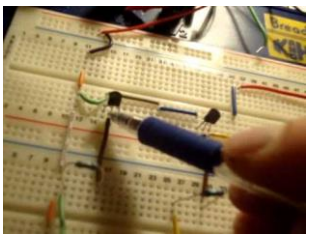
Design Patterns 1

Practicumopdracht: Full adder

Opdrachtomschrijving:



Binnen de elektrotechniek wordt vaak gebruik gemaakt van logische circuits. Deze circuits zijn een aaneenschakeling van logische binaire componenten. Een circuit heeft één of meerdere inputs waaraan de logische '0' of logische '1' wordt meegegeven. Daarnaast heeft het circuit één of meerdere output's, ook wel probes genoemd. Deze probes geven ieder een logische waarde als output.

Een voorbeeld van een logisch circuit is de full adder. Deze kan binaire getallen optellen.
De opdracht is om een simulator te maken van logische circuits als de full adder. Deze simulator gebruikt tekstbestanden in een bepaald formaat om het circuit in te lezen om deze vervolgens te simuleren.

Deze opdracht zullen jullie in duo's maken en heeft een weging van 1,5 EC.
Hieronder wordt nog toelichting gegeven over de de requirements van de opdracht. Zie voor een gedetailleerde weging de rubrics op blackboard.

Requirements

- Werk in duo's.
- Simuleer de volgende logische componenten: AND-, OR-, NOT-, NAND-, NOR-, XOR-poort.
- Beschik over input nodes.
- Beschik over output nodes (probes).
- Zorg voor een propagation delay op de logische componenten. Dit is de tijd die een component nodig heeft om de uitgangsbit stabiel te krijgen. In werkelijkheid is dit ongeveer 15 nanoseconden.
- De structuur van het circuit moeten ingelezen kunnen worden uit de bestanden die aangeleverd zijn op blackboard.
- De simulatie moet uitgevoerd kunnen worden wanneer er een bepaalde input gegeven is.
- Er wordt gelet op nette code en het juiste gebruik van verschillende design patterns. Er moeten dan ook verschillende design patterns toegepast worden en uitgelegd kunnen worden.
- De opdracht moet gemaakt worden in Java of C#, andere object geörienteerde talen enkel in overleg met de practicumdocent.
- Het circuit moet visueel weergegeven kunnen worden.
  Hierbij hoeven niet per sé blokjes met lijntjes daartussen weergegeven te worden, maar wel de verschillende nodes, van welk type ze zijn en met welke nodes ze verbonden zijn.
  Mooiere user interface wordt natuurlijk gewaardeerd!
  Let op: Bij een console application is een console.log() je user interface. Nodes mogen daar dan natuurlijk zelf geen gebruik van maken!
- De ingangswaardes moeten runtime aangepast kunnen worden. De ingangswaardes in de bestanden zijn enkel defaults.

Nice to have

Bovenstaande requirements geven je de mogelijkheid om een voldoende te behalen. Bovenstaande punten moeten dan ook allemaal behaald zijn voordat gekeken wordt naar de nice to haves.

- Geef visueel weer hoe de waarden door het circuit lopen.
- Bereken de propagation delay van het totale circuit.
- Laat aan de hand van extra testbestanden zien dat je je applicatie getest hebt.
- Laat aan de hand van nuttige unittests zien dat je je applicatie getest hebt.
- Maak het mogelijk om verschillende circuits aan elkaar te schakelen.

Beoordeling

Lever op blackboard de volgende onderdelen in:

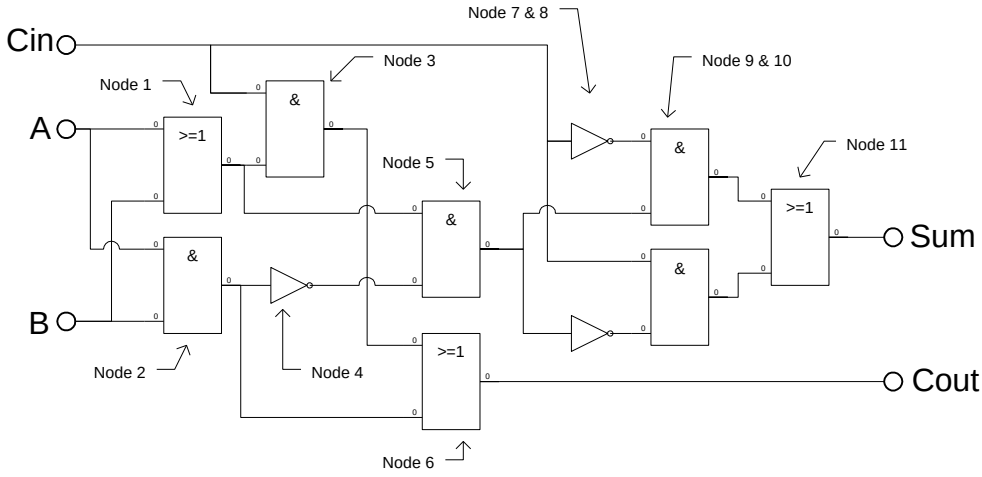
- Diagrammen, deadline: eind week 3
    - Een domeinklassediagram van je applicatie
    - Een sequencediagram van je simulatie
- Je sourcecode, deadline eind week 7
  Dit bevat onder andere:
    - Je applicatiecode
    - Eventuele extra testbestanden
    - Eventuele extra unittestprojecten

In de les van week 7 dien je jullie product de demonstreren aan de docent. Op dit moment wordt je cijfer bepaald aan de hand van de rubric die op blackboard staat.

Bijlagen

Bijlage A: Full adder circuit

Een full adder circuit is bedoeld om binaire getallen bij elkaar op te tellen en ziet er uit als in onderstaande afbeelding.



Links zie je de inputs, A, B en Cin. Dit zijn bitjes die ieder een 0 of een 1 kunnen zijn. Wanneer de simulatie gestart wordt zullen de bitjes door de verschillende nodes heen lopen en zullen uiteindelijk de outputs Sum en Cout gevuld zijn.

Om te verduidelijken hoe binaire getallen opgeteld worden, zal eerst de vergelijking gemaakt worden met een optelling uit het decimale stelstel.
Stel dat de volgende optelling gemaakt moet worden:

$$\begin{array}{r} 47 \\ 56+ \\ \hline ??? \end{array}$$

Hiervoor moeten 2 decimale optellingen gebruikt worden.
Eerst hebben we:

- A = 7
- B = 6
- Cin = 0

De Cin is de carry. Dit is het restgetal van de vorige optelling. Aangezien we nog geen vorige optelling hebben is deze 0.

7 + 6 = 13. De sum wordt dan 3 en de Cout wordt 1.

$$\begin{array}{r} \color{red}{1} \\ 47 \\ 56+ \\ \hline \mathbf{3} \end{array}$$

Nu hebben we nog één berekening te maken om tot de uitkomst te komen.
Hier hebben we de volgende getallen:

- A = 4
- B = 5

- Cin = 1, dit is het getal dat uit de vorige berekening de Cout was.

4 + 5 + 1 = 10. De sum wordt dan 0 en de Cout wordt 1.

$$\begin{array}{r} {\scriptstyle 11} \\ 47 \\ 56 + \\ \hline \mathbf{103} \end{array}$$

Binair werkt dit hetzelfde en kan je zelf met 3 full adders nagaan dat de volgende bewering juist is:

$$\begin{array}{r} {\scriptstyle 101} \\ 101 \\ 101 + \\ \hline \mathbf{1010} \end{array}$$

Bijlage B: Input file

```
# Circuit1.txt
#
# Full-adder. Deze file bevat een
correct circuit
#
#
#
# Description of all the nodes
#
A:            INPUT_HIGH;
B:            INPUT_HIGH;
Cin:      INPUT_LOW;
Cout:     PROBE;
S:            PROBE;
NODE1:    OR;
NODE2:    AND;
NODE3:    AND;
NODE4:    NOT;
NODE5:    AND;
NODE6:    OR;
NODE7:    NOT;
NODE8:    NOT;
NODE9:    AND;
NODE10:   AND;
NODE11:   OR;

#
#
# Description of all the edges
#
Cin:      NODE3,NODE7,NODE10;
A:            NODE1,NODE2;
B:            NODE1,NODE2;
NODE1:    NODE3,NODE5;
NODE2:    NODE4,NODE6;
NODE3:    NODE6;
NODE4:    NODE5;
NODE5:    NODE8,NODE9;
NODE6:    Cout;
NODE7:    NODE9;
NODE8:    NODE10;
NODE9:    NODE11;
NODE10:   NODE11;
NODE11:   S;
```

De input files bestaan uit twee secties. Als eerste worden alle *nodes* van het circuit benoemd als tweede alle *edges*. De twee secties worden gescheiden door een witregel.

Elke regel in de specificatie file bestaat uit slechts één aparte node of edge beschrijving. Alles achter een '#' teken tot aan het einde van de regel wordt gezien als commentaar. Spaties en/of andere white-space karakters zijn niet relevant. Regels worden afgesloten met een line feed en vervolgens een carriage return LF ev.CR zoals gebruikelijk bij ASCII-bestanden.

Node-beschrijving:
<node_identifier>:<node_descriptor >;

Hierin is *node_identifier* een string bestaande uit ten minste 1 karakter uit de verzameling {a-z,A-Z,_}.
Hierin is *node_descriptor* een element uit de verzameling { INPUT_HIGH, INPUT_LOW, PROBE, OR, AND, NOT, NAND, NOR, XOR}. Deze geeft dus aan welke logische bewerking op de node zal plaatsvinden.
Opmerking: De inputs INPUT_HIGH en INPUT_LOW representeren een 1 en een 0 bij de uitvoer. Dit zijn default waarden en mogen bij een interactief systeem natuurlijk aangepast worden in de userinterface voor de simulatie start.

Edge beschrijving:
<node_source_identifier>:<node_target_identifier, … ,node_target_identifier>;

Node_source_identifier beschrijft de bron vanuit waar een tak begint. De *node_source_identifier* moet in de node sectie gedefinieerd zijn.
Node_target_identifier beschrijft de nodes waar de bron tak naar wijst. De *node_target_identifier* moet in de node sectie gedefinieerd zijn.

Bijlage C: Logica functies van diverse basis componenten.

Waarheidstabellen voor diverse logische componenten. Merk op dat alle mogelijke poorten kunnen worden gemaakt van de *not*- en de *and* poort. (oftewel: de moeder van alle logische poorten is de *NAND* poort.) Kun je dit gebruiken in je object model?

De NOT poort of 'Inverter'

De uitgang van een *NOT* poort is logisch '1' wanneer de ingang logisch '0' is.

Symbool:

Vergelijking: $F = \overline{A}$

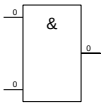
Waarheidstabel:

Input A	Output
0	1
1	0

De AND poort

De uitgang van een *AND* poort is logisch '1' wanneer beide ingangen logisch '1' zijn.

Symbool:



Vergelijking: $F = A \cdot B$

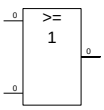
Waarheidstabel:

Input A	Input B	Output
0	0	0
0	1	0
1	0	0
1	1	1

De OR poort

De uitgang van een *OR* poort is logisch '1' wanneer een of meer ingangen logisch '1' zijn.

Symbool:



Vergelijking: $F = A + B$

Waarheidstabel:

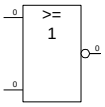
Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Stelling:
Met behulp van *AND* en *NOT* poorten kan ook een OR poort gemaakt worden. Met behulp de wetten van de propositielogica geldt: $F = A + B = \overline{\overline{A + B}} = \overline{\overline{A} \cdot \overline{B}}$. Merk op dat deze laatste term uit alleen *NOT* and *AND* functies bestaat waarmee de stelling bewezen is.

De NOR poort (Not OR)

De uitgang van een *NOR* poort is logisch '1' wanneer beide ingangen logisch '0' zijn.

Symbool:



Vergelijking: $F = \overline{A + B}$

Waarheidstabel:

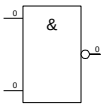
Input A	Input B	Output
0	0	1
0	1	0
1	0	0
1	1	0

Stelling:
Met behulp van *AND* en *NOT* poorten kan ook een *NOR* poort gemaakt worden. Met behulp de wetten van de propositie logica geldt: $F = \overline{A + B} = \overline{A} \cdot \overline{B}$. Merk op dat deze laatste term uit alleen *NOT* and *AND* functies bestaat waarmee de stelling bewezen is.

De NAND poort (Not AND)

De uitgang van een *NAND* poort is logisch '1' wanneer niet beide ingangen logisch '1' zijn.

Symbool:



Vergelijking: $F = \overline{A \cdot B}$

Waarheidstabel:

Input A	Input B	Output
0	0	1
0	1	1

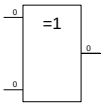
1	0	1
1	1	0

Stelling:
Met behulp van *AND* en *NOT* poorten kan ook een *NAND* poort gemaakt worden. Met behulp de wetten van de propositie logica geldt: $F = \overline{A \cdot B}$. Merk op dat deze laatste term uit alleen *NOT* and *AND* functies bestaat waarmee de stelling bewezen is.

De XOR poort (eXclusieve OR)

De uitgang van een *XOR* poort is logisch '1' wanneer beide ingangen verschillend zijn.

Symbool:



Vergelijking: $F = A \oplus B$

Waarheidstabel:

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Stelling:
Met behulp van *AND* en *NOT* poorten kan ook een *XOR* poort gemaakt worden. Met behulp de wetten van de propositie logica (zie ook de waarheidstabel) geldt:

$F = A \oplus B = \overline{A} \cdot B + A \cdot \overline{B} = \overline{\overline{A} \cdot B} \cdot \overline{A \cdot \overline{B}}$. Merk op dat deze laatste term uit alleen *NOT* and *AND* functies bestaat waarmee de stelling bewezen is.