemeneteket (pl. ha bonyolultabb elemekkel bővítjük esetleg később a modellt pl. muxi, stb.). Az egyetlen kivételt a konstans jellegre a Node class jelenti, de ezt majd ott tárgyalom részletesen.

Az OutputComponent osztály fő funkciója a jelkiküldés, míg az InputComponent class-nak aktivizálás (ha készen áll), illetve a resetelés szimuláció előtt.

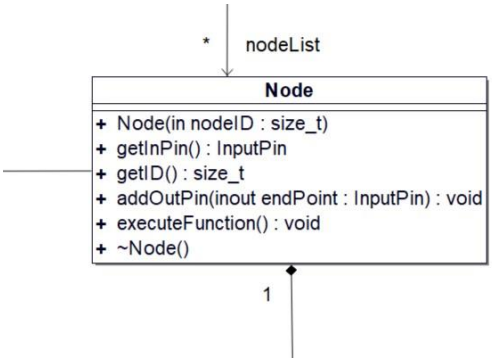
Az IOComponent csak köztes class, nem ad extra funckiót (többszörösen örököl a másik kettőtől).

Node osztály

A Node osztály fő feladat a csomópont funkcionalitás lefedése. Ezt az áramkör megadásakor a felhasználó félig implicit adja meg, hiszen számokkal jelzi, hogy melyik csomópontra kapcsolódnak az egyes elemek. Mint azt majd a konfigurálás leírásakor megfigyelhetjük, emiatt bár bemenetként konstans funkcionál (1 bemenete van ahonnan fogad jelet), mégis a kimenete folyamatosan nőni fog (ahogy több kaput csatlakoztatunk a kimenetére).

Emiatt a kimenetét nem tudjuk sima tömbben tárolni. Szerencsére rendelkezésünkre áll már ezek tárolására és bővítésére alkalmas osztály, a Queue. Emiatt azonban az OutputComponent interfész nem felel meg neki, magának kell implementálnunk a kimenetet. Mivel ez csak egy láncolt listában tárolja a kimeneti pineket, minden funkcionalitás szinte ugyanaz (bejárás persze máshogy történik, de ez semmivel nem rosszabb jelküldéskor, hiszen amúgy is az egész tömbön végig kell mennünk hogy minden pinről kiküldjük a jelet).

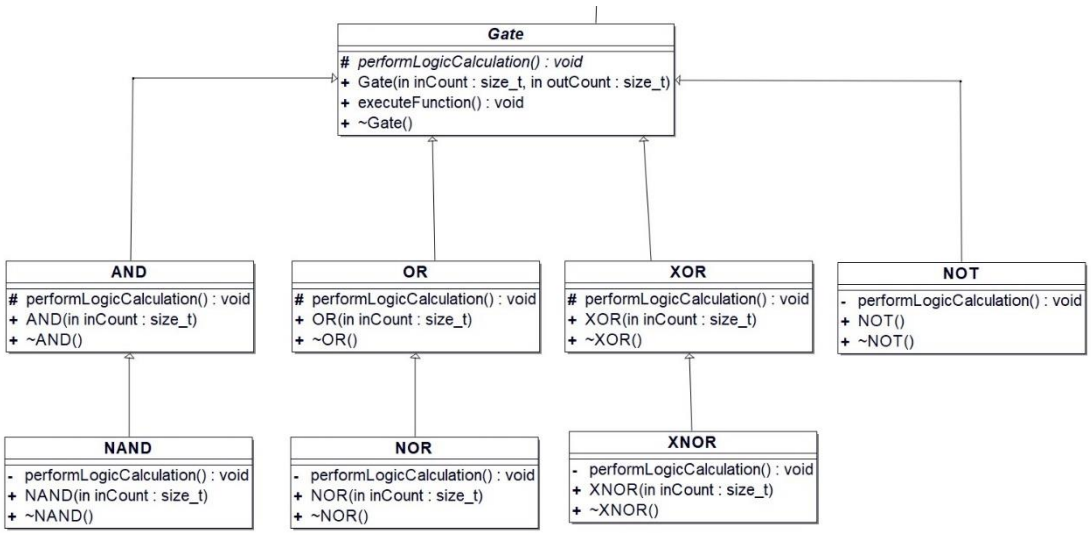
Ennek az eredménye az alábbi osztály („- id: size_t” most nem látszik, mert a felhasznált UML program reverse engingeer funckiója ezt nem adta hozzá):



(A két Pin-es elnevezésű tagfüggvény összekötéskor lesz hasznos, emellett lekérdezhetjük a hozzárendelt számot, ami ID-ként funckionál.)

Gate osztályok

A kapu osztályok valósítják meg a logikai kapuk működését. Ezek az IOComponent-től örökölnek, és nem rendelkeznek különleges saját funkciókkal, egyedül csak előállítják a logikai bemenet alapján a kimentet, amit tovább küldenek:



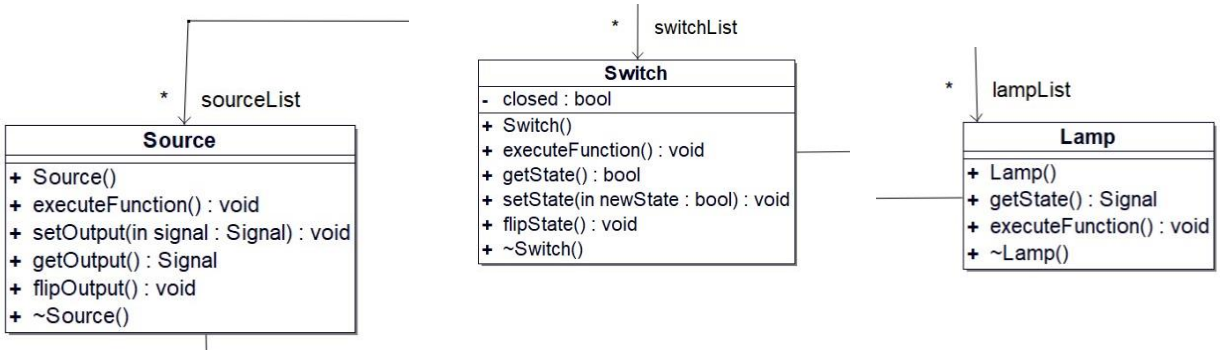
A Gate osztály főleg a többi IOComponent-től szolgál elkülönítésként, csak köztes, absztrakt osztály szerepe van.

Elsőre furcsa lehet, hogy a negált kapukat öröklés útján hozzuk létre, de itt (bár ábra nem jelzi), korlátozó öröklést alkalmazunk. Ennek fő oka, hogy minden szempontból azonosan funckionálnak, csak a kimenetüket kell megfordítani, emiatt egyszerűbb újrahasználni a sima kapukban definiált függvényeket.

Mint ahogy le is olvasható, a kapuknak tetszőleges bemenete lehet. Ennek a fő oka, hogy semmilyen komplikációt nem okoz ennek az implementálása, hiszen minden kapunak csak 1 kimenete lesz, tehát az első n-1 megadott csomópont mind bemenet, ráadásul ezek szerepe szimetrikus, ezért a sorrend mindegy.

Periféria jellegű osztályok

A specifikációban is látható volt, de alapvetően 3 elem ki és bemenetét figyeljük: a források, a kapcsolók és a lámpák. Ezek közösen hasonlóan periféria jelleget mutatnak, emiatt az alábbi módon terveztem meg őket:



A forrás osztály az egy kimeneti Pin-jén tárolja a jelét, ezt lehet állítani és lekérdezni, illetve jelet küldeni vele.

A kapcsoló tárolja, hogy zárt-e vagy nem, ez alapján dönti el, hogy milyen jelet ad tovább, ha zárt, akkor a bemeneti Pin jelét, amúgy a kimeneti pin jelét. Az állapota ennek is állítható, és lekérdezhető.

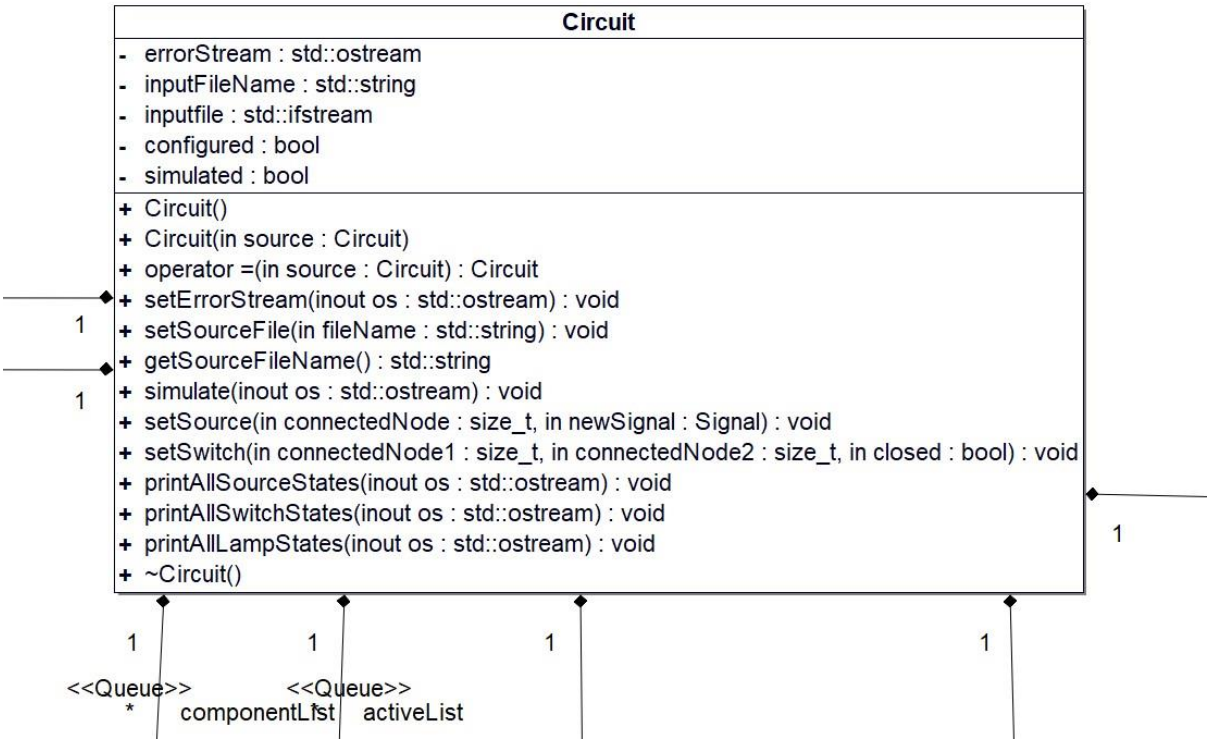
A lámpa csak az egyetlen bemeneti Pin-jén fogadja, más funckiója nincs. Az állapotot itt is le tudjuk kérdezni.

Mindhárom osztályhoz tartoznak inserter-ek is, melyekkel ki lehet íratni őket egy output stream-re.

Circuit osztály

A legfőbb osztály, és egyben a feladat „végterméke”, ezt tudja a felhasználó felkonfigurálni, adatokat beállítani, és szimulációkat futtatni rajta.

A specifikációban kitűzött célok mellett egyéb, nehezebb feladatokra is lehet esetlegesen képes, és nem kizárt, hogy fejlesztés alatt ne legyen még bővítve az API, de egyelőre kezdetlegesen az alábbi funkciókat tervezem biztosan megvalósítani:



A következők az ábráról is leolvasható, de érdemes lejegyezni őket:

- Állítható error streamje van, ha pl. file-ba karjuk átirányítani (ez alapértelmezetten a std::cerr)
- Állítható a bemeneti file-ja, amelyből felkonfigurálunk (ennek nevét is tárolja)
- Mindig tárolja, hogy fel van-e konfigurálva és le van-e szimulálva (feles konfig és szimulálás kikerüléséhez)
- Szimuláláskor megadjuk hogy hova irányítsa a kimenetet
- Be tudjuk állítani bizonyos csomópontra kapcsolt források/kapcsolók állapotát (jelzi, ha nincs megadott)
- Ki lehet vele íratni az egyes periféria elemek adatait (lesz ehhez is inserter operátor, amivel tudja a szimulátor kiírni)

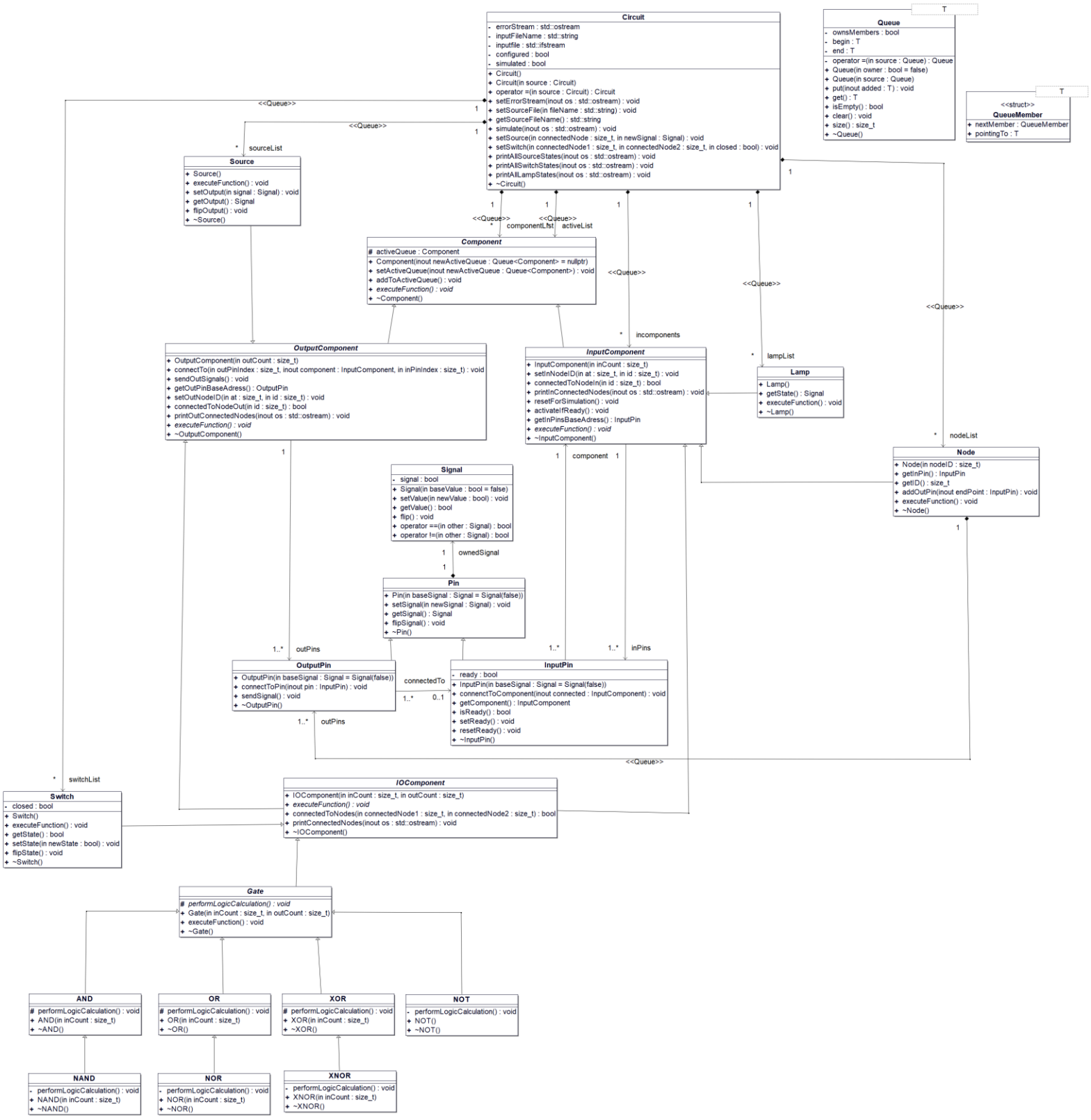
Alap konstuktorja üresen hozza létre. Felkonfigurálás automatán történik (általában első szimuláció során), de csak akkor konfigurál fel, ha szükséges, feleslegesen nem konfigurál újra.

Az áramkör maga felelős a dinamikus memória kezelésért: a copy konstruktor és assign operátor kezelik megfelelően a memóriát, azaz a copy egy teljesen azonos, független másolatot hoz létre, és ehhez hasonlóan az egyenlőség operátor is, de ez törli az előző áramkör memóriáját előtte.

Az elemek tárolására megint a Queue osztályt fogjuk felhasználni, terv szerint az alábbi listák lesznek számon tarva benne (az ábrán a vonalak ezeket jelzik, csak nem teljesen látszanak):

- Component list: Az összes komponensre pointer, ez a memória felszabadításakor lesz fontos, ez törli őket.
- Active list: Ez valósítja meg az aktív FIFO-t, minden elem ide kerül, ha készen áll a kiértékelésre
- Node list: A csomópontok listája, építéskor innen keresi elő melyikhez kell adni újabb output Pin-t
- InputComponent list: Ez a szimuláció előtti reseteléshez fog kelleni, ide rakunk minden resetelendő bemenettel rendelkező objektumra mutatót
- Switch list: a kapcsolók mutatóit tárolja a kiolvasáshoz és állításhoz (ezeket másik listán nem tudjuk elérni, és nem is lenne máshonnan elérni hatékony)
- Lamp list: a lámpák mutatóit tárolja a kiolvasáshoz (ezeket másik listán nem tudjuk elérni, és nem is lenne máshonnan elérni hatékony)
- Source list: a források mutatóit tárolja a kiolvasáshoz és állításhoz (ezeket másik listán nem tudjuk elérni, és nem is lenne máshonnan elérni hatékony)

Teljes ábra


Circuit
- errorStream : std::ostream
- inputFileName : std::string
- inputfile : std::ifstream
- configured : bool
- simulated : bool
+ Circuit()
+ Circuit(in source : Circuit)
+ operator =(in source : Circuit) : Circuit
+ setErrorStream(inout os : std::ostream) : void
+ setSourceFile(in fileName : std::string) : void
+ getSourceFileName() : std::string
+ simulate(inout os : std::ostream) : void
+ setSource(in connectedNode : size_t, in newSignal : Signal) : void
+ setSwitch(in connectedNode1 : size_t, in connectedNode2 : size_t, in closed : bool) : void
+ printAllSourceStates(inout os : std::ostream) : void
+ printAllSwitchStates(inout os : std::ostream) : void
+ printAllLampStates(inout os : std::ostream) : void
+ ~Circuit()
Queue
- ownsMembers : bool
- begin : T
- end : T
- operator =(in source : Queue) : Queue
+ Queue(in owner : bool = false)
+ Queue(in source : Queue)
+ put(inout added : T) : void
+ get() : T
+ isEmpty() : bool
+ clear() : void
+ size() : size_t
+ ~Queue()
<<struct>>
QueueMember
+ nextMember : QueueMember
+ pointingTo : T
Source
+ Source()
+ executeFunction() : void
+ setOutput(in signal : Signal) : void
+ getOutput() : Signal
+ flipOutput() : void
+ ~Source()
Component
activeQueue : Component
+ Component(inout newActiveQueue : Queue<Component> = nullptr)
+ setActiveQueue(inout newActiveQueue : Queue<Component>) : void
+ addToActiveQueue() : void
+ executeFunction() : void
+ ~Component()
OutputComponent
+ OutputComponent(in outCount : size_t)
+ connectTo(in outPinIndex : size_t, inout component : InputComponent, in inPinIndex : size_t) : void
+ sendOutSignals() : void
+ getOutPinBaseAdress() : OutputPin
+ setOutNodeID(in at : size_t, in id : size_t) : void
+ connectedToNodeOut(in id : size_t) : bool
+ printOutConnectedNodes(inout os : std::ostream) : void
+ executeFunction() : void
+ ~OutputComponent()
InputComponent
+ InputComponent(in inCount : size_t)
+ setInNodeID(in at : size_t, in id : size_t) : void
+ connectedToNodeIn(in id : size_t) : bool
+ printInConnectedNodes(inout os : std::ostream) : void
+ resetForSimulation() : void
+ activateIfReady() : void
+ getInPinsBaseAdress() : InputPin
+ executeFunction() : void
+ ~InputComponent()
Lamp
+ Lamp()
+ getState() : Signal
+ executeFunction() : void
+ ~Lamp()
Node
+ Node(in nodeID : size_t)
+ getInPin() : InputPin
+ getID() : size_t
+ addOutPin(inout endPoint : InputPin) : void
+ executeFunction() : void
+ ~Node()
Signal
- signal : bool
+ Signal(in baseValue : bool = false)
+ setValue(in newValue : bool) : void
+ getValue() : bool
+ flip() : void
+ operator ==(in other : Signal) : bool
+ operator !=(in other : Signal) : bool
Pin
+ Pin(in baseSignal : Signal = Signal(false))
+ setSignal(in newSignal : Signal) : void
+ getSignal() : Signal
+ flipSignal() : void
+ ~Pin()
OutputPin
+ OutputPin(in baseSignal : Signal = Signal(false))
+ connectToPin(inout pin : InputPin) : void
+ sendSignal() : void
+ ~OutputPin()
InputPin
- ready : bool
+ InputPin(in baseSignal : Signal = Signal(false))
+ connenctToComponent(inout connected : InputComponent) : void
+ getComponent() : InputComponent
+ isReady() : bool
+ setReady() : void
+ resetReady() : void
+ ~InputPin()
Switch
- closed : bool
+ Switch()
+ executeFunction() : void
+ getState() : bool
+ setState(in newState : bool) : void
+ flipState() : void
+ ~Switch()
IOComponent
+ IOComponent(in inCount : size_t, in outCount : size_t)
+ executeFunction() : void
+ connectedToNodes(in connectedNode1 : size_t, in connectedNode2 : size_t) : bool
+ printConnectedNodes(inout os : std::ostream) : void
+ ~IOComponent()
Gate
performLogicCalculation() : void
+ Gate(in inCount : size_t, in outCount : size_t)
+ executeFunction() : void
+ ~Gate()
AND
performLogicCalculation() : void
+ AND(in inCount : size_t)
+ ~AND()
OR
performLogicCalculation() : void
+ OR(in inCount : size_t)
+ ~OR()
XOR
performLogicCalculation() : void
+ XOR(in inCount : size_t)
+ ~XOR()
NOT
- performLogicCalculation() : void
+ NOT()
+ ~NOT()
NAND
- performLogicCalculation() : void
+ NAND(in inCount : size_t)
+ ~NAND()
NOR
- performLogicCalculation() : void
+ NOR(in inCount : size_t)
+ ~NOR()
XNOR
- performLogicCalculation() : void
+ XNOR(in inCount : size_t)
+ ~XNOR()
<<Queue>>
componentLlst
activeList
incomponents
lampList
nodeList
component
ownedSignal
outPins
inPins
connectedTo
switchList
sourceList

5. Megvalósítás (NEM VÉGLEGES)

Fejlesztés során történt változások

A tervezés során elég részletesen ki lett találva az objektummodell, ennek köszönhetően nagy részében nem is történt jelentős változások fejlesztés során, a legtöbb interface-e változatlan maradt szinte, esetlegesen csak a nevek változtak.

Azonban pár helyen voltak fontosabb változások, ezeket itt szeretném megemlíteni:

- setSourceFile függvény most már setSchematicFile-ra hallgat (setSource névvel volt zavaró ütközés)
- A források, kapcsolók és lámpák jeleit és állapotait direktbe is ki lehet olvasni (Signal és bool változókba).
  Ennek oka az egyszerűbb tesztelés, illetve logikusnak tűnt hogy ezt is ki tudja olvasni a Circuit osztály felhasználója.
- A Component osztálynak lett “simulated” tárolt bool információja, ami megadja, hogy le lett-e szimulálva az adott elem (Szemantikai ellenőrzéshez kell). Emiatt három új tagfüggvényével lehet ezt set-elni, reset-elni és kiolvasni.
- Az InputComponent és OutputComponent osztályok pin tömbjének címei már nem elérhetők getter-en keresztül, hanem egy adott indexű tömbelemet megadva, tömbön kívüli indexelést ellenőrizve lehet a pin-ekhez hozzáférni.
- Queue osztály kapott iterátorokat, hogy könnyebb legyen minden elemen végrehajtani egy funckiót, anélkül hogy másolgatni kellene a listát.
- Exception osztályok: ezeknek a szerepe a kivételek típusos kezelése, legtöbb osztály csak egy üzenetet hordoz, melyet ki akarunk írni, de esetlegesen tudhatnak többet is.
  A legtöbbet privát függvények használják, de van pár ami publikus is, amiatt érdemes említeni.
  Az egyes kivétel osztályok, és mikor használjuk őket:
  - MessagedException: közös ősosztály, fő funkciója, hogy ne kelljen a közös viselkedéseket újraimplementálni.
  - NonExistentConnection: a Pin osztályok dobják, ha nincs elvárt kapcsolódó elem (Pin vagy Component, típustól függően).
  - ShortCircuit: rövidzár létrejövésekor dobja az OutputPin osztály.

- MatchingComponentNotFound: A periféria elemek beállításakor és kiolvasásakor dob ilyet a Circuit, amennyiben nem létező perifériát akarunk elérni.
- NoFileGiven: Felkonfigurálásban dobjuk, ha nincsen megadva vagy nemlétező file-t akarunk olvasni.
- NonExistentLineType: Felkonfigurálásban dobjuk, ha nem létező típust adtak meg.
- IncorrectSyntax: Felkonfigurálásban dobjuk, ha szintaktikailag helytelen a megadott áramkör (rossz helyen felesleges vagy helytelen karakter).
- IncorrectPinCount: Felkonfigurálásban dobjuk, ha a megadott pin-ek száma nem felel meg az megadott típusnak.
- NonExistentType: Felkonfigurálásban dobjuk, ha valamiért nem kapott típus a sor.
- ConversionError: Felkonfigurálásban dobjuk, ha dynamic_cast nem sikerült, ezzel jelezve, hogy helytelen a típus, amit megadtunk.
- UnsimulatedComponent: Felkonfigurálásban dobjuk, ha találunk ellenőrzéskor egy nem leszimulált elemet.

6. Tesztelés (NEM VÉGLEGES)

Tesztelés menete

Az osztályok helyes működésének a tesztelést alapvetően a gtest_lite és a memtrace-en keresztül valósítom meg, ezeken keresztül ellenőrizzük, hogy minden a terv szerint működik.

A tesztelés kitalálásakor a fő cél az volt, hogy a Circuit osztály interface-én keresztül a működés sima legyen, hiszen ez a fő terméke a feladatunknak, ezt tudja a felhasználó hasznosítani.

Pontosan emiatt a többi osztálynak nincsenek is saját tesztjei, hiszen ezek működése garantált, amennyiben helyesen működik a Circuit osztály is, felesleges lenne külön tesztelni azokat is.

A megadott tesztesetek mellett lehetőség van egy szimpla menüvel vezérelt tesztelőre is, melyen keresztül saját file-okat is be tudunk tölteni, illetve saját bemenettel le is tudunk szimulálni.

Tesztesetek

A tesztesteket csoportosítjuk aszerint, hogy hogyan ellenőrzik a Circuit osztály helyes működését:

- SANITY:
  - Ezek az alapvető tesztek, melyek a nagyon alapvető funciókat ellenőrző műveleteket tartalmaznak (konstruktorok, másolás, konfig file beállítás, hiba stream beállítása)
- COMPONENT_CHECK:
  - Ezek az egyes áramköri elemek rendeltetésszerű működését ellenőrzik (első a kapukra, többi a periféria elemek működésére fókuszál)
- ERRORS:
  - Ezek az egyes futásidejű hibák helyes kezelését ellenőrzik (helytelen szintaxis, rövidzár, szimulálatlan elemek, stb.)
- COMPLEX_CIRCUITS:
  - Ezek komplexebb áramkörök, melyekkel bemutatjuk az implementált típusok együttműködésével kialakítható funkcionális elemeket (pl. multiplexer, dekóder, komparátor, stb.)