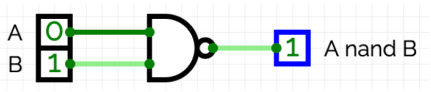
Lernziele

Praktischer Umgang mit CircuitVerse und Anwendung der in den Vorlesungsvideos gezeigten Konzepte.

Sekundäre Ziele:
- Erwerb eines Verständnisses für die mathematische Modellierung von Logik-Gattern sowie das Erkennen ihrer Grenzen.
- Experimentelle Einführung in das Konzept des Zweierkomplements.
- Betrachtung von sequentieller Logik mit Abgrenzung zur kombinatorischen Logik.

Schritt 1

Fülle zunächst die Wahrheitstabelle für ein NAND-Gatter aus.

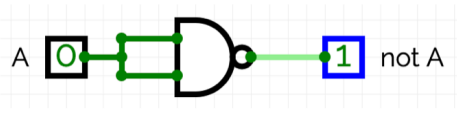


A	B	$A \barwedge B$
0	0	
0	1	
1	0	
1	1	

Hintergrund: Ein NAND-Gatter (Logisches Nicht-Und, Symbol: $\barwedge$) kann als Grundbaustein verwendet werden, um alle anderen Logik-Gatter zu konstruieren. Diese Eigenschaft ist nicht nur mathematisch von Bedeutung, sondern auch praktisch, da ein NAND-Gatter ausreichend ist, um eine Vielzahl von Logik-Funktionen zu realisieren.

NOT-Gatter: Überprüfe zunächst, ob die Wahrheitstabelle für $A \barwedge A$ korrekt ausgefüllt ist und der Ausdruck damit äquivalent zur logischen Negation von A ist. Realisiere dann den Ausdruck wie dargestellt in CircuitVerse.

A	$A \barwedge A$
0	1
1	0



Fülle für die folgenden Logik-Gatter jeweils die Wahrheitstabelle für den angegebenen Ausdruck aus. Realisiere anschließend den Ausdruck in CircuitVerse:

AND-Gatter mit 2 NAND-Gatter

A	B	$A \barwedge B$	$(A \barwedge B) \barwedge (A \barwedge B)$
0	0		
0	1		
1	0		
1	1		

OR-Gatter mit 3 NAND-Gatter

A	B	$A \barwedge A$	$B \barwedge B$	$(A \barwedge A) \barwedge (B \barwedge B)$
0	0			
0	1			
1	0			
1	1			

Schritt 2

Ein XOR (Symbol $\dot{\vee}$) kann man durch die Gleichung $A \mathbin{\dot{\vee}} B = \big(B \mathbin{\barwedge} (A \mathbin{\barwedge} A)\big) \mathbin{\barwedge} \big(A \mathbin{\barwedge} (B \mathbin{\barwedge} B)\big)$ darstellen. Bestätige das mit einer Wahrheitstabelle:

A	B	$A \barwedge A$	$B \barwedge (A \barwedge A)$	$B \barwedge B$	$A \barwedge (B \barwedge B)$	$\big(B \barwedge (A \barwedge A)\big) \barwedge \big(A \barwedge (B \barwedge B)\big)$

Zeige mit einer Wahrheitstabelle, dass auch $A \mathbin{\dot{\vee}} B = \big((A \barwedge B) \barwedge A\big) \barwedge \big((A \barwedge B) \barwedge B\big)$ gilt:

A	B	$A \barwedge B$	$(A \barwedge B) \barwedge A$	$(A \barwedge B) \barwedge B$	$\big((A \barwedge B) \barwedge A\big) \barwedge \big((A \barwedge B) \barwedge B\big)$

Aufgaben:
- Realisiere beide Ausdrücke für XOR in CircuitVerse.
- Erkläre einem Partner den praktischen Vorteil der zweiten Darstellung.
- Zusätzliche Herausforderung: Versuche, die erste Gleichung formal in die zweite Gleichung umzuformen. Du kannst dabei $A \barwedge B = 1 - A \cdot B$ verwenden, um die logische Gleichung in eine algebraische Gleichung umzuschreiben.

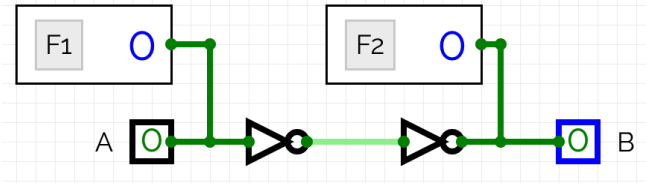
Schritt 3

Hintergrund: Die Beschreibung von Logik-Gattern mit Booleschen Funktionen ist eine mathematische Modellierung. Bei diesem Modell wird angenommen, dass ein Logik-Gatter instantan den Output liefert. In der Realität ist dies erst nach einer gewissen Zeit (im Nanosekundenbereich) der Fall. Ein genaueres Modell kann mit Hilfe von Differentialgleichungen erhalten werden, die beschreiben, wie sich analoge Werte (Spannungen und Stromstärken) in einer elektronischen Schaltung verändern. Einem analogen Wert wie der Spannung wird dann ein digitaler Wert zugeordnet:
- Ist die Spannung unterhalb einer Schranke U_0 wird der Zustand als 0 interpretiert.
- Ist die Spannung oberhalb einer Schranke U_1 wird der Zustand als 1 interpretiert.
- Liegt die Spannung im Bereich von U_0 bis U_1, ist der digitale Zustand undefiniert (man bezeichnet diesen in der Regel mit x). Bei einem Logik-Gatter ist nach einer gewissen Zeitspanne (dem "delay") der Zustand aber immer definiert, also 0 oder 1.

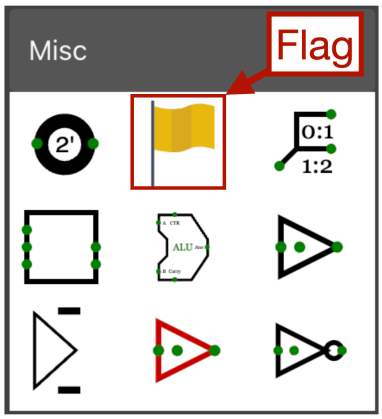
Für unsere Zwecke ist die Modellierung von Logik-Gattern mit Booleschen Funktionen in der Regel ausreichend. Es ist aber hilfreich, die Grenzen des Modells zu kennen. In CircuitVerse kann für diesen Zweck die Delays der Gatter in der Simulation berücksichtigen.

Anleitung

Baut in CircuitVerse die folgende Schaltung mit zwei NOT-Gattern:

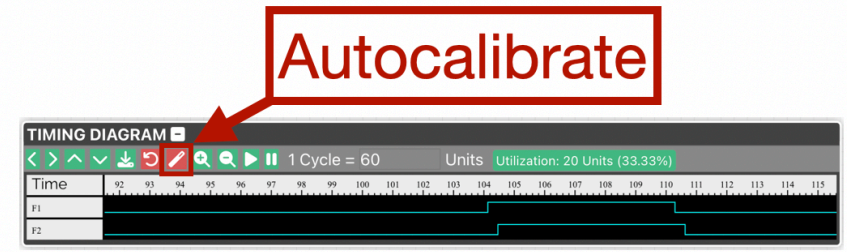


Die Symbole mit den Kennzeichnungen F1 und F2 sind dabei sogenannte Flags, die man unter "Misc" findet (in den "Properties" kann man die Bezeichnungen unter "Debug Flag Identifier" ändern):



Modelliert man die Logikschaltung mit Booleschen Funktionen, dann hat der Eingang A und der Ausgang B stets den gleichen Wert. Um in der Simulation zu sehen, dass es in der Realität aber einen Delay gibt, kann man in CircuitVerse das Timing-Diagramm betrachten:

- Klickt im Timing-Diagramm auf "Autocalibrate".



- Ändert den Wert von Eingang A von 0 auf 1 und dann wieder von 1 auf 0.

Im Timing-Diagramm wird zeitabhängig der von F1 und F2 abgegriffene Wert dargestellt. Per Voreinstellung hat jedes Logik-Gatter in CircuitVerse ein Delay von 10 Nanosekunden. Klickt man auf ein Logik-Gatter, dann kann man dies unter "Properties" verändern.

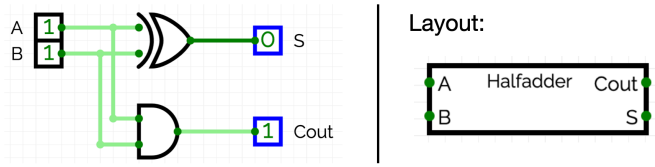
Schritt 4

Als nächstes soll die Auswirkung der Delays bei unserem 4-Bit Addierer beobachtet werden. Ein interessanter Fall tritt auf, wenn das Bitmuster 1111 mit dem Bitmuster 0001 addiert wird. Das korrekte Ergebnis ist dann 0000 (mit einem Cout-Wert von 1). Beobachtet man jedoch die Bits des Ergebnisses, nimmt dies zunächst den Wert 1110, dann 1100, dann 1000 und erst dann das endgültige Ergebnis 0000 an.

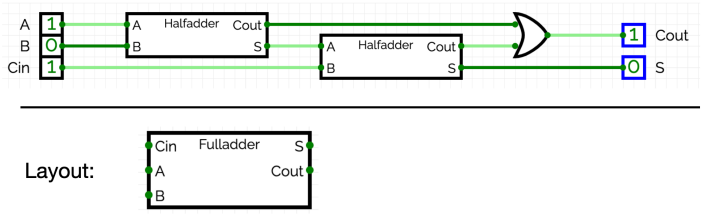
In der Praxis haben Delays Auswirkungen auf die maximale Taktrate eines Rechners. Abhängig von der "Länge der Logik-Pfade" muss man mehr oder weniger lang warten, bis ein Ergebnis für eine weitere Rechnung verwendet werden kann.

Um den beschriebenen Effekt zu beobachten, bauen wir den in der Vorlesung beschriebenen 4-Bit Addierer nach und verwenden dann Flags und das Timing-Diagramm.

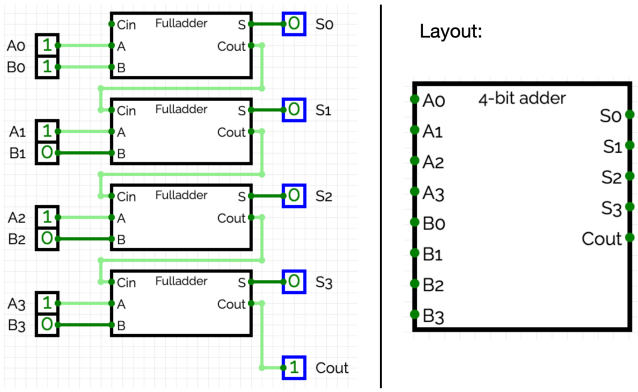
Halbaddierer



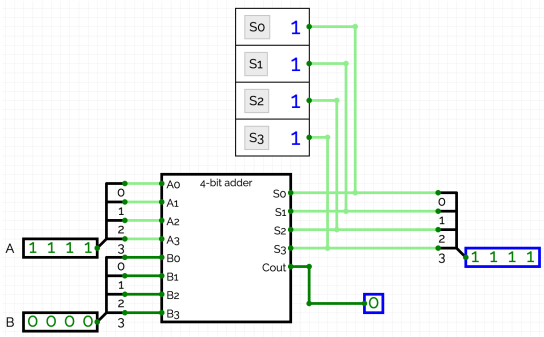
Volladdierer

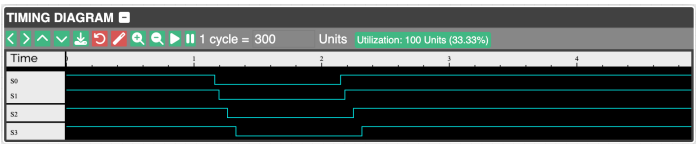


4-bit Addierer



Experiment



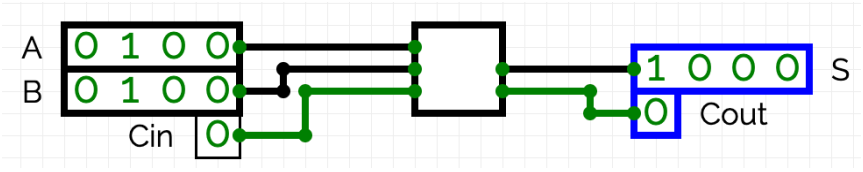


Unser 4-Bit Addierer ist ein sogenannter Carry-Ripple-Addierer. Es gibt auch Addierer, die effizientere Algorithmen verwenden, wie zum Beispiel der Carry-Look-Ahead-Addierer. Diese Addierer können das Ergebnis schneller berechnen, indem sie auf intelligente Weise die Carry-Bits vorberechnen. Die verschiedenen Algorithmen unterscheiden sich auch darin, wie viele Logik-Gatter bei ihrer Umsetzung benötigt werden. Es ist wichtig, die Vor- und Nachteile der verschiedenen Addierer-Algorithmen zu verstehen, um die passende Implementierung für die jeweilige Anwendung auszuwählen.

Schritt 5

Für unsere Zwecke wird es ausreichen, uns nur mit dem Carry-Ripple-Addierer zu beschäftigen. Interessant ist, dass dabei bei einem 4-Bit-Addierer vier Volladdierer verwendet werden. Es wäre jedoch ausreichend für die Addition, wenn man einen Halbaddierer und drei Volladdierer verwendet. Ein Vorteil ist, dass man so aus zwei 4-Bit-Addierern einen 8-Bit-Addierer bauen kann.

Aufgabe:
In CircuitVerse findet man unter "Misc" einen fertigen Addierer. Unter "Properties" kann man diesen als 4-Bit-Addierer konfigurieren.



Verwendet zwei 4-Bit-Addierer, um einen 8-Bit-Addierer zu bauen.

Schritt 6

Ein 4-Bit Addierer liefert in s nicht $a+b$ sondern $(a+b) \bmod 16$. Bezüglich dieser Rechenoperation ist $G = \{0,1\}^4$ eine Gruppe. Für jedes $a \in G$ gibt es also ein eindeutiges inverses Element $-a$ bezüglich der Addition, und dieses hat die Eigenschaft, dass $a + (-a) = 0$ gilt (wobei mit 0 ein Bitmuster bezeichnet wird, das nur aus Nullen besteht).

Die praktische Anwendung ist, dass eine Subtraktion auf eine Addition zurückgeführt werden kann. Dazu ist es nur notwendig , zu einem beliebigen Bitmuster a das zugehörige Bitmuster von $-a$ zu bestimmen. Eine Regel dafür kann man sich leicht herleiten:

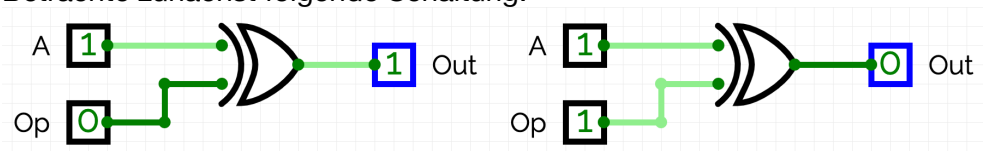
- Addiert man auf das Bitmuster 1111 das Bitmuster 0001, erhält man in s das Bitmuster 0000.
- Addiert man auf das Bitmuster (zum Beispiel 1010) das invertierte Bitmuster (in dem Fall 0101), dann erhält man in s stets 1111. Wenn man nun zusätzlich Cin auf 1 setzt, erhält man in s stets 0000.

Aufgabe:
- Realisiere in CircuitVerse eine Logikschaltung für die Subtraktion. Verwende dazu ein 4-Bit breites NOT-Gatter, um das Bitmuster von b zu invertieren. Die Subtraktion wird (analog zur Addition) dann natürlich nicht $a-b$, sondern $(a-b) \bmod 16$ berechnen.
- Ändere die Schaltung so, dass Cout mit 0 angibt, dass $a - b = (a-b) \bmod 16$ gilt. Ansonsten soll Cout den Wert 1 haben.

Schritt 7

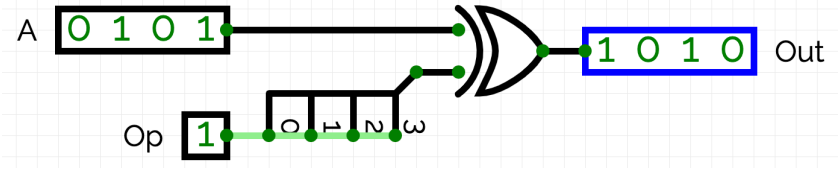
Im nächsten Schritt soll die Schaltung erweitert werden, um zwischen Addition und Subtraktion wählen zu können.

Betrachte zunächst folgende Schaltung:



Mit dem Op-Eingang kann ausgewählt werden, ob A invertiert wird. Mit Wahrheitstabellen kann leicht gezeigt werden, dass folgendes gilt: Im Fall Op $= 0$ ist Out $= A$, und im Fall Op $= 1$ ist Out $= \neg A$.

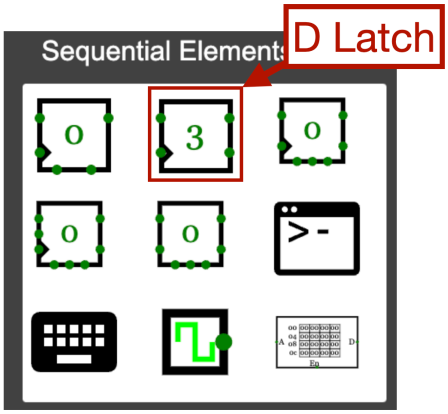
In dieser Schaltung wird ein 4-Bit breites XOR-Gatter verwendet, um das Bitmuster zu invertieren:



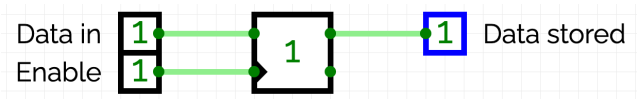
Verwende diese Methode zur selektiven Inversion eines Bitmusters, um zwischen Addition und Subtraktion zu wählen.

Schritt 8

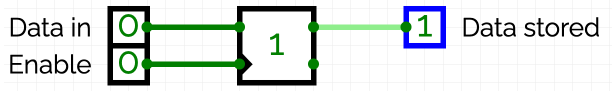
Um einen Rechner zu bauen, benötigen wir Bauteile (wir werden sogenannte D-Flipflops verwenden), mit denen man Ergebnisse speichern kann, um diese dann für weitere Rechnungen verwenden zu können. Um einen Einblick in diese Thematik zu bekommen, verwenden wir zunächst das sogenannte D-Latch. Dies findet man in CircuitVerse unter Sequential Elements.



Die Funktionsweise können wir mit folgender Schaltung experimentell herleiten:



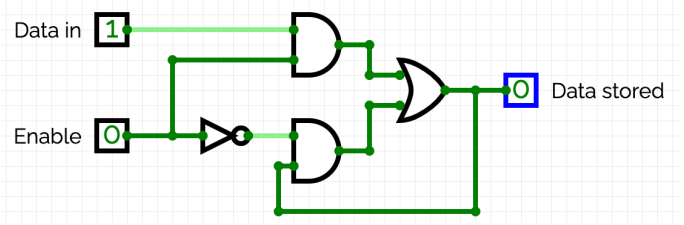
Hierbei solltet ihr folgende Beobachtung machen:
- Ist Enable auf 1, dann wird in "Data stored" der Wert von "Data in" übernommen.
- Ist Enable auf 0, dann bleibt der Wert von "Data stored" unverändert, egal welchen Wert "Data in" annimmt.



Mathematisch kann man dies mit

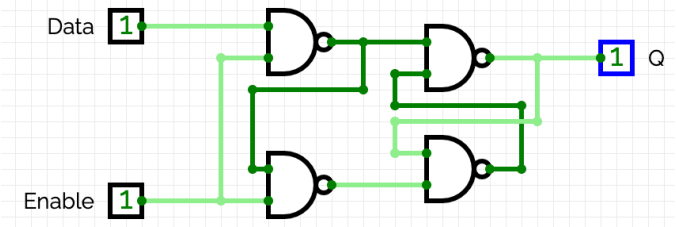
$$\text{Data}_{\text{stored}} = \text{Data}_{\text{stored}} \wedge \neg\text{Enable} \vee \text{Data}_{\text{in}} \wedge \text{Enable}$$

charakterisieren. Bezüglich $\text{Data}_{\text{stored}}$ kann man diese implizite Gleichung nicht auflösen. Deshalb kann ein D-Latch nicht mit einer Booleschen Funktion beschrieben werden. Dennoch können wir versuchen, diese Gleichung in gewohnter Weise wie folgt mit Logik-Gattern umzusetzen:



Aufgaben:

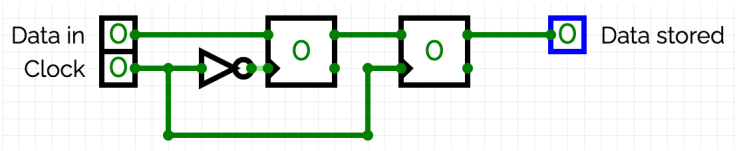
- Setze die Schaltung in CircuitVerse um. Bestätige, dass sich die Schaltung scheinbar wie das D-Latch verhält.
- Ändere das Delay des NOT-Gatters auf 20. Die Schaltung sollte sich jetzt nicht mehr wie ein D-Latch verhalten.
- Obige Gleichung charakterisiert zwar ein D-Latch, aber es gibt keine Anleitung dafür, wie man es realisieren kann. Für eine stabile Realisierung müssen die Delays der Gatter berücksichtigt werden. Wir werden darauf in dieser Veranstaltung nicht näher eingehen. Aber eine mögliche Realisierung soll euch nicht vorenthalten werden:



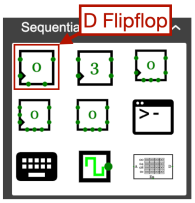
Bemerkung:
Es gilt die Faustregel:
- Was im Simulator nicht funktioniert, das funktioniert auch in der Realität nicht.
- Was im Simulator funktioniert, kann in der Realität funktionieren, muss aber nicht. Denn eine Simulation ist letztlich ein Modell, das die Realität vereinfacht wiedergibt.

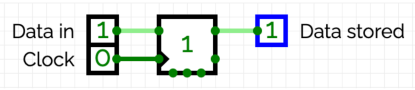
Schritt 9

Bei einem D-Latch sagt man, dass es (durch Enable) "pegelgesteuert" ist. Wir werden jedoch zum Speichern von Daten ein D-Flipflop verwenden, das "flankengesteuert" ist. Um eine Vorstellung für den Unterschied zwischen pegelgesteuert und flankengesteuert zu bekommen, realisieren wir folgende Schaltung mit zwei D-Latches und einem NOT-Gatter:



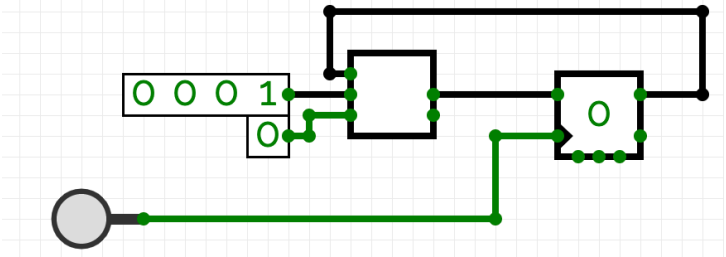
Experimentiere mit dieser Schaltung und erkläre dann deinem Nachbarn (oder uns), wieso man sagt, dass "Data in" bei einer positiven Flanke von Clock gespeichert wird. Verwendet anschließend ein D-Flipflop und bestätigt, dass sich dies genauso verhält:





Schritt 10

Wir werden später D-Flipflops in unserem Rechner für sogenannte CPU-Register verwenden. Zunächst soll jedoch folgendes Beispiel zeigen, wie man die kombinatorische Logik eines Addierers mit der sequentiellen Logik des D-Flipflops zusammenspielen kann. Baut dazu folgende Schaltung mit einem 4-Bit Addierer und einem 4-Bit breiten D-Flipflop:



Beim Experimentieren solltet ihr beobachten, dass der im Flipflop gespeicherte Wert durch Drücken des Buttons erhöht wird. Daraus kann man erahnen, dass die maximale Taktfrequenz von der Länge des längsten Logikpfads abhängt. Wenn man den Delay des Addierers genauer modellieren würde und den Button zu schnell drücken würde, dann würde das "Hochzählen" nicht funktionieren.

Macht euch auch klar, wieso in dieser Schaltung ein D-Flipflop notwendig ist und mit einem D-Latch nicht funktionieren würde. Ersetzt dazu das D-Flipflop mit einem D-Latch und testet dann die Schaltung.