

## 7 Programmierbare Logik

### 7.1 Rückblick und Vorausschau

Ich habe schon Erfahrungen mit diversen Anbietern von programmierbarer Logik gemacht. Ursprünglich habe ich ja Logik-ICs (ASICs) designt, damals gab es, abgesehen von kleinen PALs und GALs, noch gar keine Programmierbare Logik. Meistens konnten wir die Eingabe mit einem handelsüblichen Schema-Editor oder mittels speziellen Editoren der IC-Hersteller vornehmen. Funktionsblöcke gab es auch schon (z.B. ganze Nachbildungen von CMOS- oder TTL-Bausteinen), wir haben aber sehr viele Funktionen von Grund auf neu aufbauen müssen. Zum Testen der ICs mussten wir ein Testpattern schreiben, welches auf dem Testgerät Eingänge geändert hat und die dazu passenden Ausgangszustände testen konnte. Jedes kleinste Element (Logiktor: NOR, NAND) musste mindestens einmal einen Einfluss auf einen Ausgang haben. Hatten wir zum Beispiel einen 31-Bit Zähler in der Schaltung hätten wir den Clock  $2^{31}$  ( $2'147'483'648$ ) mal toggeln lassen müssen, bis sich am Ausgang etwas änderte. War dem Zähler noch ein weiterer, oder mehrere Zähler nachgeschaltet, wäre der Zeitaufwand für den Test viel zu gross geworden. Deshalb haben wir alle Zähler selber aufgebaut. Und zwar haben wir Flip-Flips so beschaltet, dass man sie sowohl als Zähler als auch als Schieberegister benutzen konnte. Alle diese «Schiebe-Zähler» haben wir dann «in Serie» geschaltet und konnten so ein Testpattern in alle Zähler «schieben» (z.B.: '1111 1111 1111 1111 1111 1111 1111 1100'), dann wieder umschalten auf die Zählerfunktion, ein paar Clocks an den Eingang bringen und der Ausgang hat geschaltet. Dieses, sogenannte «Boundary Scan Design» ist sehr ähnlich der JTAG-Schnittstelle.

Das war für ASICs notwendig, in die kann man nicht hineinschauen. CPLDs oder FPGAs sind aber ganz anders aufgebaut in die kann man reinschauen (Bei Altera mittels LAI - Logic Analyzer Interface), deshalb ist der «Boundary Scan Design» hier nicht notwendig. Es werden aber 8 Datenleitungen für die LAI-Funktion benötigt. Im Endeffekt bedeutet dies, dass man 4 Datenleitungen für die JTAG-Schnittstelle und 8 für die LAI-Schnittstelle spendieren muss. Spätestens wenn wir mit «embedded» Mikrokontrollern arbeiten, werden wir die LAI-Schnittstelle benutzen.

Komplexe Funktionen haben wir mit «state machines» realisiert, d.h. Flip-Flops, deren Ausgänge wir über ein Geflecht von logischen Verküpfungen rückgekoppelt hatten. Mit der Zeit sind wir aber dazu übergegangen komplexe Funktionen mittels VHDL zu programmieren und als Funktionsblöcke in die Schemas einzufügen. Später haben wir dann die ganzen Schaltungen in VHDL programmiert.

Unsere Faustregel damals lautete:

- Eingabe der Funktion (Schema oder VHDL):  
**30 ... 40 % der Entwicklungszeit**
- Erzeugen der Testpattern:  
**60 ... 70 % der Entwicklungszeit**

Unsere zweite Faustregel damals lautete:

- Überlege Dir genau was Du machst (Funktion: Pflichtenheft → Umsetzung)
- Vermeide unter allen Umständen Fehler
- Weil:
  - Die Herstellung der Prototypen (Wafer und IC) kostet zwischen 50'000 € und 200'000 €

Wir waren sehr erleichtert, als die ersten programmierbaren Logikbausteine auf den Markt kamen. Damit hatten wir eine Methode, um die Funktionen von kleineren Schaltungen schon vor der Umsetzung auf einen Wafer zu testen. Funktionierte die Schaltung nicht wie gewollt, konnten wir noch einmal über die Bücher, eine bessere Version erzeugen und einen neuen Chip «brennen». Die von uns benutzten Bausteine arbeiteten mit der sogenannten «Antifuse»-Technologie. Beim Programmieren wurden einfach Sicherungen durchgebrannt. Da diese nicht rekonfiguriert werden konnten, konnte ein Chip immer nur einmal «gebrannt» werden. Heutige Chips kann man praktischerweise immer wieder neu programmieren.

Je nachdem wie viele Geräte hergestellt werden sollen, kann man direkt mit programmierbaren Bausteinen arbeiten. Einen Wafer herstellen zu lassen lohnt sich nur bei sehr grossen Stückzahlen und wenn die Funktion nachträglich nicht mehr geändert werden muss.

Viele Hersteller bieten sogar 8-Bit- und 32-Bit-Cores in VHDL an. Es gibt auch Cores von kleinen PIC-Kontrollern (z.B. den Ur-PIC16C54), welche in einem CPLD oder einem FPGA viel schneller getaktet werden können als die Originalbausteine. Es bietet sich auch die Möglichkeit, den «Embedded Mikrokontrollern», interne Logik vor- oder nachzuschalten.

## 7.2 Intel-Bausteine (Ehemals Altera)

Als erstes arbeiten wir hier mit Bausteinen der Firma Altera. Beispiele mit Bausteinen anderer Hersteller werden wir in Band 3 behandeln!

Die Firma Altera wurde 1983 gegründet und 2015 von Intel aufgekauft. Ich habe «Quartus Prime Lite Edition» installiert, eine Einführung von ca. 10 Minuten auf Youtube geschaut und danach innerhalb kürzester Zeit selber einen Halb-Addierer auf einem Cyclone IV DemoBoard und auf einem MAX II DemoBoard programmieren und testen. Ein weiteres kurzes Youtube-Video und ich konnte in kürzester Zeit den Grundtakt von 66 MHz meines MAX II Demo-Boards auf einen Takt von 1 Hz runterteilen und damit eine LED blinken lassen.

## 7.3 Beispiel: Halbaddierer

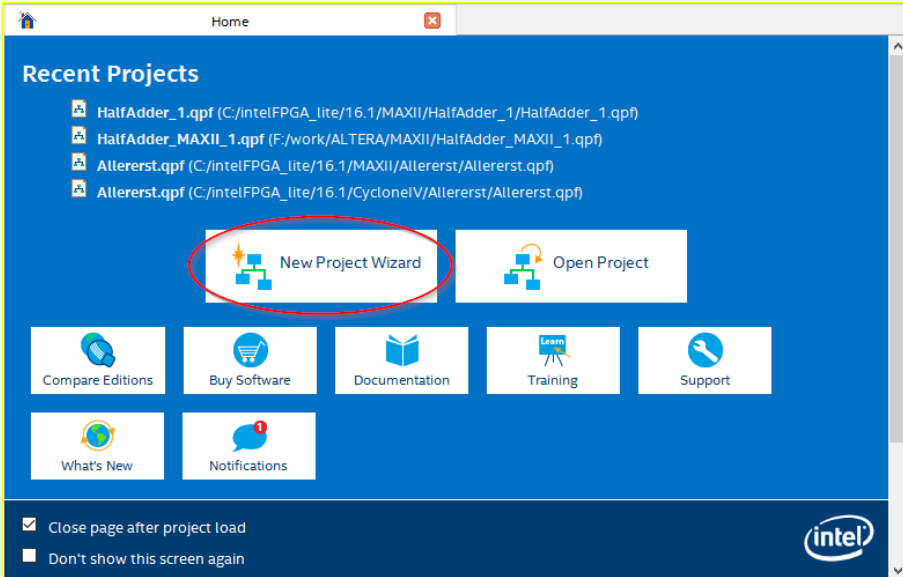
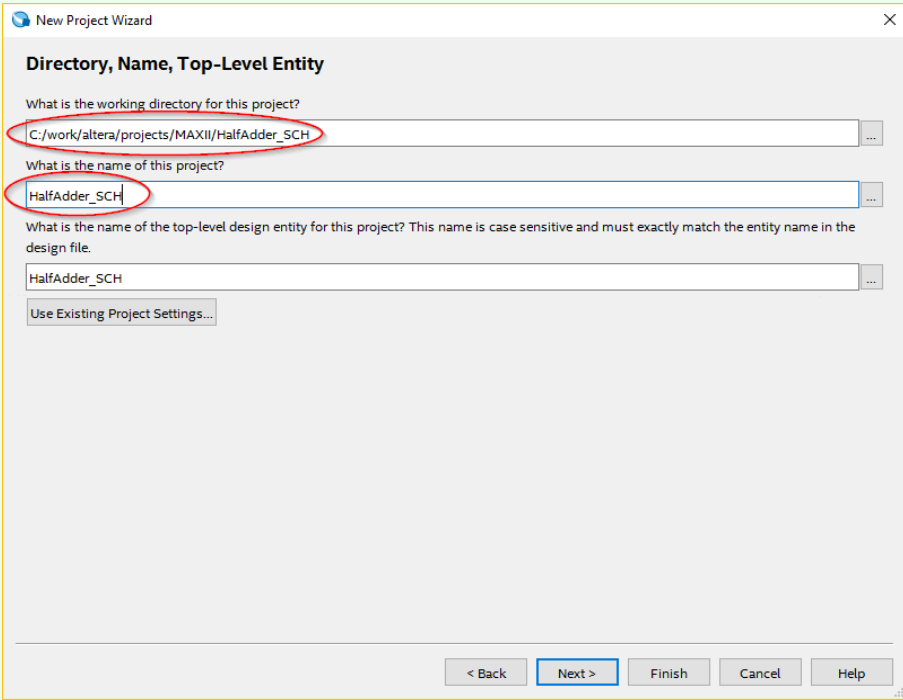
Im vorangehenden Kapitel haben wir ja schon diverse Varianten von Halb-Addierern auf kleinsten Mikrokontrollern programmiert. Die Theorie muss ich daher nicht mehr erklären und kann direkt mit der Realisierung beginnen.

Es gibt 2 grundsätzlich verschiedene Ansätze programmierbarer Logik zu programmieren:

1. Eingabe mittels Schema (also Logikbausteine oder Funktionsblöcke)
2. Eingabe mittels «Hochsprache» (VHDL, Verilog usw.)

### 7.3.1 Habaddierer mit Schemaeingabe

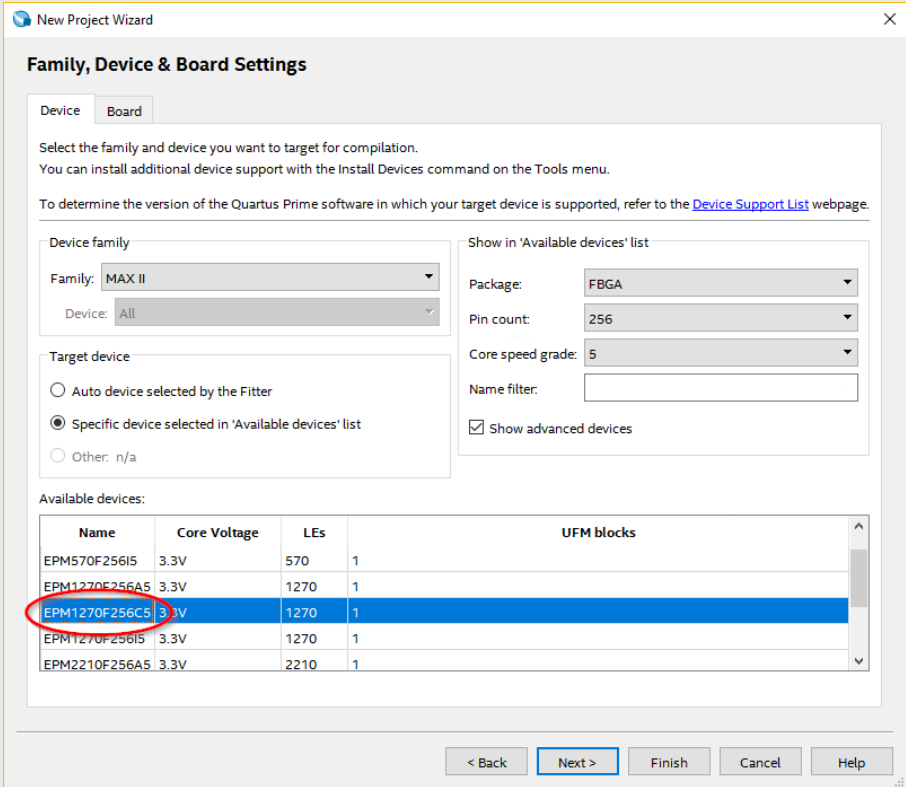
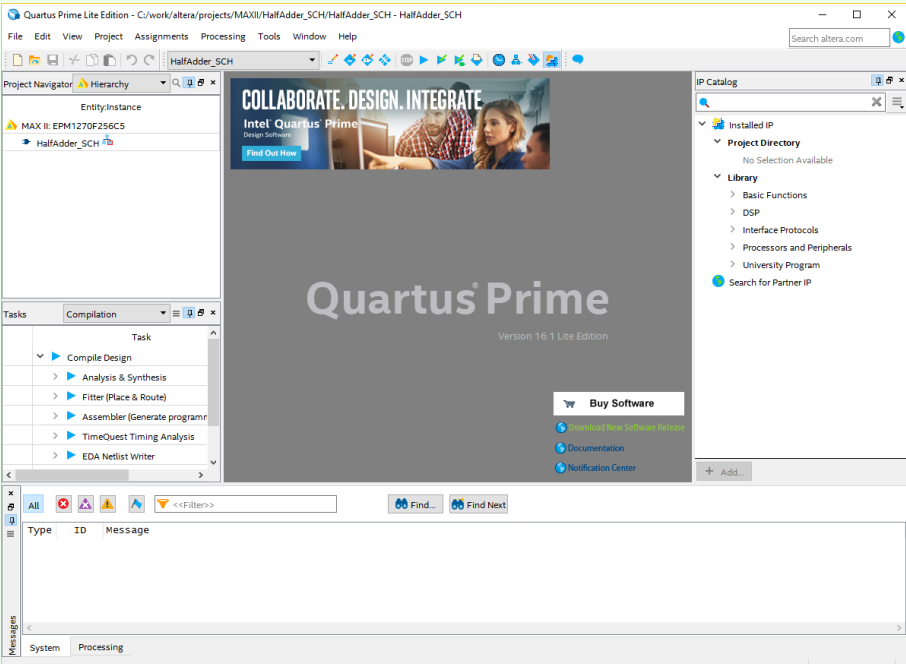
Wie schon erwähnt gibt es sehr gute Youtube-Anleitungen für den Umgang mit der Programmiersoftware «Quartus Prime Lite Edition» (Momentan bevorzuge ich die Serie von Professor Bill Kleitz). Es lohnt sich daher nicht jeden Schritt auf Papier aufzuzeigen, wenn ein kurzes Video die Vorgehensweise besser vermitteln kann und die Software bei jedem neuen Update leicht anders zu bedienen ist. Trotzdem hier mal ein «Schnelldurchlauf» (mit Quartus Prime Version 16.1.0 Build 196 10/24/2016 SJ Lite Edition):

Arbeits-schritt	Beschreibung
<p>1</p>	<ul style="list-style-type: none"> <li>• Also erst einmal starten wir Quartus Prime Lite Edition»</li> <li>• Dann wählen wir «New Project Wizzard»</li> </ul> 
<p>2</p>	<ul style="list-style-type: none"> <li>• Bei der Einführung (Introduction) klicken wir einfach auf «NEXT»</li> </ul>
<p>3</p>	<ul style="list-style-type: none"> <li>• Bei «Directory, Name, Top-Level-Entry wählen wir einen Ordner für die Ablage des Projektes</li> <li>• Ich wähle immer einen Ordner aus, welcher sich nicht im Programmordner, aber auf demselben Laufwerk befindet. Ich habe die Projekte immer im Ordner «work»</li> <li>• Dann wählen wir noch einen passenden Namen für das Projekt; hier: HalfAdder_SCH</li> </ul> 



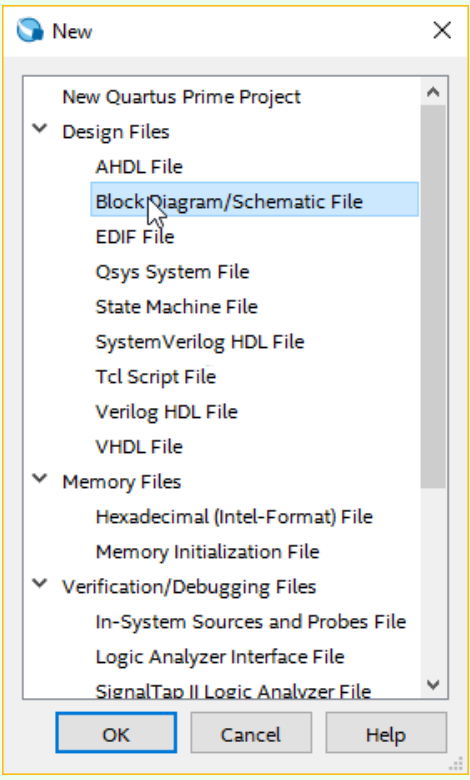
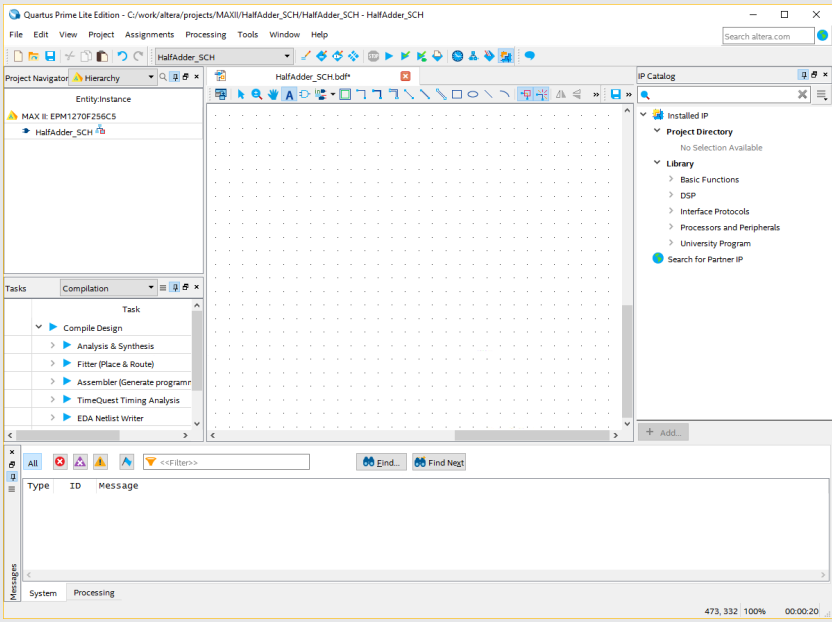
Nicht zu lange Pfade benutzen. Mit Ausnahme des «\_» keine Sonderzeichen benutzen. Den Leerschlag « » zähle ich auch zu den Sonderzeichen.

7

Arbeits-schritt	Beschreibung
4	<ul style="list-style-type: none"> <li>Im Fenster «Project Type» wählen wir «Empty project» und klicken auf «NEXT»</li> </ul>
5	<ul style="list-style-type: none"> <li>Im Fenster «Add Files» klicken wir direkt auf «NEXT»</li> </ul>
6	<ul style="list-style-type: none"> <li>Auswahl des benutzten Typen</li> <li>In unserem Beispiel: Cyclone IV - EPM1270F256C5 (kann man vom Chip ablesen)</li> </ul> 
7	<ul style="list-style-type: none"> <li>Im Fenster «EDA Tool Settings» verändern wir nichts und klicken auf «Next»</li> </ul>
8	<ul style="list-style-type: none"> <li>Im Fenster «Summary» nichts verändern und «Finish» klicken</li> </ul>
9	<ul style="list-style-type: none"> <li>Das Projekt steht. Wir sind bereit für die Eingabe der Funktion</li> </ul> 

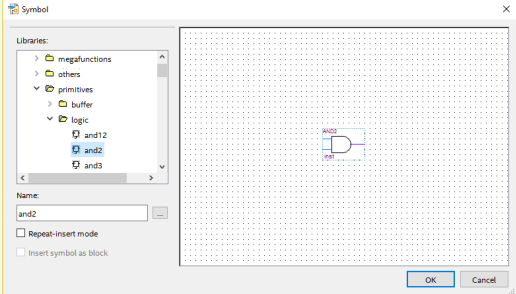
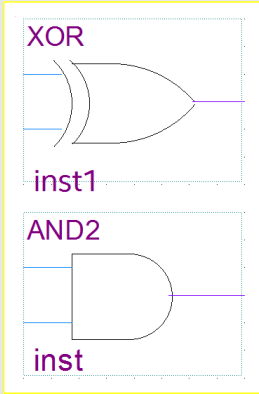
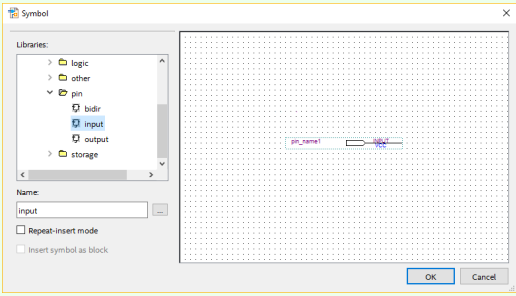
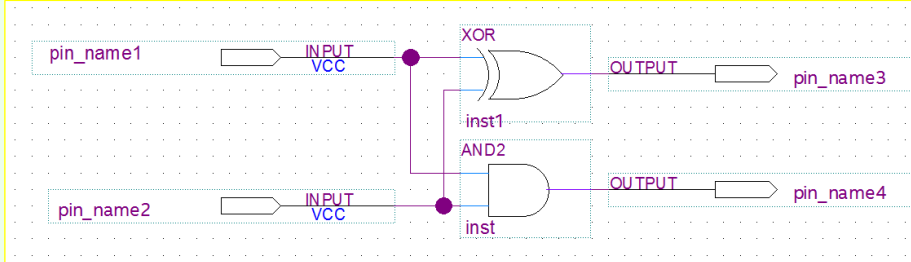


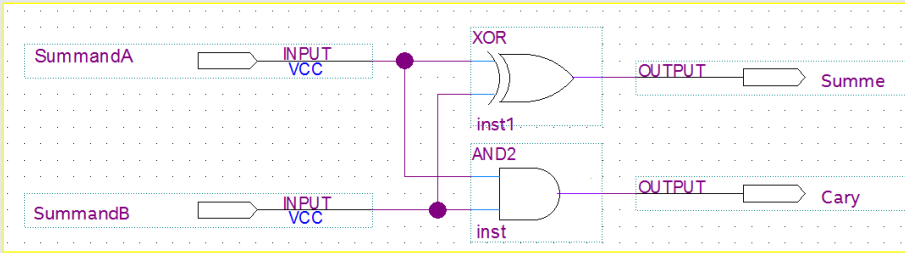

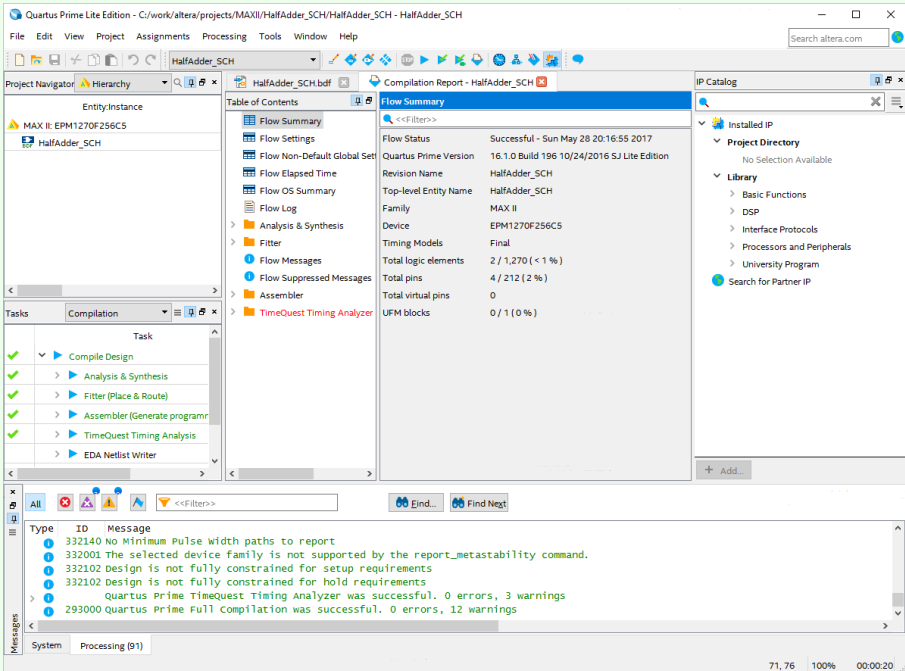
Eingabe des Schemas

Arbeits-schritt	Beschreibung
<p><b>1</b></p>	<p><b>Neues Schemablatt öffnen:</b></p> <ul style="list-style-type: none"> <li>• «File»</li> <li>• «New»</li> <li>• Wählen «BlockDiagram/Schematic File»</li> <li>• Auf «OK» klicken</li> </ul> 
<p><b>2</b></p>	<p><b>Dem Schema einen Filenamen geben:</b></p> <ul style="list-style-type: none"> <li>• Im Menü wählen:             <ul style="list-style-type: none"> <li>■ «File»</li> <li>■ «Save as»</li> </ul> </li> <li>• Eingabe des Filenamens: hier «HalfAdder_SCH»</li> </ul> 



Mit Ausnahme des «\_» keine Sonderzeichen benutzen. Den Leerschlag « » zähle ich auch zu den Sonderzeichen.

Arbeits-schritt	Beschreibung
<p><b>3</b></p>	<p><b>Suchen und platzieren des AND-Gates:</b></p> <ul style="list-style-type: none"> <li>• In gerasterter Fläche «Klick auf die rechte Maustaste»                     <ul style="list-style-type: none"> <li>• «Insert»</li> <li>• «symbol»</li> <li>• «primitives»</li> </ul> </li> <li>• in «logic» das AND «and2» anwählen, auf OK klicken und das AND platzieren</li> </ul> 
<p><b>4</b></p>	<p><b>Suchen und platzieren des XOR-Gates:</b></p> <ul style="list-style-type: none"> <li>• Den obigen Vorgang wiederholen um ein XOR auszuwählen und zu platzieren</li> </ul> 
<p><b>5</b></p>	<p><b>Eingänge platzieren:</b></p> <ul style="list-style-type: none"> <li>• In gerasterter Fläche, Klick auf die rechte Maustaste</li> <li>• «Insert»</li> <li>• «Symbol»</li> <li>• «primitives»</li> <li>• in «pin» «input» anwählen und auf OK klicken und das Eingangssymbol platzieren</li> <li>• noch ein zweites Eingangssymbol platzieren</li> </ul> 
<p><b>6</b></p>	<p><b>Ausgänge platzieren:</b></p> <ul style="list-style-type: none"> <li>• Den Vorgang wiederholen um 2 Ausgangssymbole zu platzieren</li> </ul>
<p><b>7</b></p>	<p><b>Alle Symbole mit Linien verbinden</b></p> 

Arbeits-schritt	Beschreibung
<b>8</b>	<p><b>Namen der Ein-/Ausgänge ändern</b></p> <ul style="list-style-type: none"> <li>• Doppelklick auf Symbol</li> </ul> 
<b>9</b>	<p><b>Schaltung ein erstes Mal kompilieren</b></p> <ul style="list-style-type: none"> <li>• Klick auf Dreiecksymbol der Menüleiste</li> </ul>  <ul style="list-style-type: none"> <li>• Keine Fehler «Quartus Prime TimeQuest Timing Analyzer was successful. 0 errors, 3 warnings» ist zwar rot markiert aber das ignorieren wir vorläufig noch</li> </ul> 




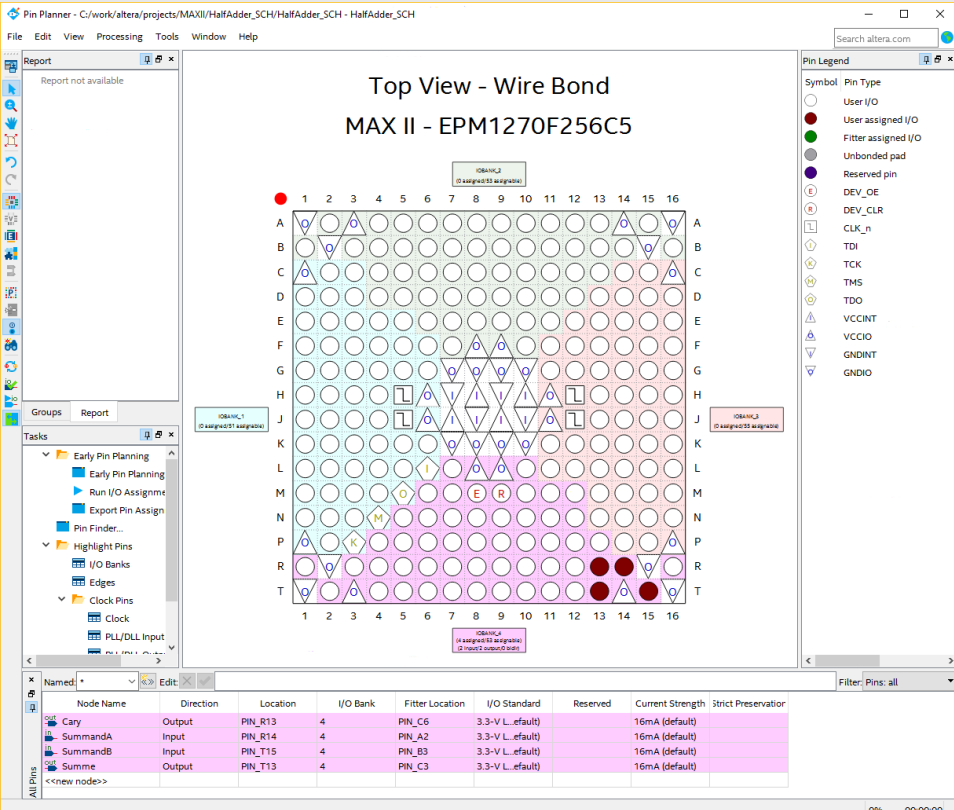


Niemals Minus-symbol (-) oder Bindestrich (-) benutzen!  
 Spätestens das Simulieren wird nicht funktionieren (Auch bei nachträglichem Ändern!  
 Von Anfang an Underscore oder «nichts» (keine Sonderzeichen) benutzen.

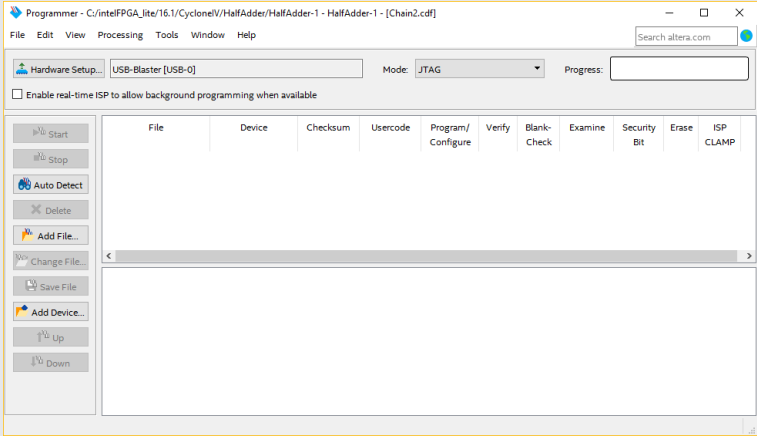
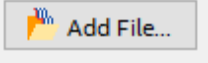
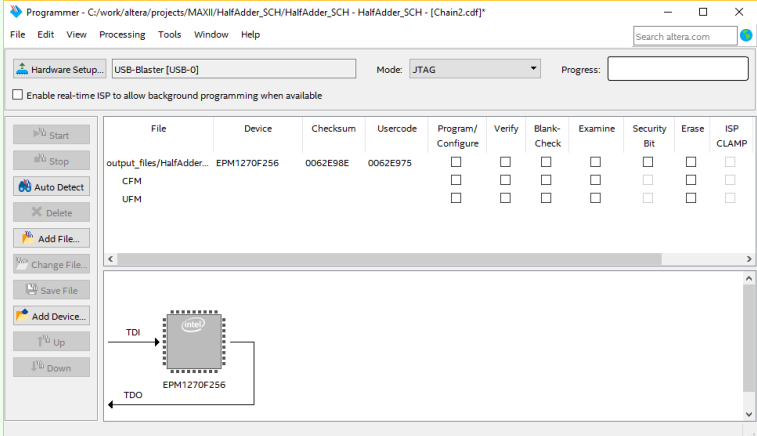
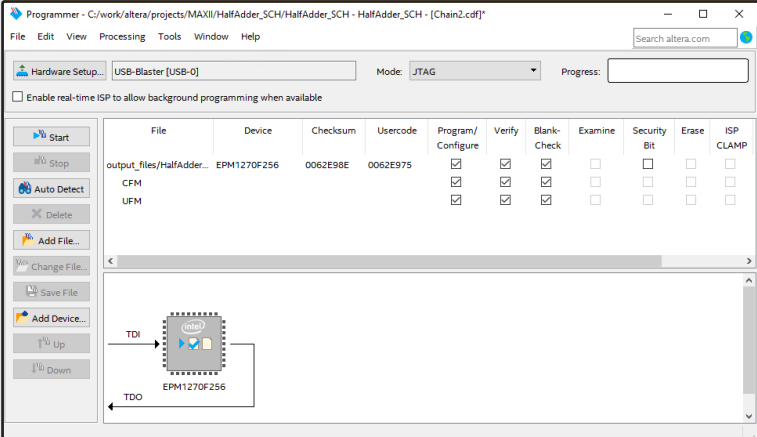
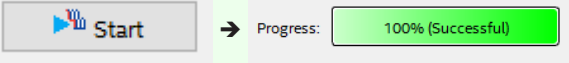
7

**Ressourcen**  
 Logic elements:  
 2/1270  
 Total pins:  
 4/212

Vom Schema zur Hardware

7

Arbeits-schritt	Beschreibung																																																						
<p>1</p>	<p><b>Zuweisung der Ein-Ausgänge zum Chip</b></p> <ul style="list-style-type: none"> <li>• Wenn beim Kompilieren keine Fehler aufgetreten sind, Klick auf folgendes Symbol</li> </ul>  <ul style="list-style-type: none"> <li>• Ein neues Fenster öffnet sich</li> </ul> 																																																						
<p>2</p>	<ul style="list-style-type: none"> <li>• Hier können die Eingänge und die Ausgänge den Ein-/Ausgängen des Chip zugeordnet werden. Wir entnehmen die passenden I/Os aus dem Benutzerhandbuch des Demo-Boards. Im Schema des Demo-Boards sehen wir, dass die Taster mit Pull-Up-Widerständen auf 3,3 Volt hochgezogen werden. Die LEDs sind auch an 3.3 V angeschlossen.</li> </ul> <table border="1" data-bbox="598 1467 1300 1590"> <thead> <tr> <th>lat.</th> <th>From</th> <th>To</th> <th>Assignment Name</th> <th>Value</th> <th>Enabled</th> <th>Entity</th> <th>Comment</th> <th>Tag</th> </tr> </thead> <tbody> <tr> <td>1</td> <td></td> <td>Carry</td> <td>Location</td> <td>PIN_127</td> <td>Yes</td> <td></td> <td></td> <td></td> </tr> <tr> <td>2</td> <td></td> <td>Summe</td> <td>Location</td> <td>PIN_126</td> <td>Yes</td> <td></td> <td></td> <td></td> </tr> <tr> <td>3</td> <td></td> <td>Summand-A</td> <td>Location</td> <td>PIN_10</td> <td>Yes</td> <td></td> <td></td> <td></td> </tr> <tr> <td>4</td> <td></td> <td>Summand-B</td> <td>Location</td> <td>PIN_30</td> <td>Yes</td> <td></td> <td></td> <td></td> </tr> <tr> <td>5</td> <td>&lt;&lt;new&gt;&gt;</td> <td>&lt;&lt;new&gt;&gt;</td> <td>&lt;&lt;new&gt;&gt;</td> <td></td> <td></td> <td></td> <td></td> <td></td> </tr> </tbody> </table> <ul style="list-style-type: none"> <li>• Danach «Close», die Zuweisungen bleiben gespeichert:</li> </ul>	lat.	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag	1		Carry	Location	PIN_127	Yes				2		Summe	Location	PIN_126	Yes				3		Summand-A	Location	PIN_10	Yes				4		Summand-B	Location	PIN_30	Yes				5	<<new>>	<<new>>	<<new>>					
lat.	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag																																															
1		Carry	Location	PIN_127	Yes																																																		
2		Summe	Location	PIN_126	Yes																																																		
3		Summand-A	Location	PIN_10	Yes																																																		
4		Summand-B	Location	PIN_30	Yes																																																		
5	<<new>>	<<new>>	<<new>>																																																				
<p>3</p>	<ul style="list-style-type: none"> <li>• Jetzt noch einmal definitiv kompilieren (Menüleiste oben)</li> </ul> 																																																						
<p>4</p>	<ul style="list-style-type: none"> <li>• Speisespannung des Demo-Boards anschliessen</li> <li>• «USB Blaster» anschliessen und im Menü auf «Programmer» klicken</li> </ul> 																																																						

Arbeits-schritt	Beschreibung
<b>5</b>	<ul style="list-style-type: none"> <li>• Ein neues Fenster erscheint</li> <li>• Wenn der Programmierer (Hier USB-Blaster [USB-0]) nicht erkannt wird, muss er über Hardware-Setup gesucht werden</li> </ul>  <ul style="list-style-type: none"> <li>• Auf «Add File» klicken:</li> </ul>  <ul style="list-style-type: none"> <li>• xxx.pof-File (Hier HalfAdder-SCH.pof) suchen und anklicken</li> </ul>
<b>6</b>	<ul style="list-style-type: none"> <li>• Jetzt wird der Chip angezeigt und oben erscheint eine Zeile mit den Daten des Chips (Checksumme usw.)</li> </ul> 
<b>7</b>	<ul style="list-style-type: none"> <li>• Checkboxes anklicken: «Programm/Configure, Verify, Blank-Check»</li> </ul> 
<b>8</b>	<ul style="list-style-type: none"> <li>• Druch anklicken von «Start» wird der Chip programmiert:</li> </ul>  <ul style="list-style-type: none"> <li>• Fertig → Testen:</li> </ul>

**Resumé**

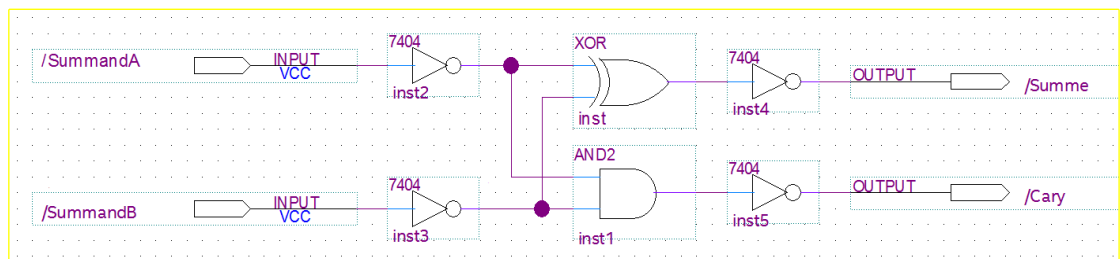
- Die Eingänge des Demo-Boards werden über Pull-Up-Widerstände «hochgezogen», d.h. Ruhezustand = «high», erst durch drücken der Taste wird ein «low» an den Eingang gelegt!  
D.H. wir haben hier negative Logik!
- Auch die LEDs sind mit Widerständen hochgezogen und leuchten nur wenn der Ausgang auf «low» geschaltet wird. Auch hier haben wir negative Logik!
- Mit etwas Denkarbeit können wir trotzdem die Funktion testen:

**Wahrheitstabelle Halb-Addierer mit negativer Logik**

	SummandB	SummandA	Carry	Summe
Taster/LED	S2	S3	LED1	LED2
Eingang/Ausgang	PIN_T15	PIN_R14	PIN_R13	PIN_T13
WT-Zeile_1	gedrückt = „0“	gedrückt = „0“	leuchtet = „0“	leuchtet = „0“
WT-Zeile_2	gedrückt = „0“	nicht gedrückt = „1“	leuchtet = „0“	1 dunkel = „1“
WT-Zeile_3	nicht gedrückt = „1“	gedrückt = „0“	leuchtet = „0“	1 dunkel = „1“
WT-Zeile_4	nicht gedrückt = „1“	nicht gedrückt = „1“	dunkel = „1“	leuchtet = „0“

Wir haben aber bereits jetzt die Fähigkeit unser Schema der Hardware (Demo-Boards) anzupassen. D.h. wir fügen einfach 2 Inverter vor die Eingänge und 2 Inverter nach den Ausgängen ins Schema ein. Dabei testen wir gleich ob das Sonderzeichen «/» von Quartus II akzeptiert wird.

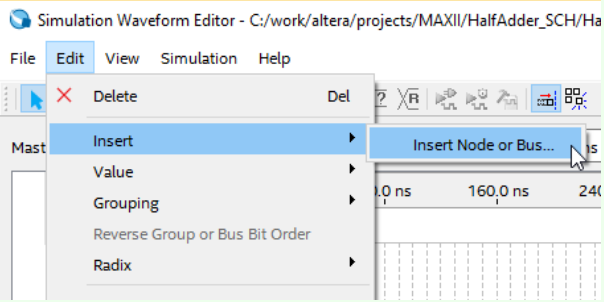
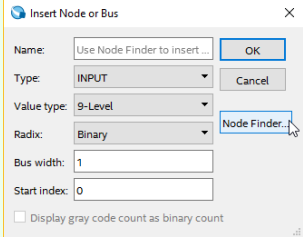
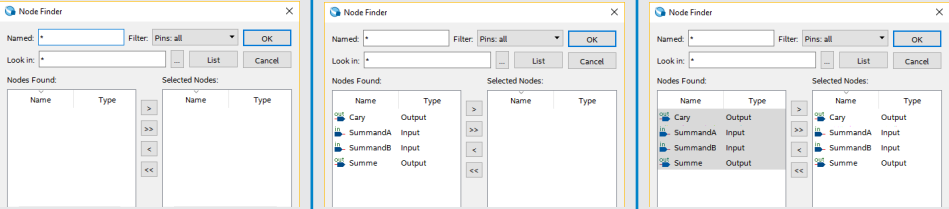
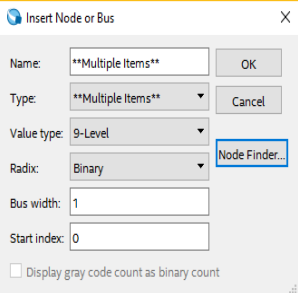
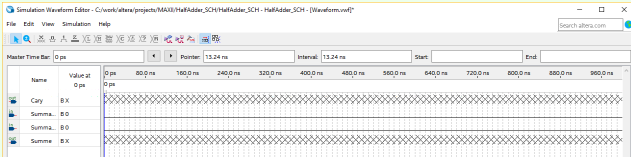
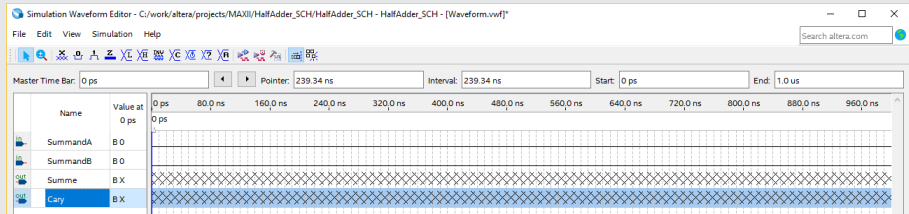
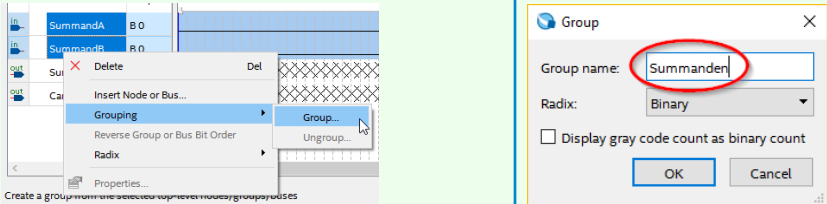
Hier das neue Schema:



Funktioniert einwandfrei!

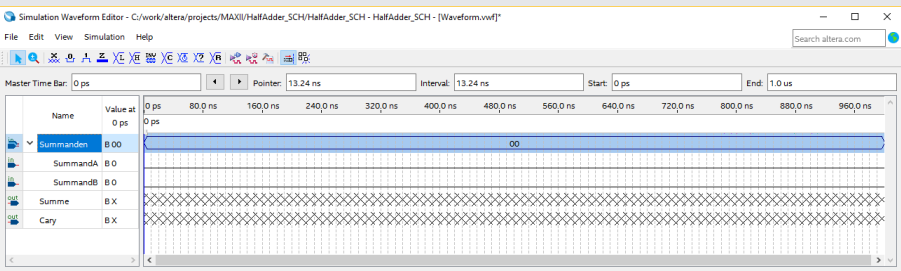
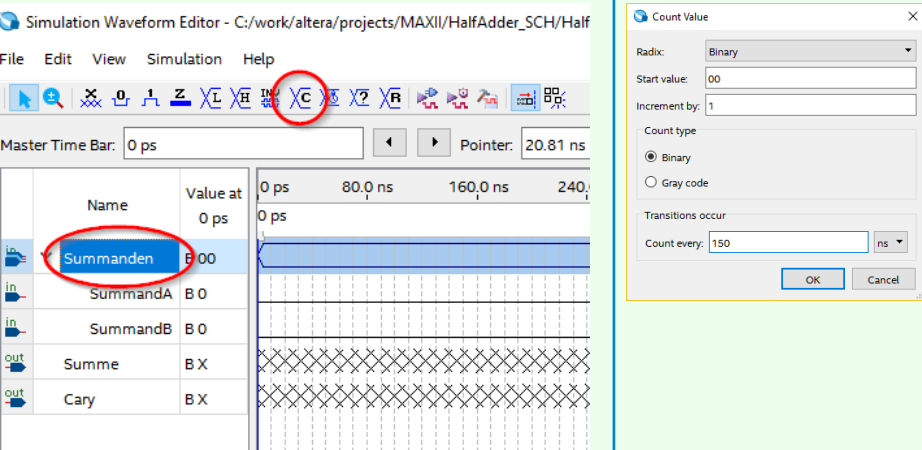
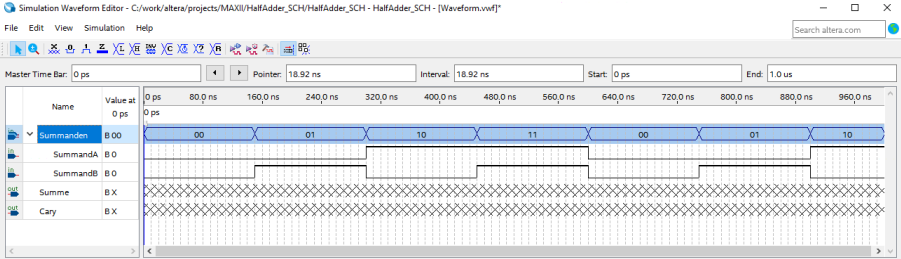
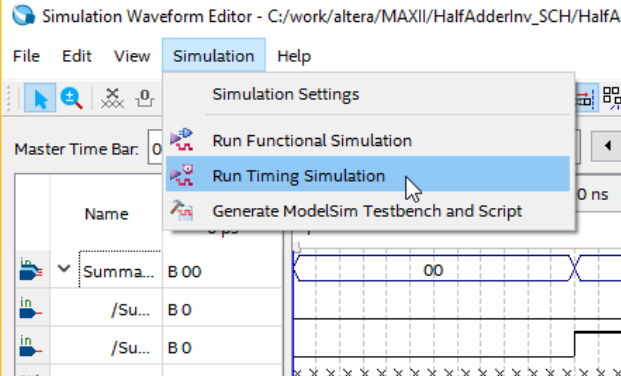
**Simulation der Schaltung**

Arbeits-schritt	Beschreibung
1	<ul style="list-style-type: none"> <li>Im Hauptmenü anwählen:                             <ul style="list-style-type: none"> <li>«File»</li> <li>«New»</li> <li>«University Program VWF»</li> </ul> </li> <li>Klick auf «OK»</li> <li>Ein neues Fenster erscheint</li> </ul>

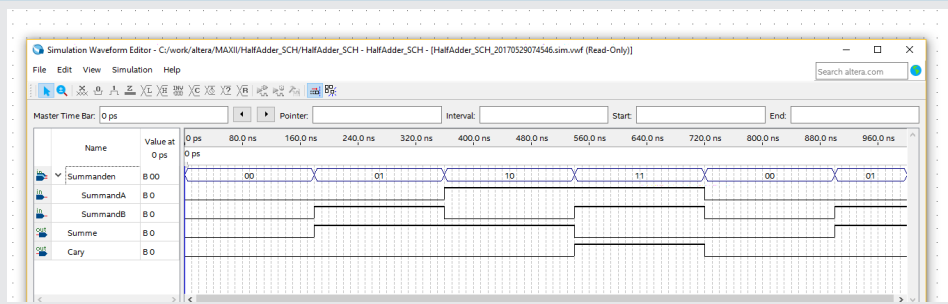
Arbeits-schritt	Beschreibung	
2	<p>• Die zu simulierenden I/Os suchen mit</p>   <p>«Node Finder» anklicken</p>	
3	 <p>«List» anklicken</p> <p>«&gt;&gt;» anklicken</p> <p>«OK» anklicken</p>	
4	<p>• Jetzt erscheinen die Signale im «Simulationsfenster»</p>   <p>«OK» anklicken</p>	
5	<p>• Jetzt sortieren wir die I/Os nach unseren Wünschen</p> <ul style="list-style-type: none"> <li>■ Signal «Carry» anklicken und nach unten ziehen</li> </ul> 	
6	<p>Die zwei Eingänge können wir zu einer Gruppe «Summanden» zusammenfassen. Dadurch können wir das Testpat-tern einfacher eingeben:</p>  <p>SummandA und SummandB anwählen und rechte Maustaste anklicken</p> <p>Dem Bus einen Namen geben: hier «Summand»</p>	

7



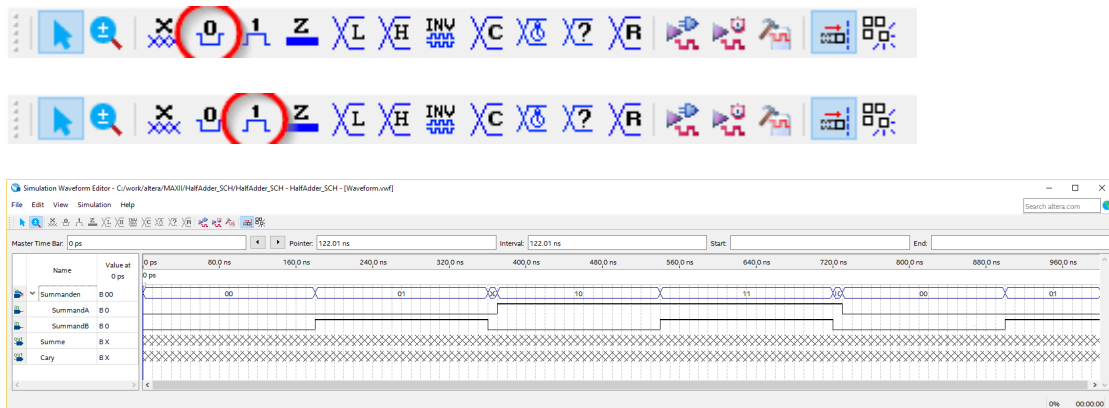
Arbeits-schritt	Beschreibung
7	<p>• Jetzt sind wir bereit für die Eingabe des Testpatterns</p> 
8	<p>• «Summanden» wählen und das obige Symbol mit dem «C» anklicken</p> <p>Eine sinnvolle Zeit wählen:</p> 
9	<p>• Das Testpattern ist jetzt bereit</p> 
10	<p>Die Simulation wird gestartet indem man aus dem Menu «Simulation» wählt und auf «Run Timing Simulation» klickt</p> 

7

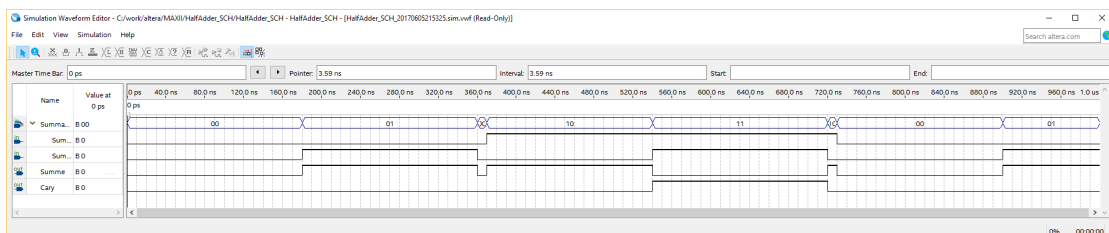
Arbeits-schritt	Beschreibung
11	 <p><b>Die Simulation entspricht der Wahrheitstabelle:</b></p> <ul style="list-style-type: none"> <li>■ beide Eingänge «low» → kein Ausgang «high»</li> <li>■ je ein Eingang «high» → nur «Summe» = «high»</li> <li>■ beide Eingänge «high» → nur «Carry» = «high»</li> </ul>

### Simulation der Schaltung mit Glitches

Bisher haben wir die Laufzeiten der Logik nicht beachtet. Jede Logikzelle erzeugt aber eine Verzögerung des Eingangssignals zum Ausgangssignal. Die Programmierbare Logik ist aber so schnell, dass die Laufzeiten einzelner Logik-Gatter im Simulator nicht ersichtlich sind. Erst bei grossen Schaltungen werden wir die Laufzeitverzögerungen sehen. Wir erzeugen aber jetzt mal eine Verzögerung «manuell» indem wir das «High-Signal» von Eingang «SummandA» von 360 bis 370 ns markieren und in der Funktionsleiste oben auf **0** klicken und dann von 720 bis 730 ns markieren und in der Funktionsleiste oben auf **1** klicken.



Diese Situation kann auftreten, wenn ein Eingangssignal direkt anliegt und das andere noch mehrere Logikzellen durchlaufen muss. Hier haben wir mal eine Laufzeitverzögerung von 10 ns gewählt.

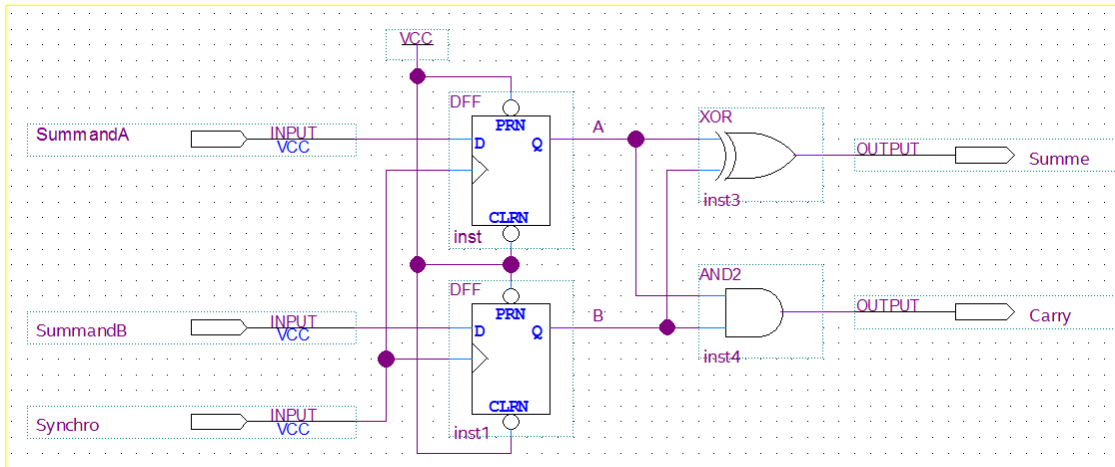


### Synchronisation der Eingänge

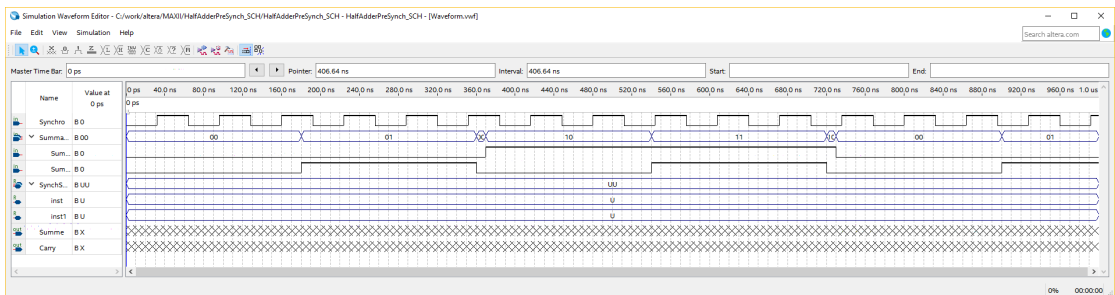
Die Eingänge unserer Schaltung können wir synchronisieren, so dass beide zur gleichen Zeit den Zustand wechseln. Dies erreichen wir mit dem Vorschalten von D-Flip-Flops in die Eingänge unseres Halb-Addierers.

Ressourcen  
Logic elements:  
2/1270  
Total pins:  
5/212

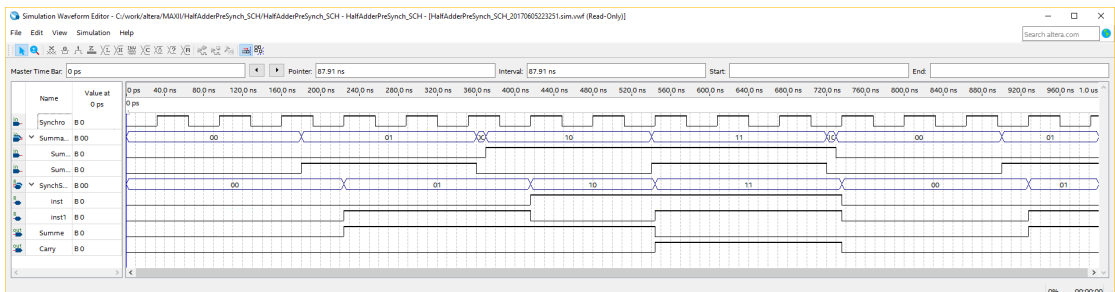
7



In der Simulation fügen wir jetzt noch den Clock-Eingang (Synchro) hinzu (Oberste Zeile). Zeile 2 zeigt die Eingänge als Bus, die Zeilen 3 und 4 als Einzelsignale. Zeile 5 zeigt die synchronisierten Eingänge als Bus und die Zeilen 6 und 7 als Einzelsignale.



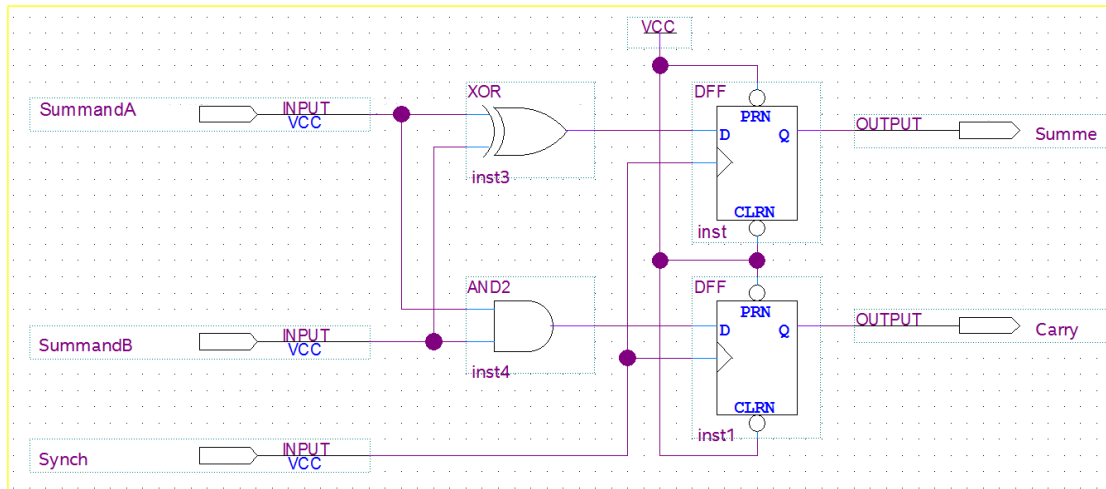
Die Glitches erscheinen jetzt nicht mehr. Je nach Zeitpunkt der steigenden Flanken des Synchronisationssignals können sie aber trotzdem noch auftreten.



Wir simulieren hier aber alles im ns-Bereich. Mit der Lite Version von Quartus II kann maximal bis zu einer Endzeit (Set End Time) von 100 µs simuliert werden. Für unser Beispiel mit dem Halb-Addierer und den Tastern kann man aber einen Synchronisationstakt von >100 ms benutzen. Synchronisation der Eingänge

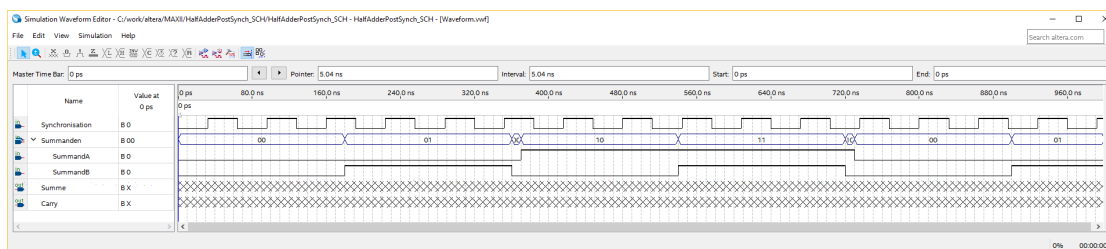
### Synchronisation der Ausgänge

Es können aber auch die Ausgänge synchronisiert werden. Dies erreichen wir mit dem Nachschalten von D-Flip-Flops in die Ausgänge unseres Halb-Addierers.

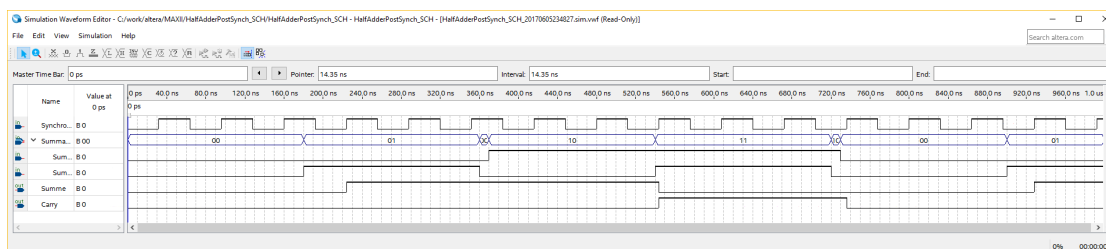


Ressourcen  
 Logic elements:  
 2/1270  
 Total pins:  
 5/212

In der Simulation fügen wir jetzt noch den Clock-Eingang (Synchro) hinzu (Oberste Zeile). Zeile 2 zeigt die Eingänge als Bus, die Zeilen 3 und 4 als Einzelsignale.



Die Glitches erscheinen jetzt nicht mehr. Je nach Zeitpunkt der steigenden Flanken des Synchronisationssignals können sie mit dieser Schaltung aber trotzdem noch auftreten.



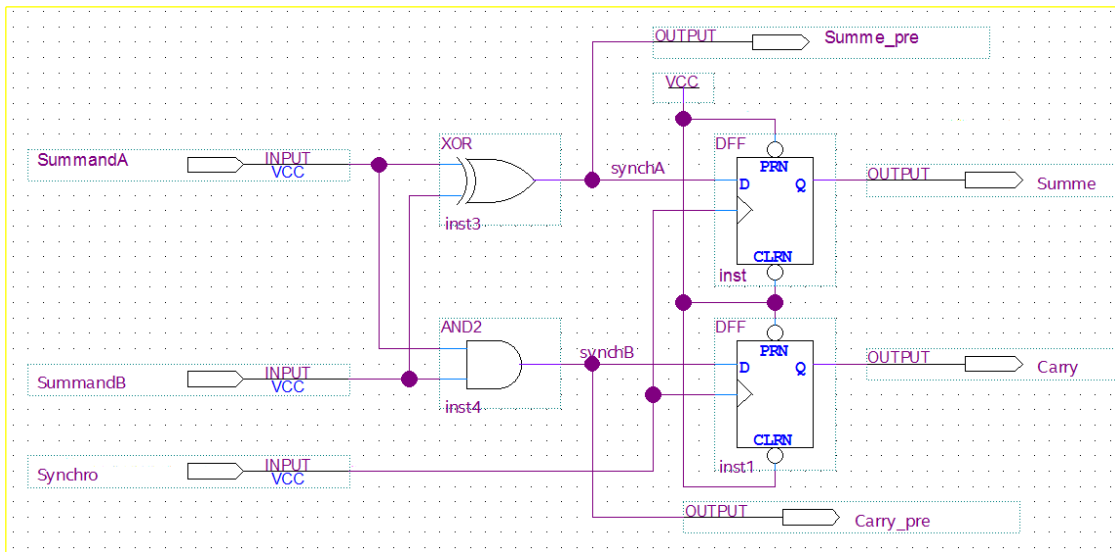
Wir wollen aber in die Schaltung hineinschauen, schaffen es aber nicht die Signale «synchA» und «synchB» anzuzeigen. Ich habe vergebens nach Testpunkten gesucht. Da wir aber lösungsorientiert sind, suche wir nicht lange und überlegen uns eine Alternative. Wir Setzen einfach zwei weitere Ausgänge in die Schaltung, verbinden sie mit «synchA» und «synchB» und simulieren erneut.

### Synchronisation der Ausgänge mit «Testpunkten»

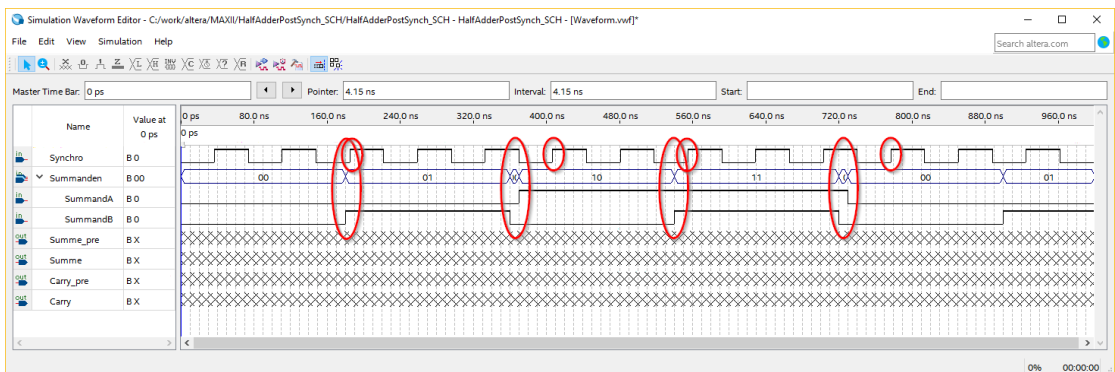
Die Testpunkt nennen wir «Summe\_pre» und «Carry\_pre».

Ressourcen  
 Logic elements:  
 2/1270  
 Total pins:  
 7/212

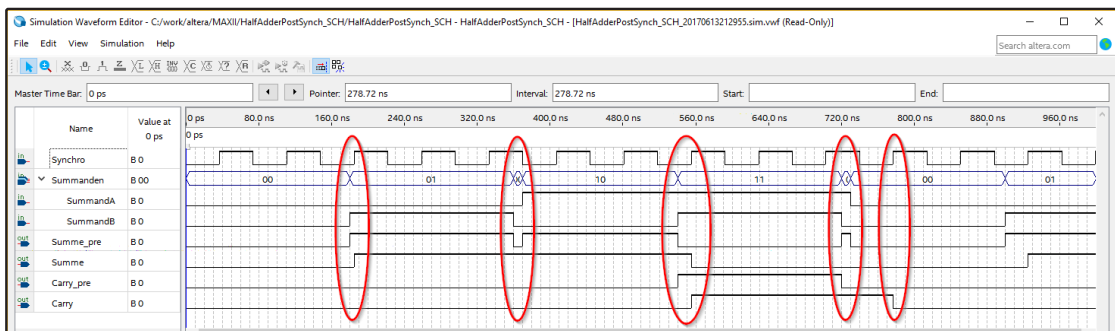
7



In der Simulation fügen die neuen Signale vor den Ausgängen «Summe» und «Carry» hinzu. So dass wir die Zustänge vor den Flip-Flops und nach den Flips-Flops untereinander haben und sie so besser vergleichen können.



Jetzt sehen wir, wie auf dem Signal «Summe\_pre» Glitches sind, welche aber nach dem Flip-Flop «weggefiltert» sind.



## 7.2.2 Halbdierer mit VHDL

Der grösste Vorteil von VHDL ist die Kompatibilität von einem Hersteller zum Anderen. Bei der Schemaeingabe, muss man das Schema beim Wechsel des Herstellers neu eingeben und riskiert dabei, dass es spezielle Funktionen beim neuen Hersteller nicht gibt und «nachgebaut» werden müssen.

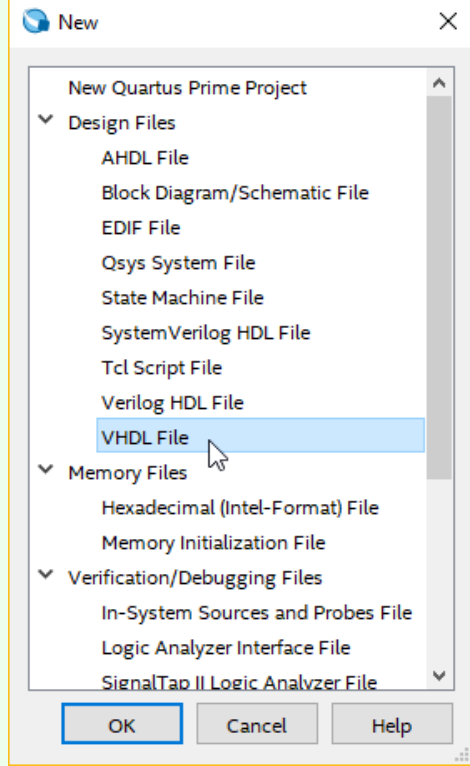
Hier der verkürzte Ablauf der Eingabe:

### Erzeugen des Projekts:

Arbeits-schritt	Beschreibung
1	<ul style="list-style-type: none"> <li>• Also erst einmal starten wir Quartus Prime Lite Edition»</li> <li>• Dann wählen wir «New Project Wizzard»</li> </ul> <p style="text-align: center;">Bild: siehe Seite 7.3</p>
2	<ul style="list-style-type: none"> <li>• Bei der Einführung (Introduction) klicken wir einfach auf «NEXT»</li> </ul>
3	<ul style="list-style-type: none"> <li>• Bei «Directory, Name, Top-Level-Entry wählen wir einen Ordner für die Ablage des Projektes</li> <li>• Ich wähle immer einen Ordner aus, welcher sich nicht im Programmordner, aber auf demselben Laufwerk befindet. Ich habe die Projekte immer im Ordner «work»</li> <li>• Dann wählen wir noch einen passenden Namen für das Projekt; hier: HalfAdder_VHDL</li> </ul> <p style="text-align: center;">Bild: analog zu Seite 7.3</p>
4	<ul style="list-style-type: none"> <li>• Im Fenster «Project Type» wählen wir «Empty project» und klicken auf «NEXT»</li> </ul>
5	<ul style="list-style-type: none"> <li>• Im Fenster «Add Files» klicken wir direkt auf «NEXT»</li> </ul>
6	<ul style="list-style-type: none"> <li>• Auswahl des benutzten Typen</li> <li>• In unserem Beispiel: Cyclone IV - EPM1270F256C5 (kann man vom Chip ablesen)</li> </ul> <p style="text-align: center;">Bild: analog zu Seite 7.4</p>
7	<ul style="list-style-type: none"> <li>• Im Fenster «EDA Tool Settings» verändern wir nichts und klicken auf «Next»</li> </ul>
8	<ul style="list-style-type: none"> <li>• Im Fenster «Summary» nichts verändern und «Finish» klicken</li> </ul>
9	<ul style="list-style-type: none"> <li>• Das Projekt steht. Wir sind bereit für die Eingabe der Funktion</li> </ul> <p style="text-align: center;">Bild: analog zu Seite 7.4</p>

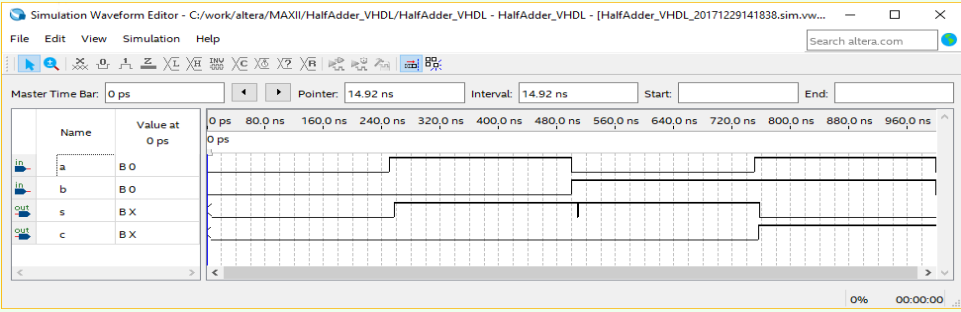
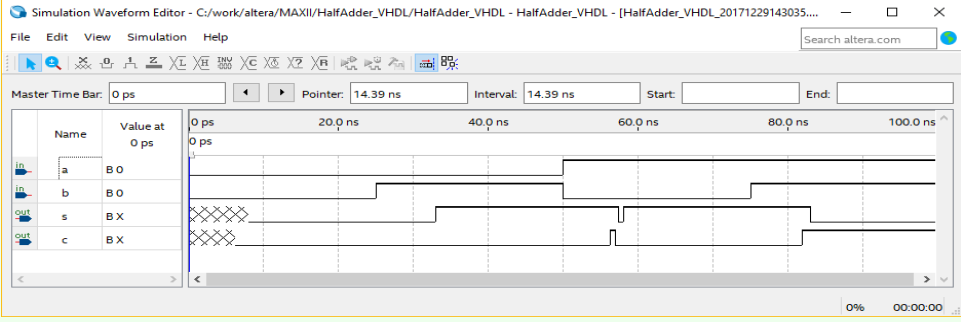
Eingabe des Schemas:

7

Arbeits-schritt	Beschreibung
1	<p><b>Neues VHDL-Blatt öffnen:</b></p> <ul style="list-style-type: none"> <li>• «File»</li> <li>• «New»</li> <li>• Wählen «VHDL File»</li> <li>• Auf «OK» klicken</li> </ul> 
2	<p><b>Dem VHDL einen Filenamens geben:</b></p> <ul style="list-style-type: none"> <li>• Im Menü wählen: <ul style="list-style-type: none"> <li>▪ «File»</li> <li>▪ «Save as»</li> </ul> </li> <li>• Eingabe des Filenamens: hier «HalfAdder_VHDL»</li> </ul> <p style="text-align: right;">Bild: siehe Seite 7.5</p>
3	<p><b>Jetzt geben wir im Editor den VHDL-Test für einen Halbaddierer ein:</b></p> <pre data-bbox="470 1310 1428 1848"> Library IEEE; use IEEE.STD_LOGIC_1164.ALL;  -- Half Adder with VHDL -- Martin Habenicht -- 2017-12-28  entity HalfAdder_VHDL is     port ( a,b : in STD_LOGIC;           s,c : out STD_LOGIC); end HalfAdder_VHDL;  architecture Behavioral of HalfAdder_VHDL is  begin     s &lt;= a xor b;     c &lt;= a and b; end Behavioral;                     </pre>
4	<p><b>Schaltung kompilieren</b></p> <ul style="list-style-type: none"> <li>• Klick auf Dreiecksymbol der Menüleiste</li> <li>• Keine Fehler</li> </ul>

Ressourcen  
Logic elements:  
2/1270  
Total pins:  
4/212



Arbeits-schritt	Beschreibung
<b>5</b>	<p><b>Simulation der VHDL-Schaltung:</b></p> <ul style="list-style-type: none"> <li>Vorgehen wie auf Seite 7-10 bis 7-13 beschrieben:</li> </ul>  <ul style="list-style-type: none"> <li>a = Summand 1</li> <li>b = Summand 2</li> <li>s = Summe</li> <li>c = Carry</li> </ul> <p>Die Funktion ist erfüllt. Man sieht aber deutlich, dass hier Schaltverzögerungen simuliert werden. Die Verzögerungen weisen einen Wert von 7 ns auf und bewirken beim gleichzeitigen Schalten (bei 757 ns) der Eingänge einen «Glitch».</p> <p>Wenn wir die Simulation 10 mal schneller ablaufen lassen (von 1000 ns auf 100 ns), sehen wir, dass es sich um je einen zeitversetzten Glitch mit einer Dauer von ca. 0.7 ns auf den Ausgängen handelt.</p> 

Da wir unsere Schaltungen im kleinen Millisekundenbereich betreiben, kümmern wir uns momentan nicht weiter um die Problematik der Laufzeitverzögerungen.

### 7.3 Beispiel: Timer

Wir beschränken uns ab jetzt auf die Eingabe mittels Schema. VHDL ist eine tolle Sache, es würde aber den Rahmen unseres Buches sprengen, einen Kurs in VHDL darin aufzunehmen. Wer sich dafür interessiert, findet gute Beiträge auf Youtube. Es gibt auch gute Einführungskurse der diversen Hersteller von programmierbarer Logik (Intel, Xilinx, Lattice, Micochip (Atmel)... ) und viele Bücher. Ich selber habe oft das Buch «VHDL: méthodologie de design et techniques avancées» von Thierry Schneider benutzt. Ich habe selber mit Thierry zusammen gearbeitet und das war auch der Grund ein französisches Buch zu kaufen.

Ich mache die folgenden Beispiele wieder für das Demo-Board mit dem MAX II. Es hat eine Taktfrequenz von 66 MHz.


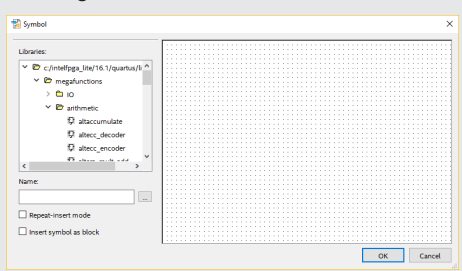
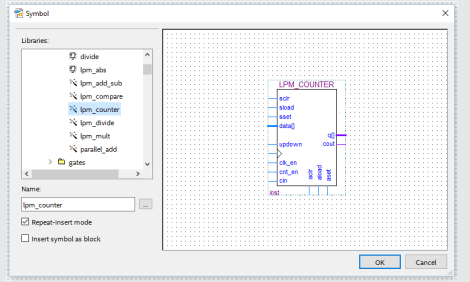
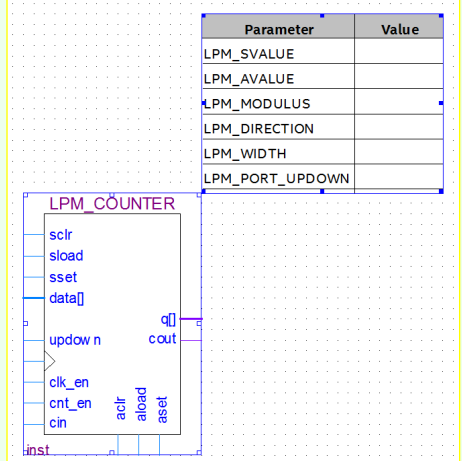
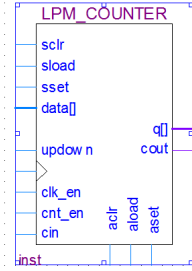
Als Erstes, lassen wir die zwei LEDs, welche wir schon für den Halb-Addierer benutzt haben mit einem Takt von 1 Hz alternierend blinken.

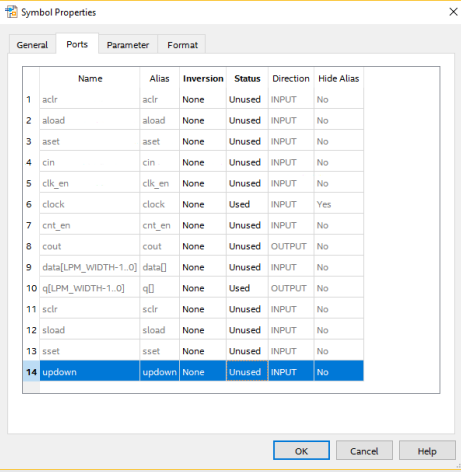
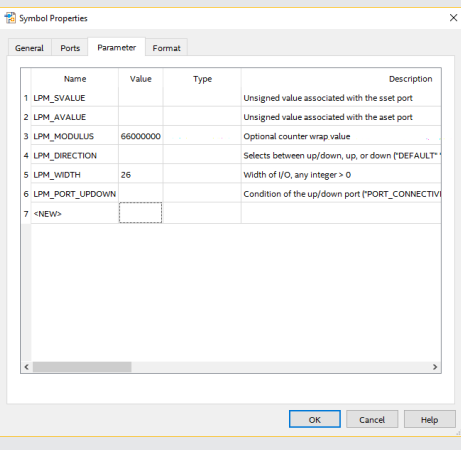
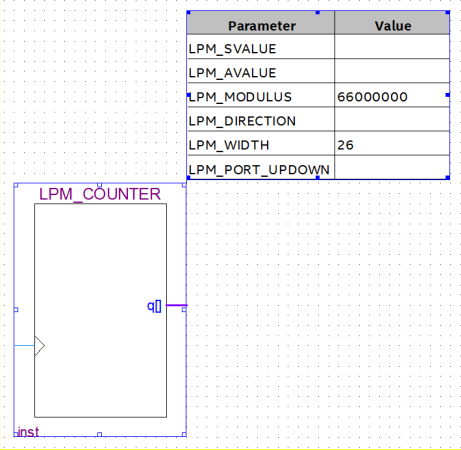
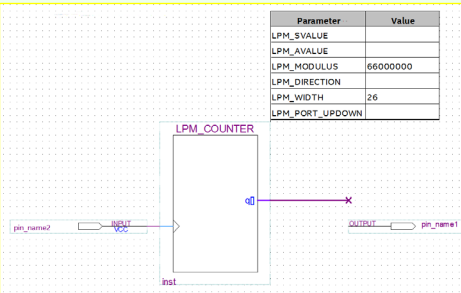
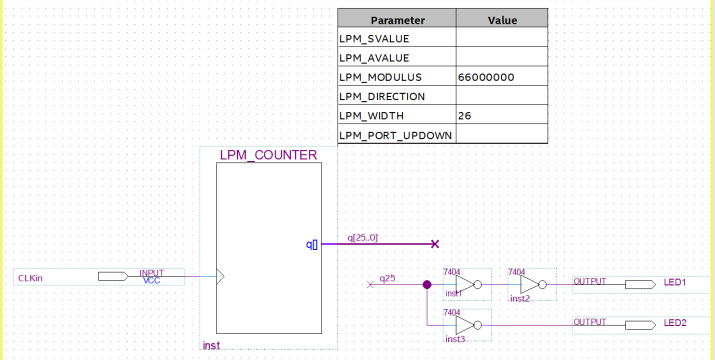
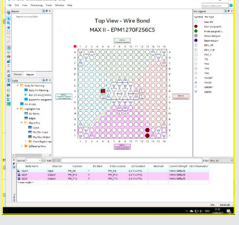
7

Wenn wir mit einem Takt von 66 MHz einen Takt von 1 Sekunde erzeugen wollen, müssen wir den Clock durch 66 000 000 teilen. Um herauszufinden, wie viele Teilerstufen eines Binärteilers wir dafür benötigen, rechnen wir:

- $\log 66\,000\,000 / \log 2 = \underline{25.97} \rightarrow$  Aufrunden auf 26 Stufen.
- Der «LMP\_Counter» kann so parametrisiert werden, dass er bis zu einer bestimmten Zahl zählt und dann neu beginnt mit  $\rightarrow$  Dem Parameter «LPM\_MODULUS» den Wert 66 000 000 zuweisen

Dafür eröffnen wir wieder ein Projekt, ein neues Schema und geben ihm einen passenden Namen, hier z.B. «Blinky».

Arbeits-schritt	Beschreibung														
<p>1</p>	<ul style="list-style-type: none"> <li>• Das Menu «Symbol Tool»  in der Menüleiste oben anklicken die Bibliothek «megafunctions &gt; arithmetic» anwählen</li> </ul>  <ul style="list-style-type: none"> <li>• Runterscrollen bis das Modul «LMP_COUNTER» erscheint</li> <li>• Das Modul «LMP_COUNTER» anklicken</li> </ul> 														
<p>2</p>	<p><b>Modul parametrieren:</b>                      Beim meiner aktuellen Installation von Quartus wird der «Mega Wizzard» nicht gestartet. Kann sein, dass der nicht mehr vorhanden ist, wir suchen aber wir nicht lange herum, sondern klicken mit der rechten Maustaste auf das Symbol &gt; Properties und schon haben wir ein Menü in welchem wir den Baustein «LMP_COUNTER» für unsere Bedürfnisse konfigurieren können.</p>  <table border="1" data-bbox="1157 1635 1412 1814"> <thead> <tr> <th>Parameter</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>LPM_SVALUE</td> <td></td> </tr> <tr> <td>LPM_AVALUE</td> <td></td> </tr> <tr> <td>LPM_MODULUS</td> <td></td> </tr> <tr> <td>LPM_DIRECTION</td> <td></td> </tr> <tr> <td>LPM_WIDTH</td> <td></td> </tr> <tr> <td>LPM_PORT_UPDOWN</td> <td></td> </tr> </tbody> </table> 	Parameter	Value	LPM_SVALUE		LPM_AVALUE		LPM_MODULUS		LPM_DIRECTION		LPM_WIDTH		LPM_PORT_UPDOWN	
Parameter	Value														
LPM_SVALUE															
LPM_AVALUE															
LPM_MODULUS															
LPM_DIRECTION															
LPM_WIDTH															
LPM_PORT_UPDOWN															

Arbeits-schritt	Beschreibung
<p><b>3</b></p>	<p><b>Parameter:</b> Wir schalten alle I/O-Optionen, ausser die beiden die wir benötigen, auf «unused»:</p> <ul style="list-style-type: none"> <li>• clock</li> <li>• q[LMP_WIDTH-1..0]</li> </ul>  <p><b>Format:</b> Hier schreiben wir die zwei Werte rein, welche wir für das Teilen benötigen (die haben wir vorangehende berechnet):</p> <ul style="list-style-type: none"> <li>• LMP_WIDTH = 26</li> <li>• LMP_MODULUS = 66 000 000</li> </ul> 
<p><b>4</b></p>	<p><b>Reduzierte Funktion des Moduls:</b></p>  <p><b>Wir platzieren jetzt noch die I/Os:</b></p> <ul style="list-style-type: none"> <li>• Inputs und Outputs</li> <li>• Eine Buslinie, welche wir offen lassen</li> </ul> 
<p><b>5</b></p>	<p><b>Das Schema ist jetzt bereit:</b></p> <ul style="list-style-type: none"> <li>• Eingang «CLKin» ist benannt</li> <li>• Der Ausgang «q25» wir noch invertiert damit wir 2 invertierte Ausgänge «LED1» und «LED2» erhalten, damit sie auch alternierend blinken werden.</li> </ul>  <p><b>Zuweisung der Pins:</b></p> <ul style="list-style-type: none"> <li>• CLKin → PIN_H5</li> <li>• LED1 → PIN_R13</li> <li>• LED2 → PIN_T13</li> </ul> 

Programmieren → es blinkt!

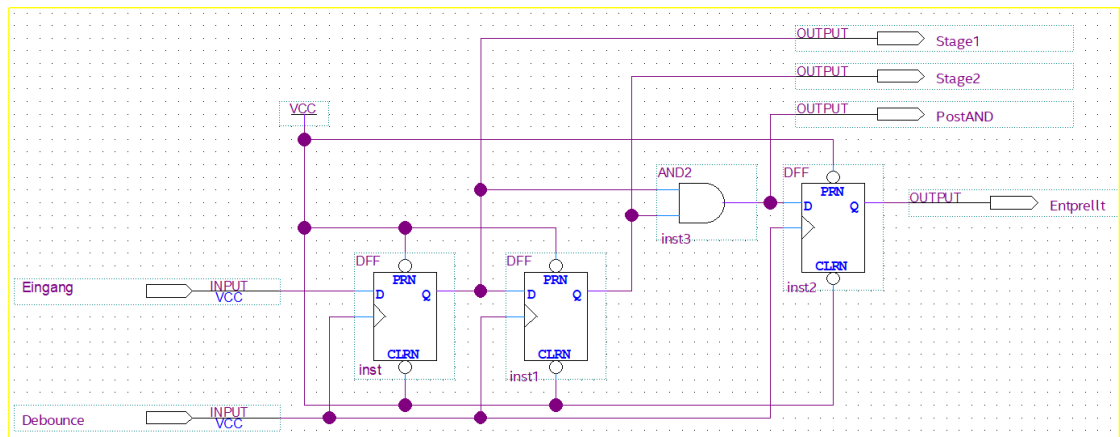
**Ressourcen**  
Logic elements: 37/1270  
Total pins: 3/212

### 7.3.1 Entprellen mit Flip-Flops

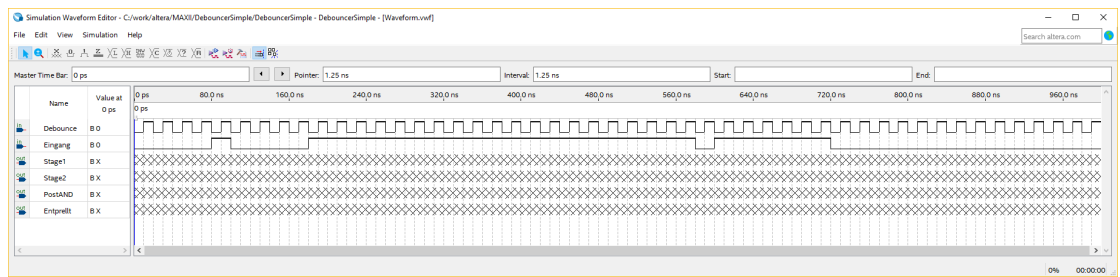
#### 7.3.1.1 Stufe 1: nur Einschalten mit D-FF entprellt:

Ressourcen  
Logic elements:  
3/1270  
Total pins:  
6/212

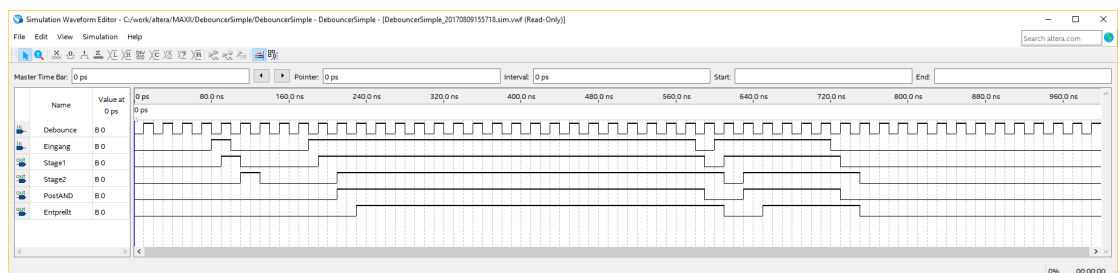
7



Die Signale «Stage1», «Stage2» und «PostAND» habe ich nur hinzugefügt, damit wir in der Simulation «in die Schaltung» hineinschauen können.



In der Simulation steht das «high»-Signal zuerst nur kurz an, beim zweiten mal bleibt das Signal «high». Danach machen wir dasselbe mit dem «low»-Signal, zuerst einmal nur kurz «low» and danach bleibt es «low».



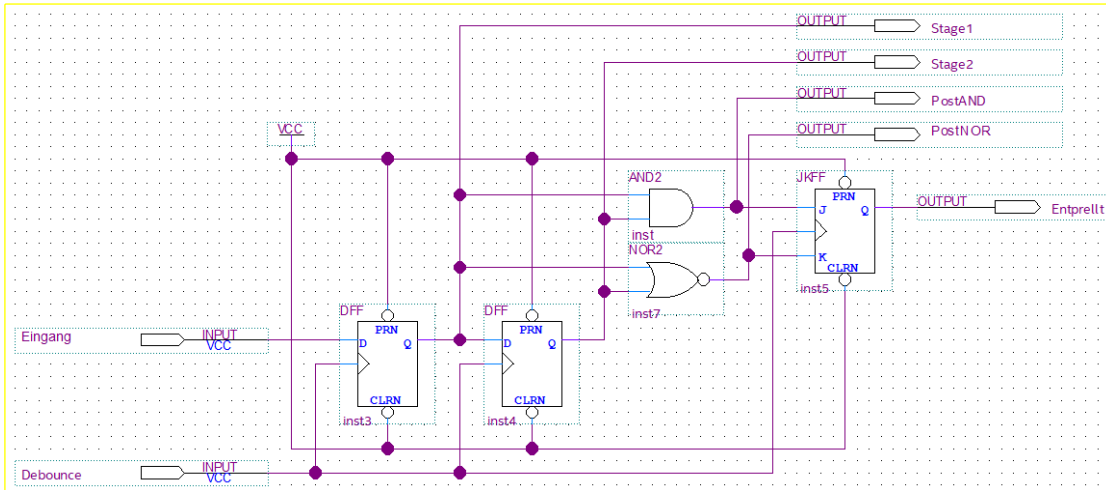
Es ist schön ersichtlich, wie das «high»-Signal von den Flip-Flops «weitergereicht» wird. Es reicht aber zuerst nicht, dass beide Eingänge des AND-Gates gleichzeitig «high» sind. Der Ausgang wird nicht gesetzt.

Wenn das Signal länger auf «high» ist, reicht es, dass der Ausgang geschaltet wird.

Ist der Ausgang «high» und der Eingang geht auf «low» wird der Ausgang schon beim nächsten Clock-Signal aus «low» gesetzt.

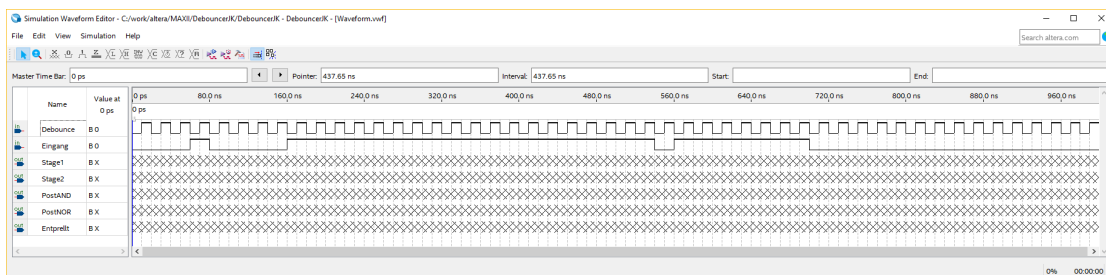
Es wird also nur das Einschalten entprellt, das Ausschalten nicht.

7.3.1.2 Stufe 2: beide Flanken mit JK-FF entprellt :

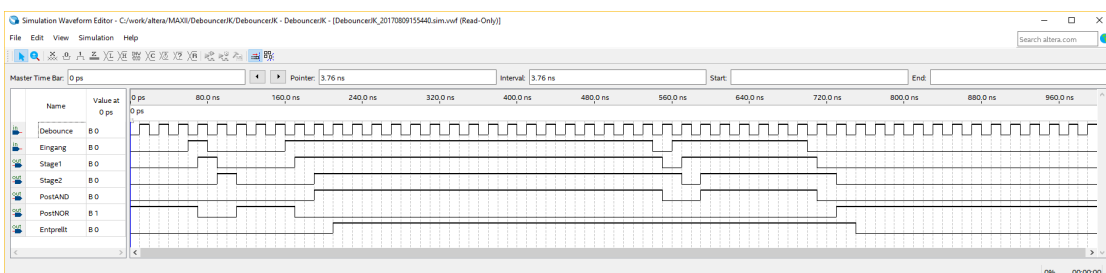


Ressourcen  
 Logic elements:  
 3/1270  
 Total pins:  
 7/212

Jetzt benutzen wir ein JK-Flip-Flop. Eingeschaltet wird es, wenn der J Eingang «high» ist (wenn beim AND-Gatter beide Eingänge «high» sind). Ausgeschaltet wird es, wenn der K Eingang «high» ist (wenn beim NOR-Gatter alle Eingänge «low» sind).



In der Simulation steht das «high»-Signal nur kurz an und der Ausgang reagiert nicht. Steht das «high»-Signal länger an, schaltet der Ausgang auf «high». Ist der Ausgang «high» richtet ein kurzes «low»-Signal am Eingang nichts aus, es muss schon 3 Clocks (Debounce) anstehen, bis der Ausgang wieder auf «low» geht.



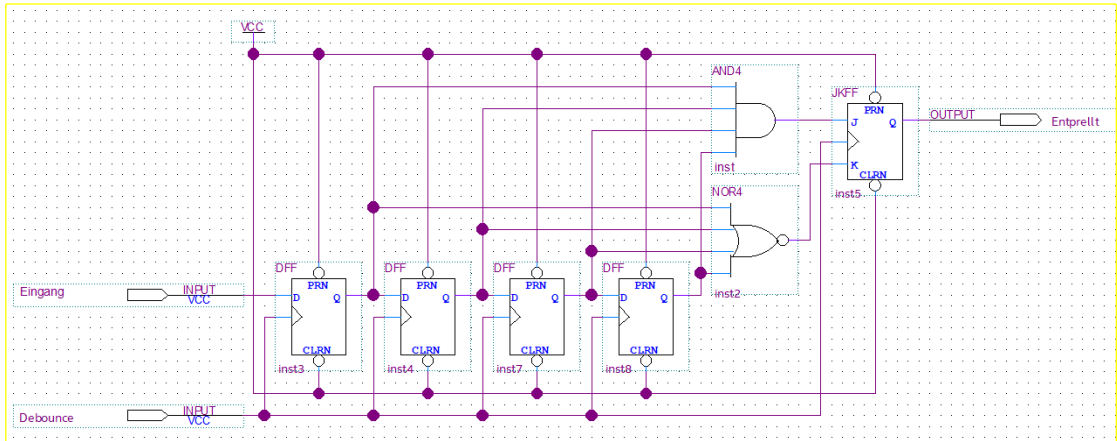
Jetzt werden das Einschalten und das Ausschalten entprellt.

Der CMOS-Baustein MC14490 ist intern ähnlich aufgebaut. Er hat aber 4 Stufen, das können wir auch machen indem wir weitere Flip-Flops anhängen und die zweifach Logik-Gatter durch 4-fache ersetzen:

AND2 → AND4

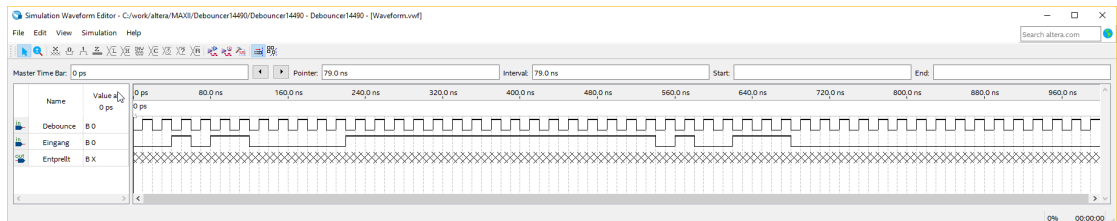
NOR2 → NOR4

7.3.1.3 Stufe 3: 4-Stufiges Entprellen mit JK-FF :

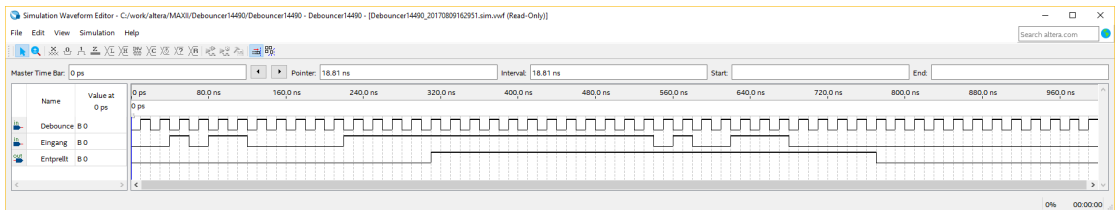


7

Das Schieberegister wurde jetzt um 2 Stufen erweitert. Auf die Zustände der einzelnen Stufen verzichten wir jetzt, das Prinzip haben wir ja begriffen.



In der Simulation steht das «high»-Signal zuerst nur kurz an, dann dauert es immer länger. Danach machen wir dasselbe mit dem «low»-Signal



Meine absolute Liebesschaltung. Das ist «boolesche Poesie». Einfach, effizient und erweiterbar. In unserem Buch Mechatronik Band 1 haben wir auf den Seite 87/88 beschrieben wie man mit einem PIC16F876 Eingänge mit grossem Aufwand entprellen kann. Die obige Schaltung können wir zu einem Block (Zwei Eingänge und ein Ausgang) zusammenfassen und in unserem nächsten Projekt für alle unsere Eingänge einsetzen.

Die Clockfrequenz müssen wir aber jeweils anpassen. In unserer Simulation läuft er viel zu schnell. Wollen wir aber einen billigen Taster entprellen, benutzen wir einen Takt von 50... 100 ms. Soll der Clockeingang des folgenden Sequenzers entprellt werden und der Motor soll mit bis zu 1 kHz getaktet werden, müssen wir einen viel kleineren Entprelltakt «debounce» wählen, z.B. 100 us oder noch schneller.


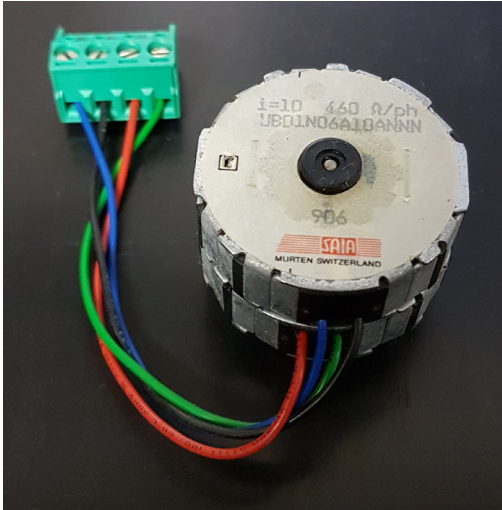
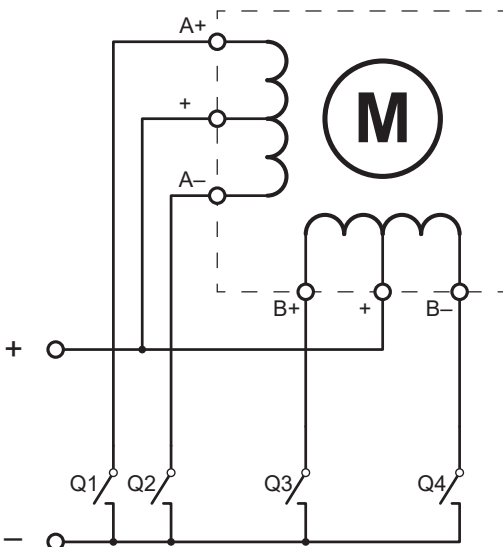
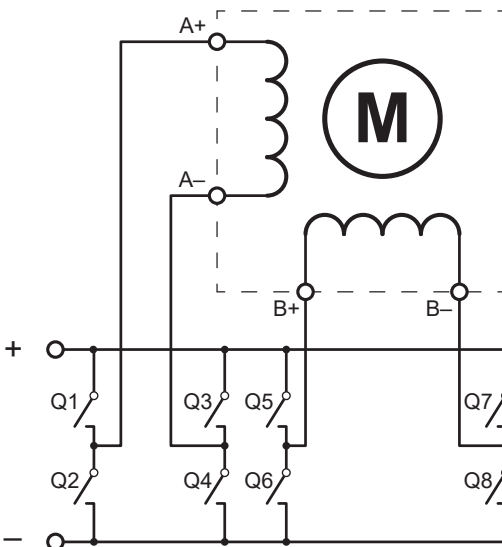
### 7.4 Beispiel: Sequenzer für Schrittmotoren

Zu allererst schauen wir uns die verschiedenen Schrittmotortypen an. Es gibt handelsübliche Schrittmotoren mit 1, 2, 3, 5 und mehr Wicklungen (Phasen).

- Wenn man die Anzahl produzierter Lavet-Motoren (Einphasenschrittmotor) für Armbanduhren nicht berücksichtigt, sehen wir diese eher selten bis nie. Ein Beispiel haben wir im Kapitel «kleinste Mikrokontroller» angeschaut
- Einen 3-Phasenschrittmotor habe ich noch nie benutzt
- 5-Phasenschrittmotoren benutzen wir, wenn wir eine höhere Laufruhe benötigen. Der Hardware-Aufwand und der Programmieraufwand werden aber recht gross
- 2-Phasenschrittmotoren sind die gängigsten und davon gibt es von der Beschaltung her 2 Varianten:
  - Unipolare Motoren
  - Bipolare Motoren

In den folgenden Beispielen werden wir uns mit 2-Phasenschrittmotoren befassen.

#### 7.4.1 Einleitung: Unipolare und Bipolare Schrittmotoren

Unipolarer Schrittmotor	Bipolarer Schrittmotor
	
	

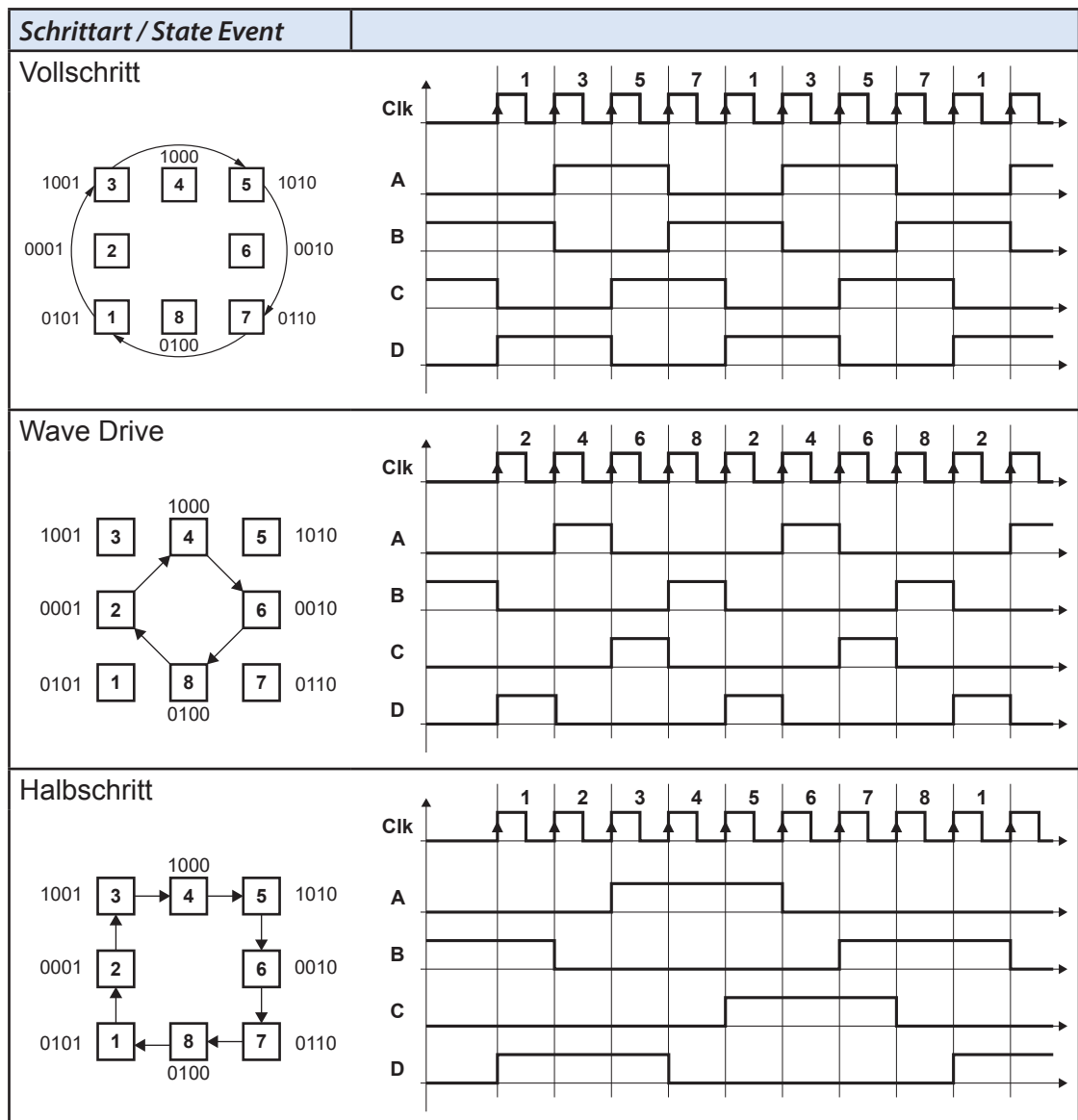
Schaltungsvarianten der beiden Schrittmotortypen



Wir werden hier nicht die Motoren beschreiben, darüber wurde schon genug geschrieben. Wir wollen ja einen Sequenzer für die Ansteuerung 2-phasiger Schrittmotoren aufbauen. Dafür suchen wir zuerst die notwendigen Sequenzfolgen. In der Literatur und bei den verschiedenen Herstellern gibt es leider unterschiedliche Benennungen der Anschlüsse. Wir vergleichen mal ein paar:

Spule 1			Spule 2		
Anfang	Mittelabgriff	Ende	Anfang	Mittelabgriff	Ende
A+	+	A-	B+	+	B-
A	+	A'	B	+	B'
A	O	B	C	O	D
1A		1B	2A		2B
1a		1b	2a		2b
A1		A2	B1		B2
A		A_n	B		B_n
Q1		Q2	Q3		Q4

Und hier sind die Sequenzen zu den verschiedenen Schrittarten:



Wahrheitstabellen der verschiedenen Schrittarten:

Vollschritt				
Schritt	Ausgänge			
	A	B	C	D
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	1	0	1	0
5	0	1	0	1
6	1	0	1	0
7	0	1	0	1
8	1	0	1	0
1	0	1	0	1
"	"	"	"	"

Tabelle 7.1

Wave Drive				
Schritt	Ausgänge			
	A	B	C	D
1	0	0	0	1
2	1	0	0	0
3	0	0	1	0
4	1	0	0	0
5	0	0	1	0
6	1	0	0	0
7	0	0	1	0
8	1	0	0	0
1	0	0	1	0
"	"	"	"	"

Tabelle 7.2

Halbschritt				
Schritt	Ausgänge			
	A	B	C	D
1	0	1	0	1
2	0	0	0	1
3	1	0	0	1
4	1	0	0	0
5	1	0	1	0
6	0	0	1	0
7	0	1	1	0
8	0	1	0	0
1	0	1	0	1
"	"	"	"	"

Tabelle 7.3

Beim Vollschritt-Modus werden nur die ungeraden Schritte der Halbschritttabelle ausgeführt. Beim Wave-Drive-Modus werden nur die geraden Schritte der Halbschritttabelle ausgeführt. Wenn wir einen universellen Driver aufbauen wollen, müssen wir bei diesen beiden Modi immer einen Schritt überspringen.

Wenn wir aber einen Driver für nur für den Wave-Drive oder nur für den Vollschrittmodus aufbauen wollen, straffen wir das Ganze und zählen nur die «ungeraden, oder die «geraden» Schritte. Damit ergeben sich für diese beiden Modi folgende vereinfachte Tabellen:

Vollschritt				
Schritt	Ausgänge			
	A	B	C	D
1	0	1	0	1
2	1	0	0	1
3	1	0	1	0
4	0	1	1	0
1	0	1	0	1
"	"	"	"	"

Tabelle 7.4

Wave Drive				
Schritt	Ausgänge			
	A	B	C	D
1	0	0	0	1
2	1	0	0	0
3	0	0	1	0
4	0	1	0	0
1	0	0	0	1
"	"	"	"	"

Tabelle 7.5

Diese Tabellen entsprechen dem altbewährten Sequenzer L297, anstelle von der Triggerrung auf negative Flanken, habe ich hier auf positive Flanken getriggert.

In der Literatur und im Internet findet man eine Fülle von ähnlichen und abweichenden Sequenzfolgen. Je nach Logik sind sie einfach invertiert oder für stromlose Spulen ist 2 mal «1» anstelle von 2 mal «0» angegeben (Bei einem bipolaren Schrittmotor fließt kein Strom, wenn beide Seiten «high» oder beide Seiten «low» sind, bei einem unipolaren Schrittmotor muss man schauen ob der Mittelabgriff auf +V oder auf GND geschaltet ist und dafür die Wahrheitstabelle anpassen), zum Teil sind auch nur die Spalten vertauscht (B, A; D, C), es gibt aber auch andere abweichende Sequenzen. Wir gehen dem hier nicht auf den Grund und arbeiten mit obigen Sequenzen.

### 7.4.1 Vorbereitung: Symbole, Terminologie, Schreibweisen logischer Ausdrücke und boolesche Algebra

Wir haben bisher schon von «HIGH» und «LOW» gesprochen. In obigen Tabellen und Diagrammen hat es «0» und «1». Ich benutze wenn es geht nur positive Logik und da werde ich ab jetzt noch öfter die Begriffe «Nullen» und «Einsen» benutzen

**Ab jetzt gilt für Begriffe:**

- NULL = LOW = 0 : meistens 0 Volt
- EINS = HIGH = 1 : meistens +5 Volt, manchmal auch +3 oder +3,3 Volt

**Boolesche Algebra:**

Wir haben bei den Halbaddierereen schon die logischen Verknüpfungen «AND» und «EXOR» benutzt. Machen wir doch mal eine kleine Übersicht über die logischen Verknüpfungen, welche wir ab jetzt benutzen:

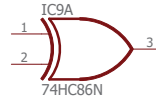
7

Verknüpfung	Alte Symbole DIN 40700 (vor 1976)	IEC-Symbole IEC 60617-12	Schreibweise in Boolescher Algebra
AND			Die UND-Verknüpfung $Q = A \cdot B$ $Q = AB$
NAND			Die NICHT-UND-Verknüpfung $Q = \overline{A \cdot B}$ oder $\overline{Q} = A \cdot B$ $Q = \overline{AB}$ oder $\overline{Q} = AB$
OR			Die ODER-Verknüpfung $Q = A + B$
NOR			Die NICHT-ODER-Verknüpfung $Q = \overline{A + B}$ oder $\overline{Q} = A + B$
EXOR			Die EX-OR-Verknüpfung (auch Entweder-Oder) $Q = A \oplus B$
EXNOR			Die NICHT-EXOR-Verknüpfung $Q = \overline{A \oplus B}$ oder $\overline{Q} = A \oplus B$
Invertieren			Die NICHT-Verknüpfung $Q = \overline{A}$ oder $\overline{Q} = A$
Clock- eingang			Beim Beispiel D-FF (Flip-Flop) sehen wir die Symbolik eines Clockeinganges und invertierter Eingänge

Ich persönlich finde die alten DIN-Symbole für einfache Verknüpfungen viel übersichtlicher. In Schaltungen mit vielen Gattern sind die Funktionen einfacher erkennbar. Bei komplexen Schaltungen (Schieberegister, Zähler, Multiplexer usw.) sind die neuen IEC-Symbole unschlagbar. Sie sind logisch und praktisch, hingegen bin ich bei AND, OR und EXOR mit der IEC-Symbolik immer wieder unsicher.

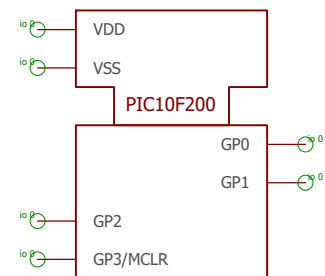


Für die EXOR-Verknüpfung sieht man auch das Symbol :



Ich persönlich hatte das alte US-Symbol bevorzugt:

Weiterhin legen wir fest, dass Eingänge immer links und Ausgänge immer rechts von den Symbolen und der Schemablätter gezeichnet werden. Schema zeichne ich immer so, dass die Leserichtung (Weitergabe von Signalen) von links nach rechts und von oben nach unten erfolgt (Natürlich geht es nicht ohne Ausnahmen, z.B. Rückkopplungen). Als Beispiel sei hier das Symbol eines PIC10F200 mit 2 Eingängen und 2 Ausgängen gezeigt:



Wenn das Schema einfacher wird, erzeuge ich ein individuelles Symbol, das an die I/O-Benutzung angepasst ist.

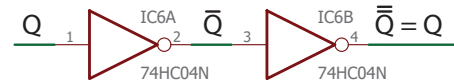
Das kann nicht immer zu 100% eingehalten werden. Im Zweifelsfalle schaue ich auf die Lesbarkeit der Schemas.

### 7.4.1.1 Vorbereitung: Symbole, Terminologie, Schreibweisen logischer Ausdrücke und boolesche Algebra

Ich benutze hier eine Schreibweise, ähnlich der normalen Algebra. Andere gebräuchlichen Schreibweisen sind mit einem Textverarbeitungstool meistens nicht kompatibel.

- Eine AND-Verknüpfung schreiben wir mit einem  $\bullet$   
Beispiel:  $Q = A \bullet B$ , wie in der normalen Algebra (Mathematik) kann ich den Punkt auch weglassen  $\rightarrow Q = AB$
- Eine OR-Verknüpfung schreiben wir mit einem  $+$   
Beispiel:  $Q = A + B$
- Eine EXOR-Verknüpfung schreiben wir mit einem  $\oplus$   
Beispiel:  $Q = A \oplus B$

- Die Darstellung eines invertierten Signales ist ein Querstrich.  
Beispiel:  $\bar{Q}$



Zwei Querstriche heben sich auf:  $\bar{\bar{Q}} = Q$

- Ein Querstrich kann aufgehoben werden, wenn wir die Funktion austauschen AND  $\rightarrow$  OR oder OR  $\rightarrow$  AND.

Beispiele:

$Q = \overline{A + B} = \bar{A} \cdot \bar{B}$		
Eingänge		Ausgang
B	A	Q
0	0	1
0	1	0
1	0	0
1	1	0

Tabelle 7.6

$Q = \overline{A \cdot B} = \bar{A} + \bar{B}$		
Eingänge		Ausgang
B	A	Q
0	0	1
0	1	1
1	0	1
1	1	0

Tabelle 7.7

7

- Dank dieser Schreibweise gilt für die Funktionen AND und OR, wie in der normalen Mathematik: **Punkt vor Strich**

- Die Klammerregeln gelten auch für die boolesche Algebra

- Das Kommutativgesetz gilt auch für boolesche Ausdrücke.

Beispiel:  $Q1 = A \bullet B = B \bullet A$

$Q2 = A + B = B + A$

$Q3 = A \oplus B = B \oplus A$

Das wenden wir intuitiv richtig an.

- Das Assoziativgesetz gilt auch für boolesche Ausdrücke.

Beispiel:  $Q1 = A \bullet (B \bullet C) = (A \bullet B) \bullet C$

$Q2 = A + (B + C) = (A + B) + C$

- Das Distributivgesetz gilt auch für boolesche Ausdrücke.

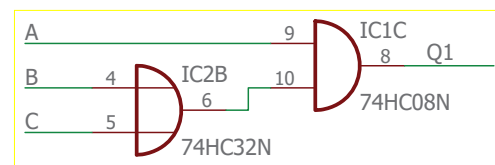
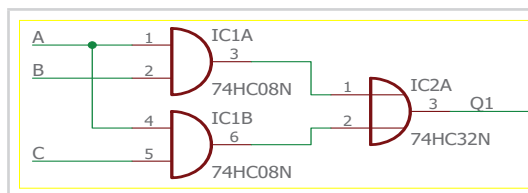
Beispiel:  $Q1 = A \bullet (B + C) = (A \bullet B) + (A \bullet C)$

$Q2 = (A + B) \bullet (C + D) = (A \bullet C) + (A \bullet D) + (B \bullet C) + (B \bullet D)$

Das erste Beispiel Q1 wenden wir an, wenn wir boolesche Bergiffe vereinfachen wollen (Ausklammern)

$Q1 = (A \bullet B) + (A \bullet C) = A \bullet (B + C)$

Und in der in der Praxis:



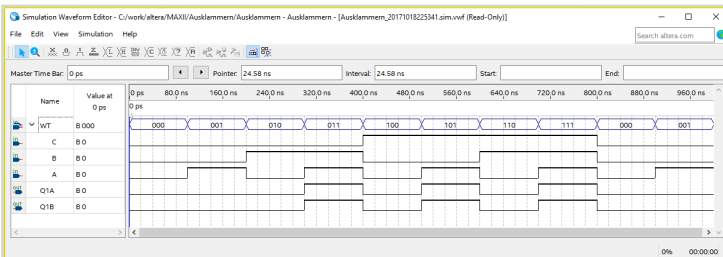
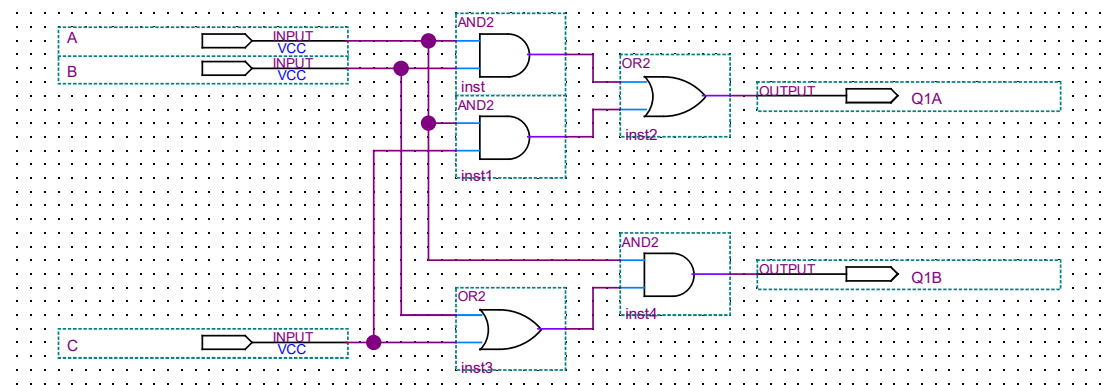
- Und wenn man nicht sicher ist, kann man dies in 10 Minuten in Quartus eingeben und simulieren:

Date: October 18, 2017

Ausklammern.bdf

Project: Ausklammern

Ressourcen  
Logic elements:  
1/1270  
Total pins:  
5/212



Das Verhalten von Q1A und Q1B ist identisch. Das bestätigt obige Behauptung

- Wir definieren hier, dass wir nur EXOR-Veknüpfungen mit 2 Eingängen benutzen. Über die Funktionsweise von EXOR mit mehr als 2 Eingängen, gehen die Meinungen auseinander. Je nach Interpretation funktioniert eine solche Schaltung als Paritätstester (ungerade Anzahl Eingänge «high»), oder als Detektor ob nur ein einziger Eingang «high» ist. Um sicher zu gehen, dass die Schaltung die Funktion ausübt welche wir wollen, schreiben wir unsere Wunschverhalten in eine Wahrheitstabelle und realisieren das konventionell mit den Standard-Gattern (AND, OR, NAND, NOR). Muss man eine Funktion mit IC's aufbauen, findet man ev. auch raffiniertere Lösungen. Dazu hier ein Beispiel:

Wenn wir wirklich eine Schaltung benötigen, welche testet, ob nur ein einziger Eingang «high» ist, ergibt sich folgende Wahrheitstabelle:

Eingänge						Ausgang
E5	E4	E3	E2	E1	E0	Q
0	0	0	0	0	0	0
0	0	0	0	0	1	1
0	0	0	0	1	0	1
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	1	0	0	0	0	1
1	0	0	0	0	0	1
Alle anderen Kombinationen						0

Tabelle 7.8

Wir müssen nicht alle 64 Zeilen des Wahrheitstabelle aufzeichnen.

Jetzt sind wir eigentlich schon dafür gerüstet, logische Schaltungen zu beschreiben und zu vereinfachen. Ab dem übernächsten Beispiel werden wir das hier gesehene benutzen.

### 7.4.2 Sequenzer mit JK-FlipFlop

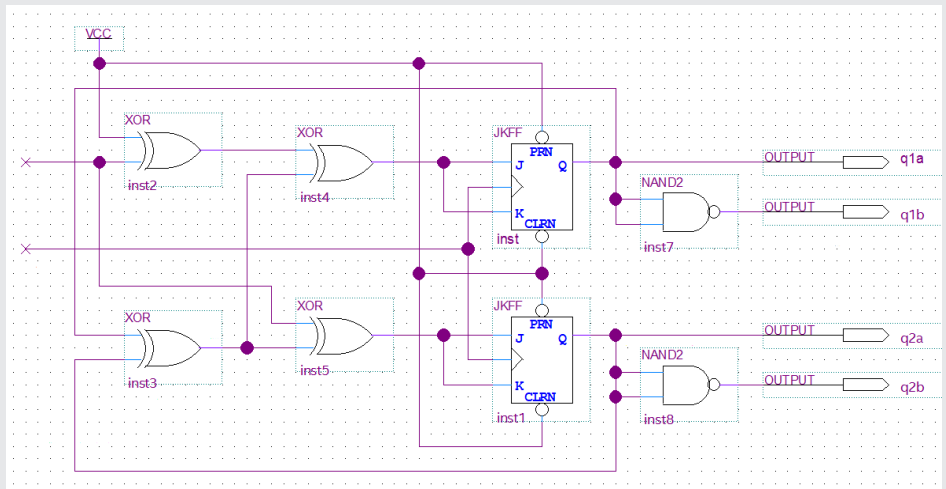
Bevor wir selber einen Sequenzer herleiten, machen wir einen ersten Test mit einer Standardschaltung. Damit können wir noch neue Funktionen von Quartus II lernen.

Wir testen diese Standardschaltung auf unserem bewährten MAX II Demo-Board mit 66 MHz. Den Motortakt simulieren wir über Tastendruck die Ausgänge geben wir auf die 4 LEDs.

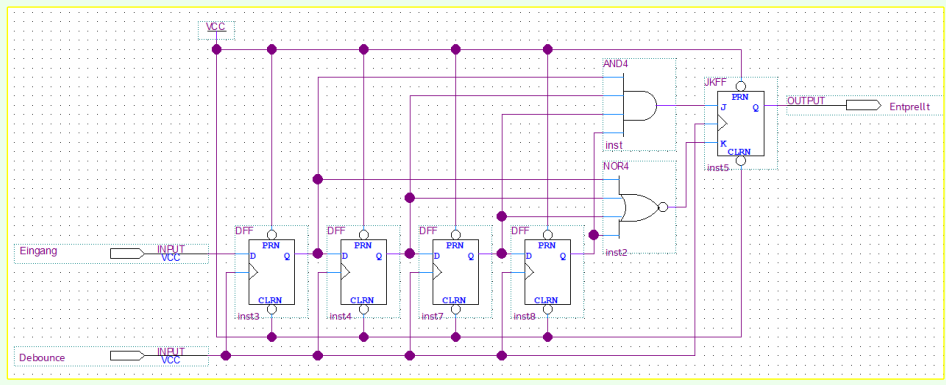
- Zuerst eröffnen wir eine neues Projekt «Sequenzer1»
- Dann geben wir das Grundschema des Sequenzers ein
- Dann erzeugen wir einen Schemablock für das Entprellen und einen für den Entprelltakt und platzieren diese im Hauptschema

7

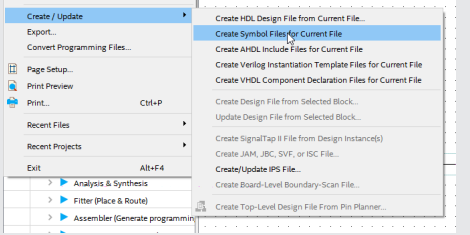
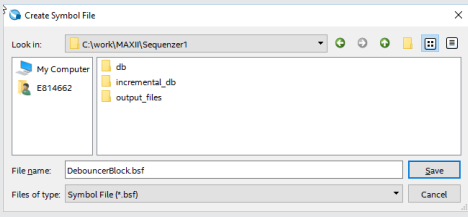
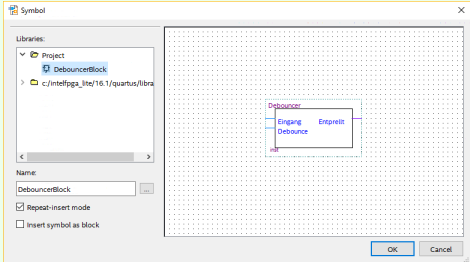
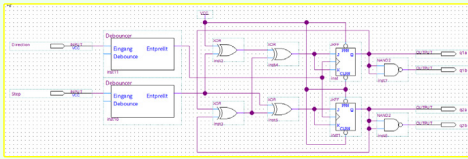
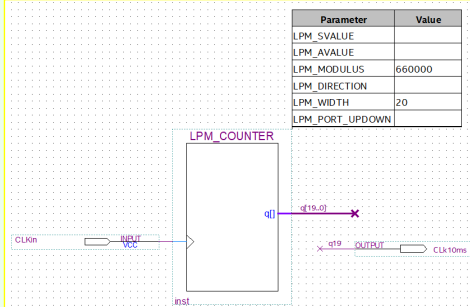
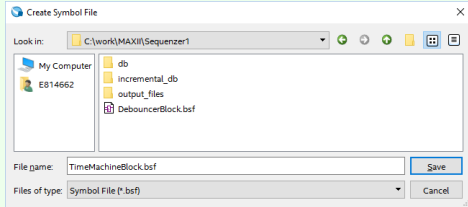
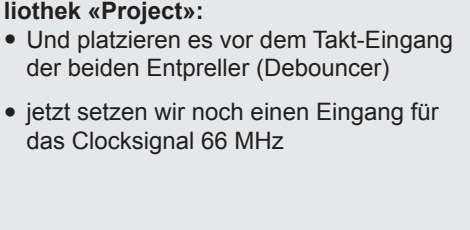
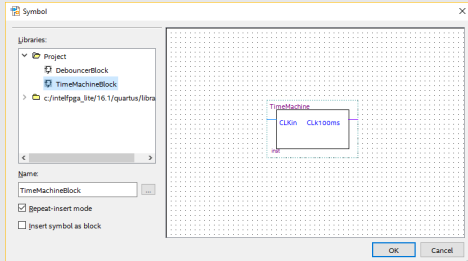
Ressourcen  
Logic elements:  
2/1270  
Total pins:  
6/212

Arbeits-schritt	Beschreibung
<b>1</b>	<p><b>Schema eingeben:</b> Wir geben den kompletten Sequenzer ein. Dies ist eine Lösung mit 4 EXOR-Gates und 2 JK-Flip-Flops. Da die Standard-JK-Flip-Flops keine invertierten Ausgänge haben, haben wir die 2 Q-Ausgänge mittels 2 NAND-Gates invertiert und erhalten somit die gewünschten Ausgänge. Diese Schaltung kann mit 2 Standard-ICs aufgebaut werden (1 mal 74HC73 und ein mal 74HC86).</p> 

Ressourcen  
Logic elements:  
5/1270  
Total pins:  
3/212

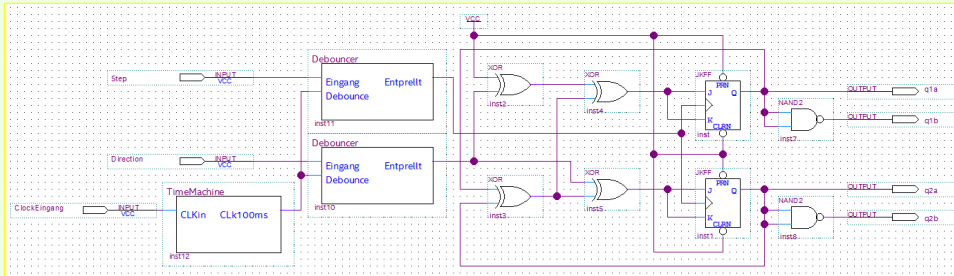
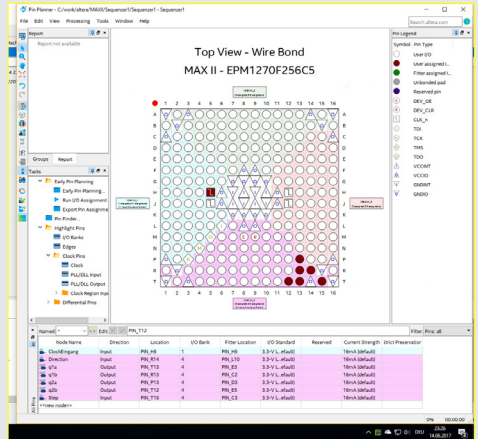
<b>2</b>	<p><b>Taster-Eingänge entprellen:</b> Wir haben im vorangehenden Kapitel eine Entprellschaltung angeschaut. Wir Erzeugen ein neues Schema, kopieren das Schema aus dem vorangehenden Kapitel hinein und erweitern es auf einen 4-stufigen-Entpreller:</p> 
----------	--



Arbeits-schritt	Beschreibung															
3	<p><b>Erzeugen eines Schemablockes (Symbol) aus diesem Schema</b></p> <p>Jetzt gehen wir ins Menü «File &gt; Create/Update &gt; Create Symbol Files for Current File»</p> 	<p><b>Den Block als neues Symbol abspeichern:</b></p> <ul style="list-style-type: none"> <li>• Sicherheitshalber ändere ich den Name von «Debouncer» auf «DebouncerBlock»</li> </ul> 														
4	<p><b>Wir holen das neue Symbol aus der Bibliothek «Project»:</b></p> <ul style="list-style-type: none"> <li>• Und platzieren je einen für die beiden Eingänge «Direction» und «Step»</li> </ul> 	<p><b>Das Schema ist jetzt schon fast fertig:</b></p> <ul style="list-style-type: none"> <li>• Es fehlt nur noch der Takteingang für die «Entpreller»</li> </ul>  <ul style="list-style-type: none"> <li>• Wir wählen mal 10 ms als Entprelltakt (das sollte für Taster ausreichend sein)</li> <li>• dafür müssen wir die Oszillatorfrequenz des Demo-Boardes durch 660 000 teilen</li> </ul>														
5	<p><b>Erzeugung des Taktsignals für das Entprellen:</b></p> <p>Wir haben im diesem Kapitel schon einen Timer zwei LEDs mit einem Takt von 1 Sekunde blinken lassen. Zum Entprellen benötigen wir jetzt einen Takt von 10 ms. Dafür geben wir in «Properties» neue Werte ein</p> <ul style="list-style-type: none"> <li>■ «LPM_MODULUS» = 660 000</li> <li>■ «LPM_WIDTH» = 20: aus <math>\log 660\,000 : \log 2 = 19.33</math> aufgerundet</li> </ul> <p><b>Neues Schema</b></p>  <table border="1" data-bbox="580 1249 756 1366"> <thead> <tr> <th>Parameter</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>LPM_SVALUE</td> <td></td> </tr> <tr> <td>LPM_AVALUE</td> <td></td> </tr> <tr> <td>LPM_MODULUS</td> <td>660000</td> </tr> <tr> <td>LPM_DIRECTION</td> <td></td> </tr> <tr> <td>LPM_WIDTH</td> <td>20</td> </tr> <tr> <td>LPM_PORT_UPDOWN</td> <td></td> </tr> </tbody> </table>	Parameter	Value	LPM_SVALUE		LPM_AVALUE		LPM_MODULUS	660000	LPM_DIRECTION		LPM_WIDTH	20	LPM_PORT_UPDOWN		<p><b>Erzeugen eines Schemablockes (Symbol) aus diesem Schema</b></p> <ul style="list-style-type: none"> <li>• Sicherheitshalber ändere ich den Namen indem ich «Block» anfüge</li> </ul> 
Parameter	Value															
LPM_SVALUE																
LPM_AVALUE																
LPM_MODULUS	660000															
LPM_DIRECTION																
LPM_WIDTH	20															
LPM_PORT_UPDOWN																
6	<p><b>Wir holen das neue Symbol aus der Bibliothek «Project»:</b></p> <ul style="list-style-type: none"> <li>• Und platzieren es vor dem Takt-Eingang der beiden Entpreller (Debouncer)</li> <li>• jetzt setzen wir noch einen Eingang für das Clocksignal 66 MHz</li> </ul> 															

Ressourcen  
Logic elements:  
29/1270  
Total pins:  
2/212

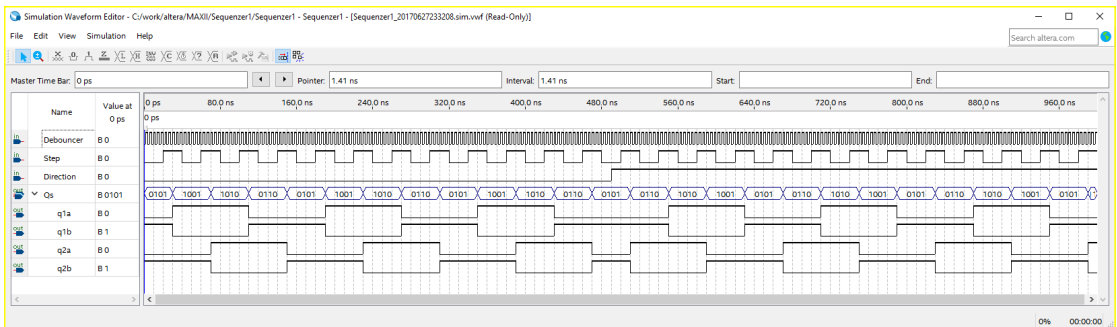
Ressourcen  
Logic elements:  
41/1270  
Total pins:  
7/212

Arbeits-schritt	Beschreibung
<b>7</b>	<p><b>Das Schema ist nun komplett</b></p> <ul style="list-style-type: none"> <li>• Kompilieren</li> </ul> 
<b>8</b>	<p><b>Zuweisung der Pins:</b></p> <ul style="list-style-type: none"> <li>• ClockEingang → PIN_H5 (Systemfrequenz)</li> <li>• Step → PIN_T15</li> <li>• Direction → PIN_R14</li> <li>• LED1 → PIN_R13</li> <li>• LED2 → PIN_T13</li> <li>• LED3 → PIN_P13</li> <li>• LED4 → PIN_T11</li> </ul> 

7

Zuerst simulieren wir die Schaltung um zu schauen ob das Schema in etwa stimmt.

Pattern am Ausgang:

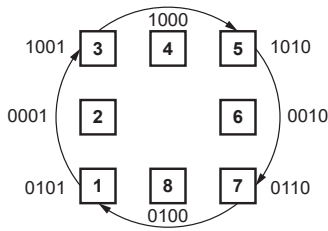


Wahrheitstabelle aus Pattern herausgelesen:

		Rückwärts				
Eingänge		Ausgänge				
Signal	Step	Direction	q1a	q1b	q2a	q2b
Start Schritt 1	↑	0	0	1	0	1
Schritt 2	↓	0	1	0	0	1
Schritt 3	↓	0	1	0	1	0
Schritt 4	↑	0	0	1	1	0
Schritt 5 = 1	↑	0	0	1	0	1
Schritt 6 = 2	↓	0	1	0	0	1
Schritt 7 = 3	↓	0	1	0	1	0

		Vorwärts				
Eingänge		Ausgänge				
Signal	Step	Direction	q1a	q1b	q2a	q2b
Start Schritt 1	↑	1	0	1	0	1
Schritt 4	↓	1	0	1	1	0
Schritt 3	↓	1	1	0	1	0
Schritt 2	↓	1	1	0	0	1
Schritt 1	↑	1	0	1	0	1
Schritt 4	↑	1	0	1	1	0
Schritt 3	↑	1	1	0	1	0

Vorwärts:



Mit folgenden Zuweisung der Flip-Flop-Ausgänge:

- q1a = A
- q1b = B
- q2a = C
- q2b = D

Dies entspricht dem Standard-Pattern für Vollschritt-Betrieb von Schrittmotoren.

Jetzt programmieren wir die Funktion ins Demo-Board programmieren → es funktioniert.

*Es geht noch einfacher. Im Internet findet man Schaltungen mit 2 Flip-Flops, ohne EXOR-Bausteine. Die können sich aber nur in eine Richtung bewegen unsere Schaltung kann sich immerhin schon Vor- und Rückwärts bewegen.*

*Wir wollen aber eine universelle Schaltung aufbauen. Dafür beginnen wir einem komplizierteren Aufbau und bauen den dann noch weiter aus.*

### 7.4.3 Wave-Drive-Sequenzer

Schrittart / State Event	Wave-Drive mit 2-Bit-Zähler								
	Schritt	Zählwert		Verknüpfung	Ausgänge				
		dezi-mal	binär		A	B	C	D	
		Q1	Q0						
	1	0	0	0	$\overline{Q0} \bullet \overline{Q1}$	0	0	0	1
	2	1	0	1	$Q0 \bullet \overline{Q1}$	1	0	0	0
	3	2	1	0	$\overline{Q0} \bullet Q1$	0	0	1	0
	4	3	1	1	$Q0 \bullet Q1$	0	1	0	0
	1	0	1	0	$\overline{Q0} \bullet \overline{Q1}$	0	0	0	1
	2	1	0	1	$Q0 \bullet \overline{Q1}$	1	0	0	0
	"	"	"	"	"	"	"	"	"

Tabelle 7.9

Aus der Wahrheitstabelle sehen wir, dass wir immer auf 4 zählen und dann wieder bei 1 beginnen. Das können wir mit einem binären Zähler erzeugen. Die binären Ausgänge des Zählers, werden wir so verknüpfen, dass sie die Signale A, B, C und D bilden.

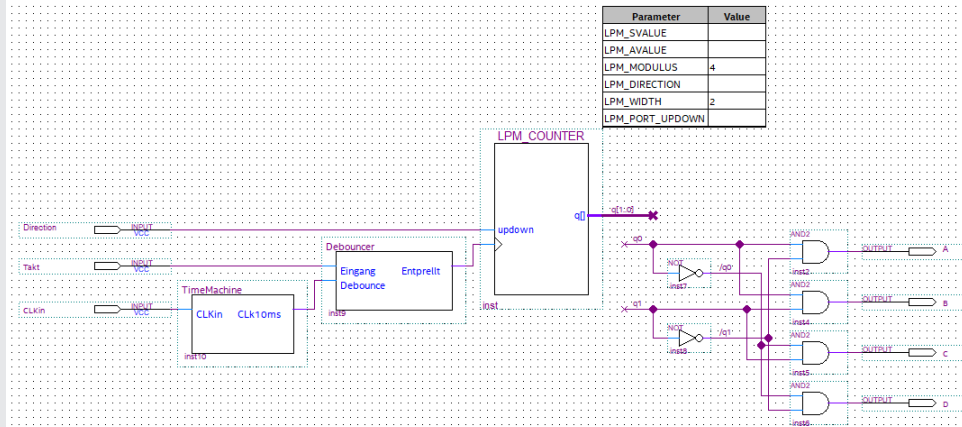
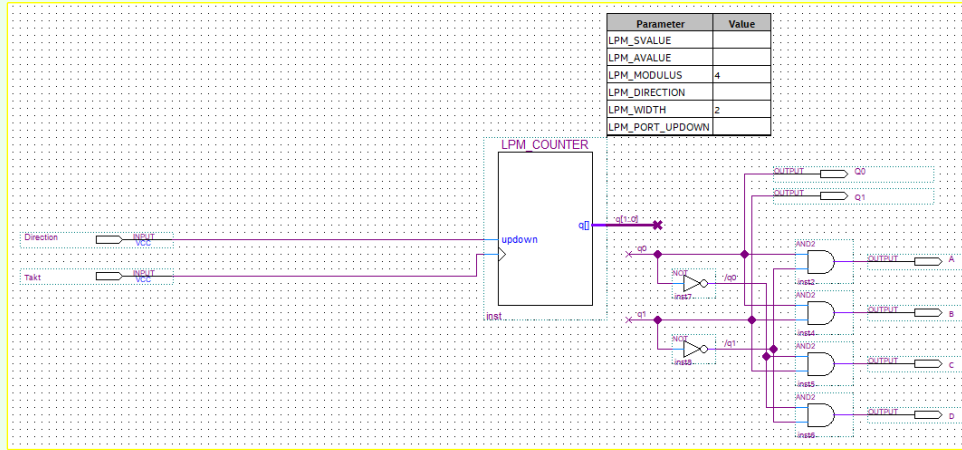
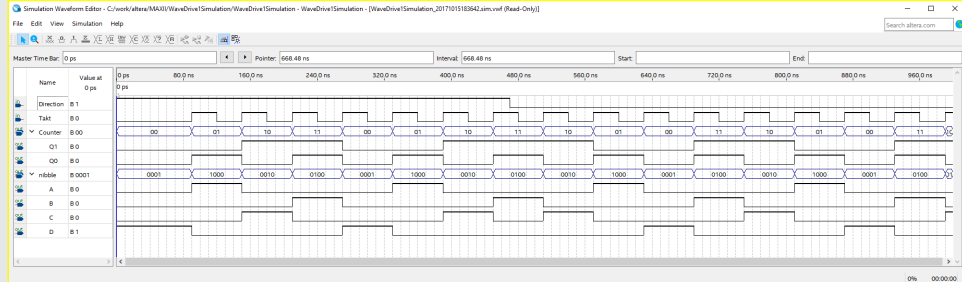
Wir können die notwendigen Verknüpfungen für die Ausgänge A...D aus der Tabelle 7.8 herauslesen:

- A =  $Q0 \bullet \overline{Q1}$
- B =  $Q0 \bullet Q1$
- C =  $\overline{Q0} \bullet Q1$
- D =  $\overline{Q0} \bullet \overline{Q1}$

7.4.3.1 Wave-Drive-Sequenzer mit Zähler

Ressourcen  
Logic elements:  
40/1270  
Total pins:  
7/212

7

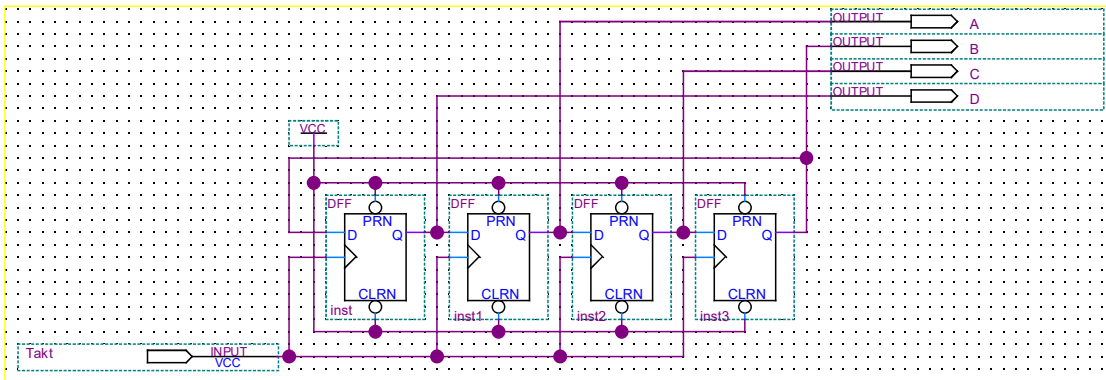
Arbeits-schritt	Beschreibung														
<b>1</b>	<p><b>Schema eingeben:</b> Wir geben den kompletten Sequenzer ein. Dafür benutzen wir den Zähler «LPM_COUNTER» aus der Bibliothek und stellen folgende Parameter ein (Analog zum Beispiel «7.4 Sequenzer für Schrittmotoren»):</p> <ul style="list-style-type: none"> <li>■ updown = used</li> <li>■ «LPM_MODULUS» = 4 ; wir zählen immer nur auf 4!</li> <li>■ «LPM_WIDTH» = 2 ; wir benötigen nur 2 Bit um auf 4 (0...3) zu zählen.</li> </ul> <div style="border: 1px solid #ccc; padding: 5px; margin: 10px 0;"> <table border="1"> <thead> <tr> <th>Parameter</th> <th>Value</th> </tr> </thead> <tbody> <tr><td>LPM_SVALUE</td><td></td></tr> <tr><td>LPM_AVALUE</td><td></td></tr> <tr><td>LPM_MODULUS</td><td>4</td></tr> <tr><td>LPM_DIRECTION</td><td></td></tr> <tr><td>LPM_WIDTH</td><td>2</td></tr> <tr><td>LPM_PORT_UPDOWN</td><td></td></tr> </tbody> </table> </div>  <p>Da wir die vorangehend schon aufgebauten Blöcke «TimeMachine» und «Debouncer» benutzen, müssen wir deren Files (Debouncer.bdf, Debouncer.bsf, TimeMachine.bdf und TimeMachine.bsf) aus dem alten Ordner in den neuen Ordner rüber kopieren.</p>	Parameter	Value	LPM_SVALUE		LPM_AVALUE		LPM_MODULUS	4	LPM_DIRECTION		LPM_WIDTH	2	LPM_PORT_UPDOWN	
Parameter	Value														
LPM_SVALUE															
LPM_AVALUE															
LPM_MODULUS	4														
LPM_DIRECTION															
LPM_WIDTH	2														
LPM_PORT_UPDOWN															
<b>2</b>	<p><b>Vorbereitung der Schaltungssimulation:</b> Für die Simulation lassen wir das Entprellen weg. Wir wissen ja das es funktioniert.</p> <div style="border: 1px solid #ccc; padding: 5px; margin: 10px 0;"> <table border="1"> <thead> <tr> <th>Parameter</th> <th>Value</th> </tr> </thead> <tbody> <tr><td>LPM_SVALUE</td><td></td></tr> <tr><td>LPM_AVALUE</td><td></td></tr> <tr><td>LPM_MODULUS</td><td>4</td></tr> <tr><td>LPM_DIRECTION</td><td></td></tr> <tr><td>LPM_WIDTH</td><td>2</td></tr> <tr><td>LPM_PORT_UPDOWN</td><td></td></tr> </tbody> </table> </div> 	Parameter	Value	LPM_SVALUE		LPM_AVALUE		LPM_MODULUS	4	LPM_DIRECTION		LPM_WIDTH	2	LPM_PORT_UPDOWN	
Parameter	Value														
LPM_SVALUE															
LPM_AVALUE															
LPM_MODULUS	4														
LPM_DIRECTION															
LPM_WIDTH	2														
LPM_PORT_UPDOWN															
<b>3</b>	<p><b>Simulation</b></p>  <p>Das entspricht genau dem Pattern der Wahrheitstabelle. Programmieren ins Demoboard und testen → funktioniert wie gewünscht.</p>														

Ressourcen  
Logic elements:  
6/1270  
Total pins:  
8/212

Wir haben ja vorangehend boolsche Vereinfachungen angeschaut, so wie dieser Wave-Drive Sequenzer aufgebaut ist, gibt es nichts zu vereinfachen. Wenn man aber die Wahrheitstabelle anschaut, könnte man auf die Idee kommen den Sequenzer mit einem Schieberegister zu realisieren. Das schauen wir uns schnell mal an:

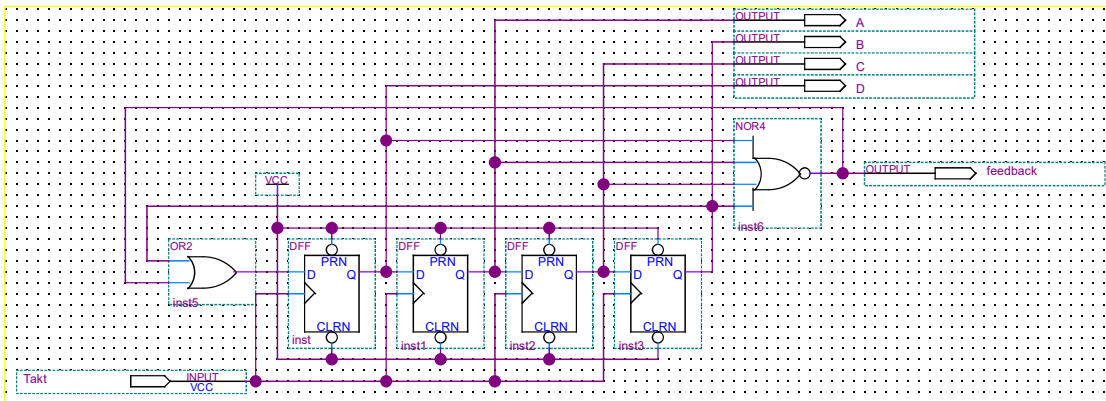
### 7.4.3.2 Wave-Drive-Sequenzer mit Schieberegister (D-FF)

Mit 4 D-Flip-Flop ist ein Schieberegister schnell aufgebaut. Wird der letzte Ausgang an den Eingang des Schieberegisters zurückgeführt wird die EINS immerfort von einem D-FF zum nächsten weitergeleitet. Unser Problem ist bei folgender Schaltung, dass wir gar nie eine EINS zum weiterleiten haben.



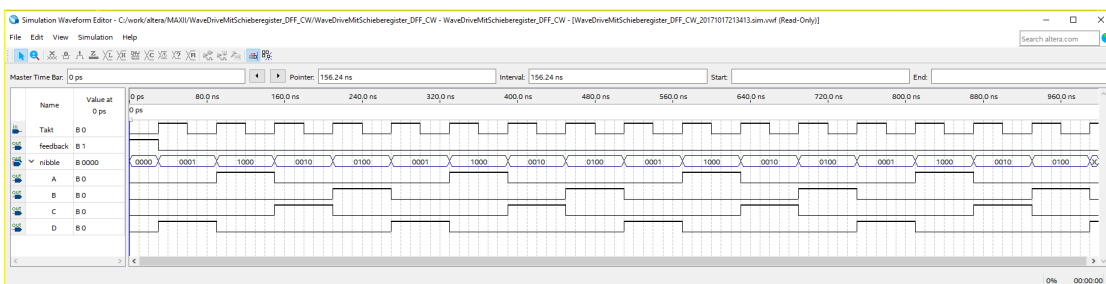
Ressourcen  
Logic elements:  
4/1270  
Total pins:  
6/212

Wir testen ob alle 4 Ausgänge NULL sind. In diesem Falle leiten wir eine EINS an den ersten Eingang. Das machen wir mit einem NOR- und einem OR-Gatter.



Ressourcen  
Logic elements:  
4/1270  
Total pins:  
6/212

Und hier ist noch die Simulation:

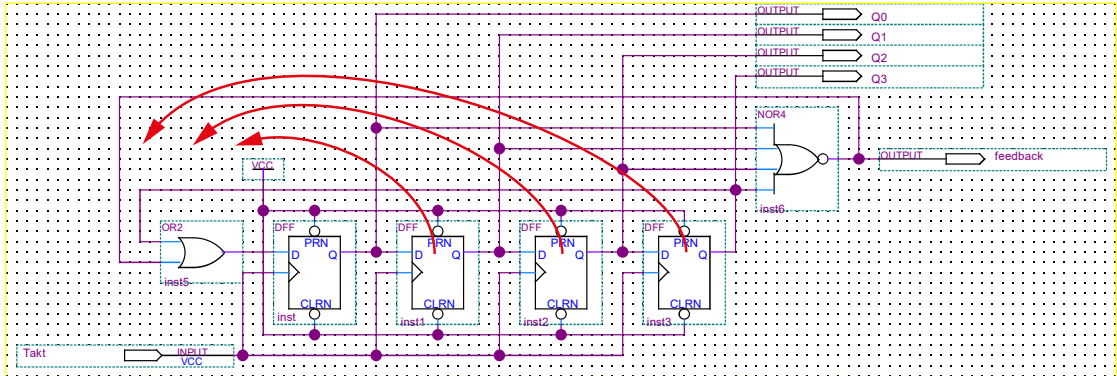


Was hier noch fehlt, ist der Richtungswechsel. D.h. diese Schaltung läuft nur in eine Richtung: vorwärts (CW). Wenn wir die Ausgänge vertauschen, können wir auch eine Rückwärtsbewegung (CCW) realisieren. Eine umschaltbare Variante wird leider aufwändiger.

Jetzt leiten wir mal einen umschaltbaren Sequenzer her:

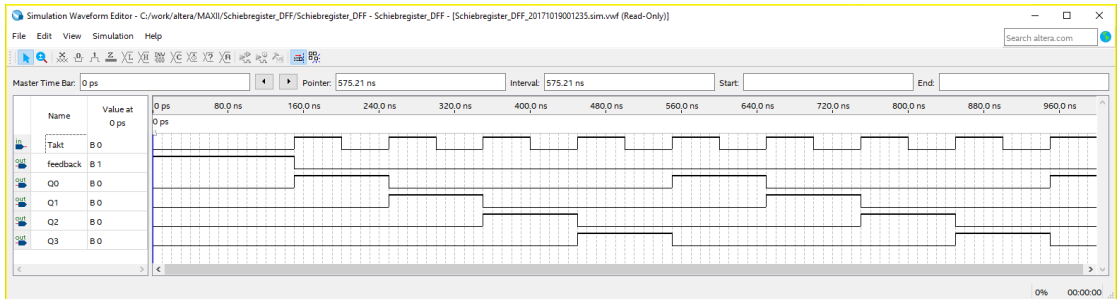
### 7.4.3.3 Herleitung Rückwärts-Schieberegisters mit DD-FF

Zum besseren Verständnis gehen wir zuerst von einem klassischen Schieberegister (Mit der Einprägung der ersten EINS) aus  
 → Sequenzielle Reihenfolge Q0-Q1-Q2-Q3.



7

#### Simulation: Vorwärts

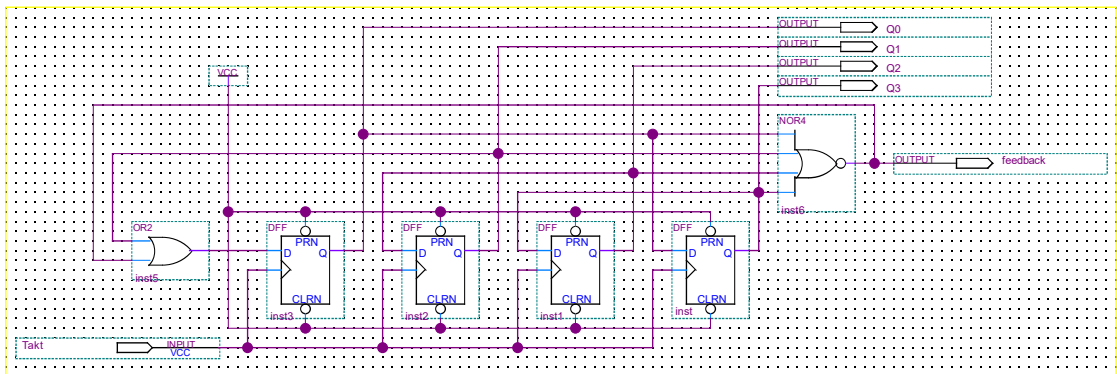


Jetzt vertauschen wir die Reihenfolge der Schieberegisterstufen:

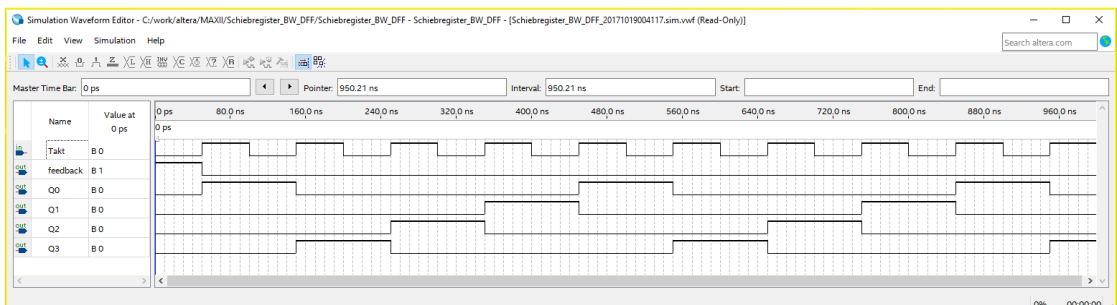
- Wir schieben «inst1» auf die linke Seite von «inst».
- Wir schieben «inst2» auf die linke Seite von «inst1».
- Wir schieben «inst3» auf die linke Seite von «inst2».

Diesen eigentlich unnötigen Aufwand betreiben wir um die Vorwärts- und die Rückwärts-Schaltungen nachher besser kombinieren zu können.

Ressourcen  
 Logic elements:  
 4/1270  
 Total pins:  
 6/212



#### Simulation



In der Simulation sehen wir, dass es funktioniert. Also wenn wir die beiden Schaltungen (Vorwärts und Rückwärts) «verheiraten» können, sind wir am Ziel.

### 7.4.3.5 Herleitung eines Vor- und Rückwärts-Schieberegisters mit D-FF

Wir haben bereits ein Vorwärts- und ein Rückwärtsschieberegister mit D-FF aufgebaut. Einmal leiten wir das Ausgangssignal Q der D-FF nach rechts, und einmal nach links. Wenn wir jetzt in beide Richtungen schieben wollen, müssen wir eine Schaltung haben, welche ein Signal in zwei Richtungen weiterleiten kann. Das machen wir mit einem 1-to-2 Demultiplexer. Ein 1-to-2 Demultiplexer ist sehr einfach aufzubauen. Mit 2 AND-Gattern ist das erledigt. Damit das Schema übersichtlicher bleibt, erzeugen wir dafür einen Funktionsblock «DMUX\_1T2».

Arbeits-schritt	Beschreibung
<p><b>1</b></p>	<p><b>Schema der Demultiplexers eingeben:</b> Wir habend die Eingangssignale:</p> <ul style="list-style-type: none"> <li>• DIR (DIRection) = Angabe der Richtung: «NULL» = Richtung rechts «EINS» = Richtung links</li> <li>• IN (INput) = Signal Q vom D-Flip-Flop, welche wir weiterleiten wollen</li> </ul> <p>Wir habend die Ausgangssignale:</p> <ul style="list-style-type: none"> <li>• LS (Left Side) = Weitergabe Richtung links (rückwärts)</li> <li>• RS (Right Side) = Weitergabe Richtung rechts (vorwärts)</li> </ul> <div style="display: flex; justify-content: space-between;"> <div data-bbox="300 936 1018 1160"> <p>Date: December 29, 2017      DEMUX_1T2.bdf      Project: DEMUX_1T2</p> </div> <div data-bbox="1034 936 1257 1160"> <p><b>DEMUX_1T2</b></p> <p>inst</p> </div> </div>
<p><b>2</b></p>	<p><b>Das umschaltbare Schieberegister:</b> Jetzt werden wir breit. Die Funktionsblöcke nehmen doch viel Platz ein und weil die Eingänge der D-FF aus 2 Richtungen kommen, benötigen wir an den D-Eingängen noch je ein OR-Gatter.</p>
<p><b>3</b></p>	<p><b>Simulation</b></p> <p>Funktioniert einwandfrei!</p>

7

Ressourcen  
Logic elements:  
2/1270  
Total pins:  
4/212

Ressourcen  
Logic elements:  
5/1270  
Total pins:  
7/212

Wir haben uns im Punkt 2 der Tabelle schon das Stichwort für eine Vereinfachung gegeben. Betrachten wir die Umschaltung mal von der anderen Seite. Nicht der Ausgang Q wird in zwei Richtungen geschaltet, sondern der Eingang kann aus zwei Richtungen kommen und wird deshalb geschaltet. Das lösen wir mit dem «Gegenteil» eines Demultiplexers, einem Multiplexer.

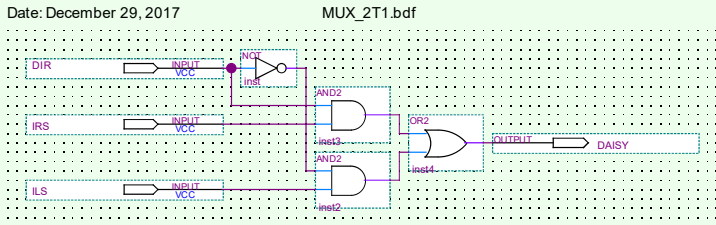
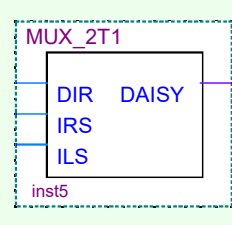
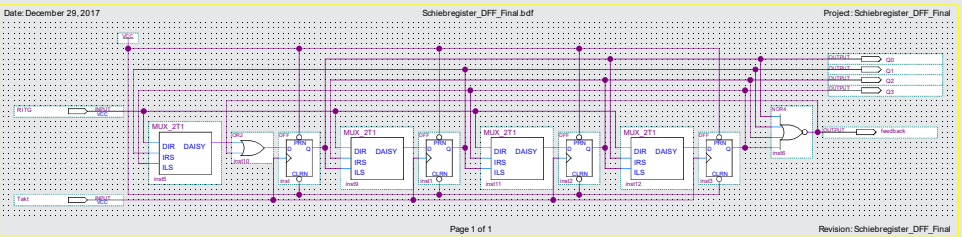
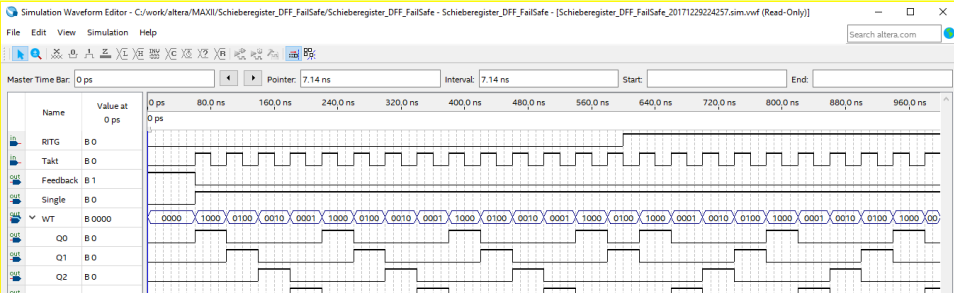


Auch hier erzeugen wir einen Funktionsblock für den 2-to-1 Multiplexer.

Ressourcen  
Logic elements:  
1/1270  
Total pins:  
4/212

7

Ressourcen  
Logic elements:  
5/1270  
Total pins:  
7/212

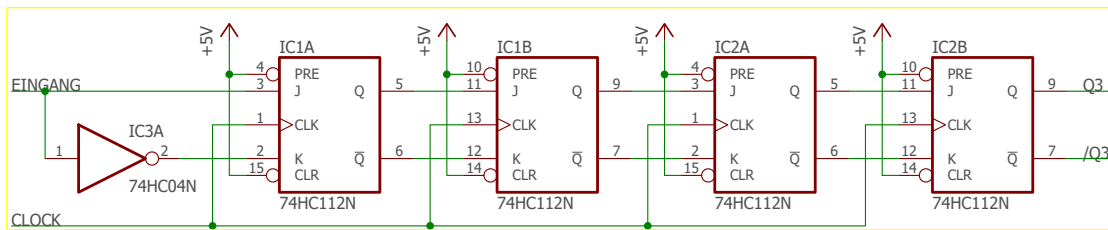
Arbeits-schritt	Beschreibung
1	<p><b>Schema der Multiplexers eingeben:</b> Wir habend die Eingangssignale:</p> <ul style="list-style-type: none"> <li>• DIR (DIRection) = Angabe der Richtung: «NULL» = Richtung rechts «EINS» = Richtung links</li> <li>• IRS (Input Right Side) = Eingangssignal kommt von rechts (rückwärts)</li> <li>• ILS (Inpit Right Side) = Eingangssignal kommt von links (vorwärts)</li> <li>• DAISY (Von «daisy chain» = Gänseblümchenkette) = Ausgang</li> </ul> <p>Date: December 29, 2017 MUX_2T1.bdf</p>  
2	<p><b>Das umschaltbare Schieberegister mit Multiplexer:</b> Das ist doch schon viel übersichtlicher als mit den Demultiplexern.</p> 
3	<p><b>Simulation</b></p>  <p>Funktioniert einwandfrei! Wenn wir jetzt die Ausgänge Q1=D, Q2=A, Q3=C und Q4=B zuweisen, haben wir schon einen Wave-Drive-Sequenzer mit Vorwärts- und Rückwärtswahl.</p>

Wenn man die beiden Schemata genau betrachtet, bemerkt man, dass wir genau die gleichen Gatter in der genau gleichen Menge benutzt haben. Die Lösung mit den Multiplexern ist aber «schöner».

Ich habe diese Funktion noch mit meinen sonst bevorzugten JK-Flip-Flops aufgebaut, das Ergebnis ist aber sehr, sehr viel komplizierter. Ausserdem gilt hier mein meistbenutztes Argument gegen die D-Flip-Flops nichts: diese Schaltung ist synchron. Was wieder einmal beweist, dass man flexibel im Denken und in handeln sein muss um die besten Lösungen zu finden.



Hier zur Veranschaulichung ein normales Vorwärts-Schieberegister (nach rechts) mit JK-Flip-Flops.



Hier müssen wir nicht nur die Signale von Q nach D in 2 Richtungen schicken, sondern die Signale von Q-nach J und auch von Q̄ nach K.

**Fail-Safe!**

Wir können davon ausgehen, dass immer nur ein Flip-Flop aktiv ist (Hier eine EINS an nur einem Ausgang Q), es gibt eigentlich keine Möglichkeit das 2 Flip-Flops gleichzeitig aktiv werden. Beim Aufstarten des Bausteines haben wir aber keine Garantie, welche Zustände die Flip-Flops einnehmen. Deshalb benutzen wir die asynchronen Set- oder Reset-Eingänge der Flips-Flops und erzeugen kurz nach dem Aufstarten einen kurzen Puls der die ganze Schaltung in den «Anfangszustand» bringt. Wir haben das jetzt hier nicht und gehen auch davon aus, dass ein solcher Zustand auftreten könnte. Dafür erzeugen wir eine neue Zusatzschaltung welche überwacht, dass immer nur ein Ausgang Q aktiv ist. Jetzt sind wir wieder bei der Frage ob das mit mehreren EXOR lösbar ist. Doch zuerst schauen wir uns die Funktion in einer Wahrheitstabelle an:

Test auf mehr als 1 Eingang = EINS						
Fall	Eingänge				Ausgang	Verknüpfungen
	E3	E2	E1	E0	S	
0	0	0	0	0	0	
1	0	0	0	1	1	$F1 = \overline{E3} \cdot \overline{E2} \cdot \overline{E1} \cdot E0$
2	0	0	1	0	1	$F2 = \overline{E3} \cdot \overline{E2} \cdot E1 \cdot \overline{E0}$
3	0	0	1	1	0	
4	0	1	0	0	1	$F4 = \overline{E3} \cdot E2 \cdot \overline{E1} \cdot \overline{E0}$
5	0	1	0	1	0	
6	0	1	1	0	0	
7	0	1	1	1	0	
8	1	0	0	0	1	$F8 = E3 \cdot \overline{E2} \cdot \overline{E1} \cdot \overline{E0}$
9	1	0	0	1	0	
10	1	0	1	0	0	
11	1	0	1	1	0	
12	1	1	0	0	0	
13	1	1	0	1	0	
14	1	1	1	0	0	
15	1	1	1	1	0	

Tabelle 7.4

Die vier Verknüpfungen können wir auf ein 4-fach ODER-Gatter führen und haben dann die gewünschte Funktion:

$$S = F1 + F2 + F4 + F8$$

Dafür erzeugen wir gleich mal eine Testschaltung und testen danach auch noch die Lösung mit EXOR-Gattern.

Ressourcen  
Logic elements:  
1/1270  
Total pins:  
5/212

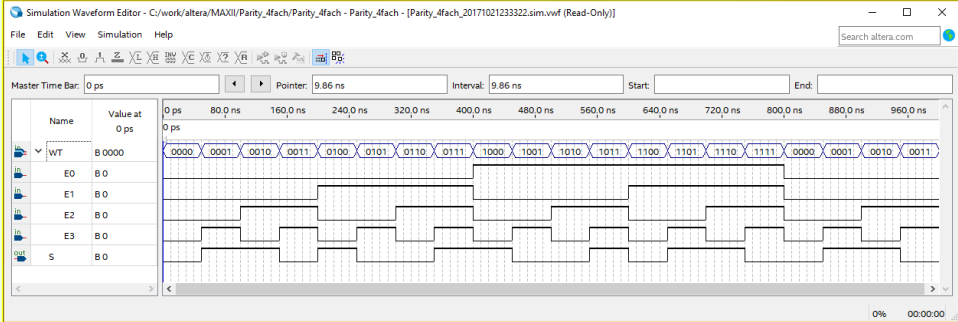
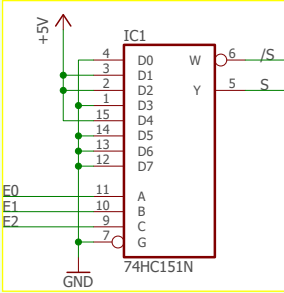
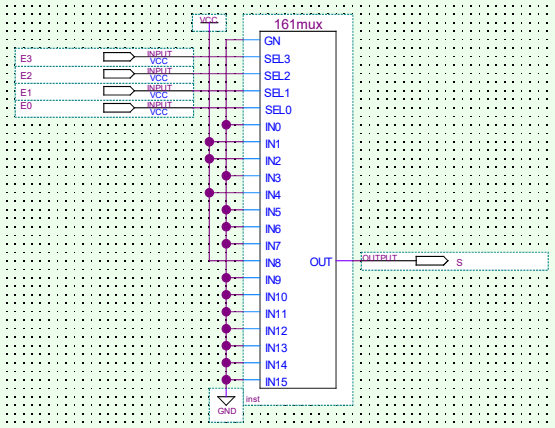
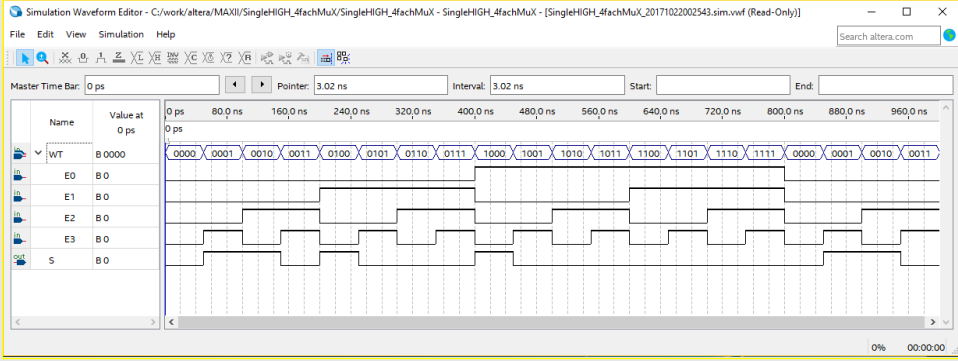
7

<b>Arbeits-schritt</b>	<b>Beschreibung</b>
<b>1a</b>	<p><b>Schema gemäss der Wahrheitstabelle:</b></p> <ul style="list-style-type: none"> <li>• Zuerst invertieren wir die Eingangssignale, dann haben wir alle 8 nachher benötigten Signale zur Verfügung.</li> <li>• Jetzt geben wir die Eingangssignale und die invertierten Eingangssignale stur, gemäss den vorangehend ermittelten Verkopplungen and vier 4-fach AND-Gatter</li> <li>• Alle 4 Ausgänge werden noch verODERT und das Schema ist komplett.</li> </ul> <div style="border: 1px solid black; padding: 5px;"> <p>Date: December 29, 2017      SingleHIGH.bdf      Project: SingleHIGH</p> <p style="text-align: center;">Page 1 of 1      Revision: SingleHIGH</p> </div> <p>Will man das konventionell aufbauen, benötigt man 4 ICs (1 mal 74HC04, 2 mal 74HC21 und 1 mal 74HC25). 4 der 6 Inverter des 74HC04 benötigen wir für die Invertierung der Eingänge und einen der Inverter am Ende des 74HC25 um aus dem NOR-Gatter ein OR-Gatter zu machen.</p>

<b>1b</b>	<p><b>Simulation der Schaltung «SingleHIGH» (nur eine Eingang EINS):</b></p> <div style="border: 1px solid black; padding: 5px;"> </div> <p>Genau das was wir brauchen, je nach Weiterverarbeitung muss das Ausgangssignal noch invertiert werden.</p>
-----------	--

Ressourcen  
Logic elements:  
1/1270  
Total pins:  
5/212

<b>2a</b>	<p><b>Jetzt testen wir den Aufbau mit EXOR-Gattern:</b> Das ist doch schon viel übersichtlicher als mit den Demultiplexern.</p> <div style="border: 1px solid black; padding: 5px;"> </div> <p>Will man das konventionell aufbauen, benötigt man 1 IC (1 mal 74HC86).</p>
-----------	---

Arbeits-schritt	Beschreibung
<p><b>2b</b></p>	<p>Wie schon in Kapitel 7.4.1.1 gesagt, funktioniert die Verschaltung von EXOR-Gattern als Parity-Checker (Test auf ungerade oder gerade Eingänge)</p>  <p>Die Simulation bestätigt das. Egal wie wir die XOR-Gatter kombinieren, wir erhalten immer dieselbe einen Parity-Checker.</p> <p>→ Für diese Anwendung unbrauchbar</p>
<p><b>3a</b></p>	<p><b>Die raffinierte Lösung</b></p> <p>Müssen wir diese Funktion mit ICs aufbauen gibt es für 3 Eingänge eine elegante Lösung mit einem 8-to-1 Multiplexer (1 IC 74HC151). Da wir aber 4 Eingänge haben, geht dieses IC nicht. Wir nehmen diesen Ansatz und finden in der Bibliothek von Quartus einen 16-to-1 Multiplexer. Mit dem können wir unsere gewünschte Funktion realisieren.</p>
<p><b>3b</b></p>	<div style="display: flex; justify-content: space-between;"> <div style="width: 48%;"> <p><b>Die raffinierte Lösung für 3 Eingänge:</b></p>  <p>Je nach Eingangskombination werden die Eingänge D0...D7 an den Ausgang S geschaltet. Wir müssen jetzt nur noch dafür sorgen, dass bei den Kombinationen «001», «010» und «001» eine EINS geschaltet wird und sonst immer eine NULL.</p> </div> <div style="width: 48%;"> <p><b>Die raffinierte Lösung für 4 Eingänge</b></p> <p>Wir nehmen diesen Ansatz und finden in der Bibliothek von Quartus einen 16-to-1 Multiplexer. Mit dem können wir unsere gewünschte Funktion realisieren;</p>  </div> </div>
<p><b>3c</b></p>	<p>Simulation der Schaltung «SingleHIGH_fachMUX» (nur eine Eingang EINS):</p>  <p>Genau das was wir brauchen, je nach Weiterverarbeitung muss das Ausgangssignal noch invertiert werden.</p>

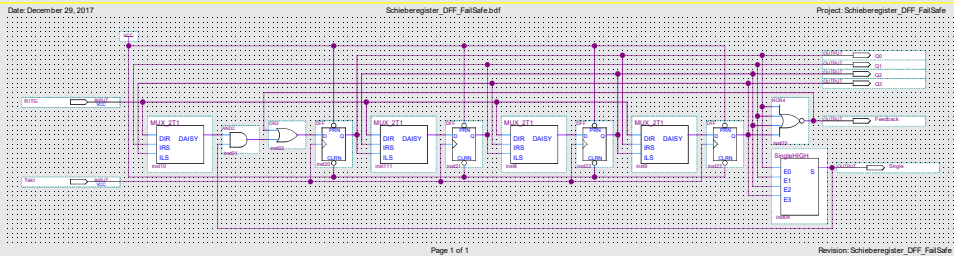
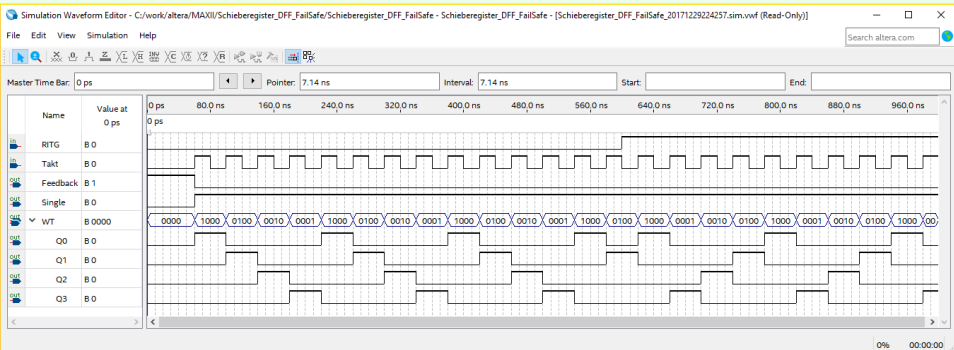
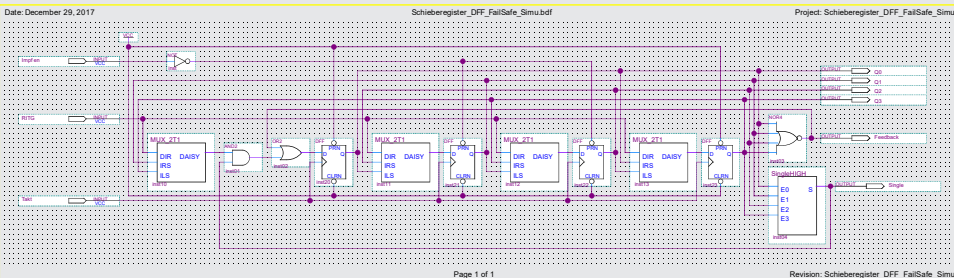
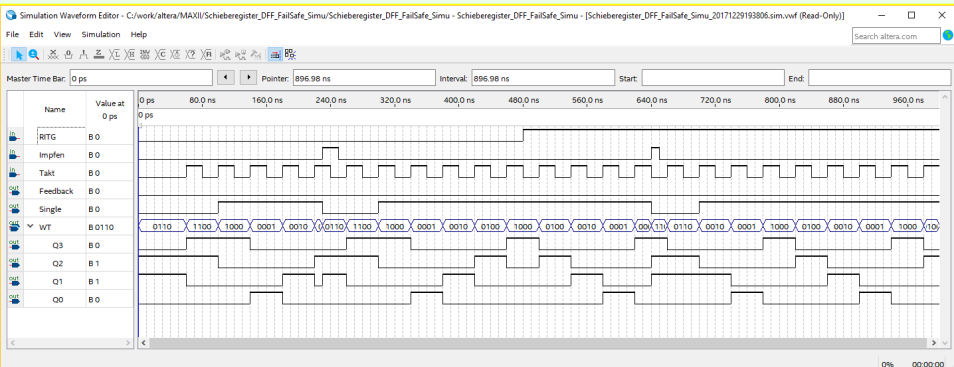
Ressourcen  
 Logic elements: 1/1270  
 Total pins: 5/212

Bei unserem MAX II Baustein benötigen beide Schaltungen zu wenig Gatter (Total logic elements 1 / 1,270 ( < 1 % )) um sie vergleichen zu können. Soll die Funktion aber in ein PAL oder GAL programmiert werden, wird die Wahl auf die erste Lösung

treffen. Der Multiplexer benötigt sicher mehr Gatter.

Ressourcen  
Logic elements:  
6/1270  
Total pins:  
8/212

7

Arbeits-schritt	Beschreibung
<p>1</p>	<p><b>Eingabe des Schemas:</b> Zuerst erzeugen wir aus dem Schema «singleHIGH» einen Funktionsblock, dann bauen wir ihn parallel zum «NULL-Detektor» in das Schema ein.</p> <p>Den Ausgang «Single» könnten wir auf die RESET-Eingänge der D-Flip-Flop führen und alle Zustände auf NULL setzen. Wir lassen die Störung aber «rauswachsen» indem wir den ersten Eingang so lange auf NULL setzen, bis nur noch ein Ausgang EINS ist.</p> 
<p>2</p>	<p><b>Simulation</b></p> 
<p>3</p>	<p><b>Eingabe des Test-Schemas:</b> Für Testzwecke, «impfe» ich ein Flip-Flop indem ich dessen SET-Eingang für ein. zwei Takte aktiviere. Danach wird so lange eine NULL ins Schieberegister geschoben, bis nur noch ein Ausgang EINS ist.</p> 
<p>2</p>	<p><b>Simulation</b></p> 

Ressourcen  
Logic elements:  
6/1270  
Total pins:  
7/212

### 7.4.3 Vollschritt-Sequenzer mit Zähler

Aus der Wahrheitstabelle sehen wir, dass wir immer von 0 auf 3 zählen und dann wieder bei 0 beginnen. Das können wir mit einem binären Zähler erzeugen. Die binären Ausgänge des Zählers, werden wir so verknüpfen, dass sie die Signale A, B, C und D bilden.

Vollschritt mit 2-Bit-Zähler							
Schritt	Zählwert		Ausgänge				Verknüpfungen
	dezimal	binär	A	B	C	D	
	0	00	0	1	0	1	$A = Q0 \overline{Q1} + \overline{Q0} Q1$
	1	01	1	0	0	1	$B = \overline{Q0} \overline{Q1} + Q0 Q1$
	2	10	0	1	1	0	$C = \overline{Q0} Q1 + Q0 Q1$
	3	11	1	0	0	0	$D = \overline{Q0} \overline{Q1} + Q0 \overline{Q1}$
	4	00	0	1	0	1	
	5	01	1	0	0	1	
	6	10	0	1	1	0	
	7	11	1	0	0	0	

Tabelle 7.4

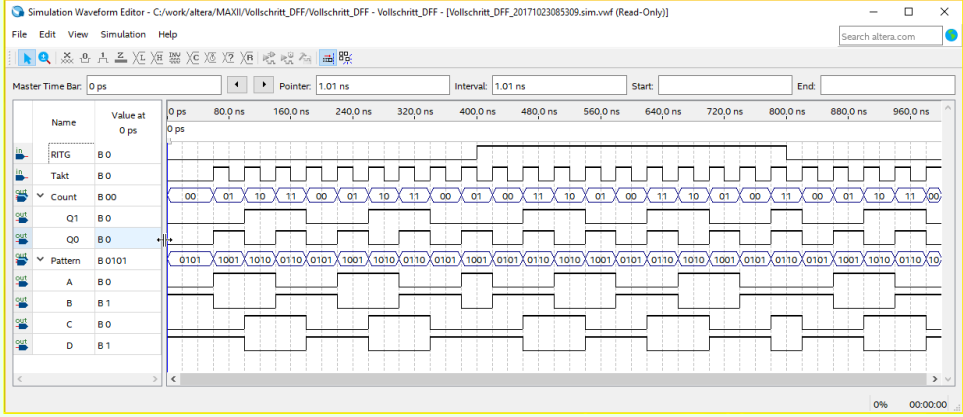
Den Wave-Drive-Sequenzer haben wir ja zuerst mit einem Zähler aus der Bibliothek von Quartus und dann mit einem Schieberegister aufgebaut mit D-Flip-Flops aufgebaut. Will man vollständig unabhängig vom Hersteller sein programmiert man die Funktionen mit VHDL. Programiert man hingegen mit mittels Schemaeingabe und will weniger abhängig vom Hersteller sein, baut man alle komplexeren Funktionen selber auf (Keine Megafunktionen usw.). Also erzeugen wir den Zähler diesmal mit D-Flip-Flops.

Arbeits-schritt	Beschreibung
1	<p><b>Schema des 2-Bit-Zählers mit D-FF als Funktionsblock:</b> Auch mit D-Flip-Flops kann man synchrone Up/Down-Zähler bauen. Hier ist der Multiplexer mit Gattern aufgebaut:</p>
2	<p><b>Das Schema des Vollschritt-Sequenzer gemäss Wahrheitstabelle:</b></p>

Ressourcen  
Logic elements:  
2/1270  
Total pins:  
5/212

Ressourcen  
Logic elements:  
3/1270  
Total pins:  
8/212

7

Arbeits-schritt	Beschreibung
<b>3</b>	<p><b>Simulation</b></p>  <p style="text-align: center;">Funktioniert einwandfrei!</p>

Jetzt schauen wir ob wir da noch etwas vereinfachen können:

Verknüpfung A:  $= Q0 \cdot \overline{Q1} + \overline{Q0} \cdot Q1 = Q0 \oplus Q1$

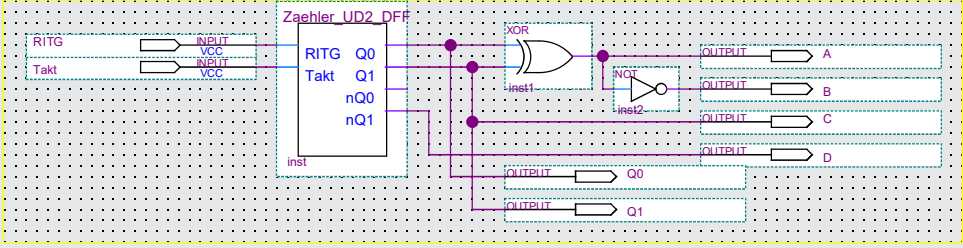
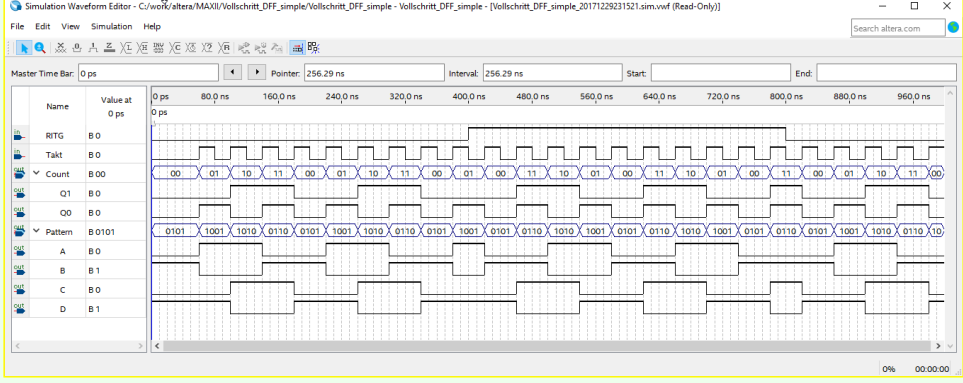
Verknüpfung B:  $= \overline{Q0} \cdot \overline{Q1} + Q0 \cdot Q1 = \overline{Q0 \oplus Q1}$

Verknüpfung C:  $= \overline{Q0} \cdot Q1 + Q0 \cdot \overline{Q1} = Q1 \cdot (Q0 + \overline{Q0}) = Q1 \cdot 1 = Q1$

Verknüpfung D:  $= \overline{Q0} \cdot \overline{Q1} + Q0 \cdot \overline{Q1} = \overline{Q1} \cdot (Q0 + \overline{Q0}) = \overline{Q1} \cdot 1 = \overline{Q1}$

Damit können wir noch ein paar Gatter einsparen:

Ressourcen  
Logic elements:  
3/1270  
Total pins:  
8/212

Arbeits-schritt	Beschreibung
<b>1</b>	<p><b>Das Schema gemäss vereinfachter Wahrheitstabelle:</b></p> 
<b>2</b>	<p><b>Simulation</b></p>  <p style="text-align: center;">Diese Simulation ergibt die selben Resultate wie die vorangehende ohne Vereinfachung.</p>

### 7.4.4 Halbschritt-Sequenzer mit Zähler

Aus der Wahrheitstabelle sehen wir, dass wir immer von 0 auf 7 zählen und dann wieder bei 0 beginnen. Das können wir mit einem binären Zähler erzeugen. Analog zum Vollschrittzähler, verknüpfen wir die binären Ausgänge des Zählers so, dass sie die Signale A, B, C und D bilden.

Vollschritt mit 2-Bit-Zähler									
Schritt	Zählwert			Ausgänge					
	dezimal	binär			A	B	C	D	
		Q2	Q1	Q0					
1001 → 3 → 1000 → 4 → 1010 → 5	1	0	0	0	0	1	0	1	
0001 → 2	2	1	0	0	1	0	0	1	
0101 → 1	3	2	0	1	0	1	0	1	
	4	3	0	1	1	1	0	0	
	5	4	1	0	0	1	0	1	0
	6	5	1	0	1	0	0	1	0
	7	6	1	1	0	0	1	1	0
0100 → 8	8	7	1	1	1	0	1	0	0
	1	0	0	0	0	0	1	0	1
	2	1	0	0	1	0	0	0	1
..	..	..	..	..	..	..	..	..	..

Tabelle 7.4

Die notwendigen Verknüpfungen lesen wir aus der Wahrheitstabelle raus:

Verknüpfung A:  $= \overline{Q0} \cdot Q1 \cdot \overline{Q2} + \overline{Q0} \cdot Q1 \cdot Q2 + Q0 \cdot \overline{Q1} \cdot \overline{Q2}$

Verknüpfung B:  $= \overline{Q0} \cdot \overline{Q1} \cdot \overline{Q2} + Q0 \cdot Q1 \cdot \overline{Q2} + Q0 \cdot Q1 \cdot Q2$

Verknüpfung C:  $= Q0 \cdot \overline{Q1} \cdot \overline{Q2} + Q0 \cdot \overline{Q1} \cdot Q2 + Q0 \cdot Q1 \cdot \overline{Q2}$

Verknüpfung D:  $= \overline{Q0} \cdot \overline{Q1} \cdot \overline{Q2} + \overline{Q0} \cdot \overline{Q1} \cdot Q2 + \overline{Q0} \cdot Q1 \cdot \overline{Q2}$

Diese Verknüpfungen lösen wir genau so wie aus der Wahrheitstabelle ausgelesen.

Wir werden auch nichts mehr vereinfachen, weil wir:

A nicht an Gattern sparen müssen

B es nicht wirklich viel einfacher wird

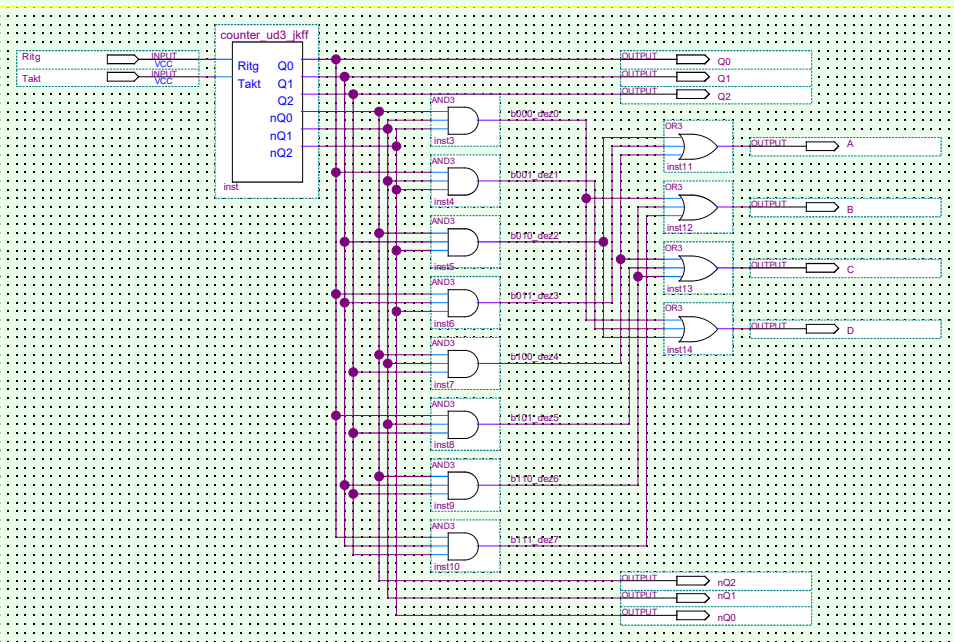
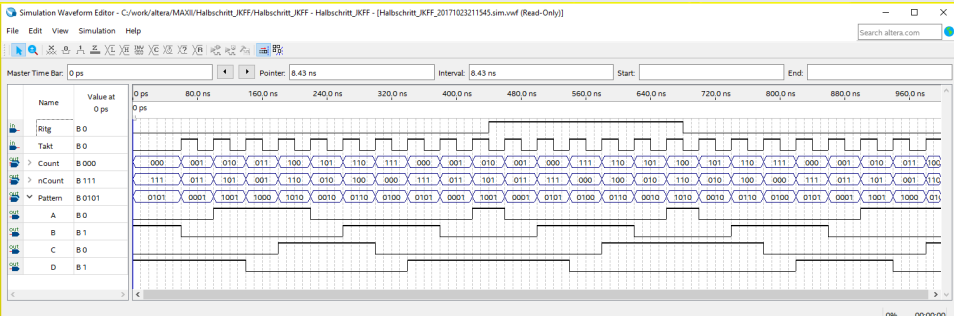
Diesmal bauen wir einen Up/Down-Zähler aus JK-Flip-Flops.

Arbeits-schritt	Beschreibung
1	<p><b>Schema des 3-Bit-Zählers mit JK-FF als Funktionsblock:</b></p>

Ressourcen  
 Logic elements:  
 3/1270  
 Total pins:  
 8/212

Ressourcen  
Logic elements:  
7/1270  
Total pins:  
12/212

7

Arbeits-schritt	Beschreibung
<p><b>2</b></p>	<p><b>Das Schema gemäss Wahrheitstabelle:</b></p> 
<p><b>3</b></p>	<p><b>Simulation</b></p> 

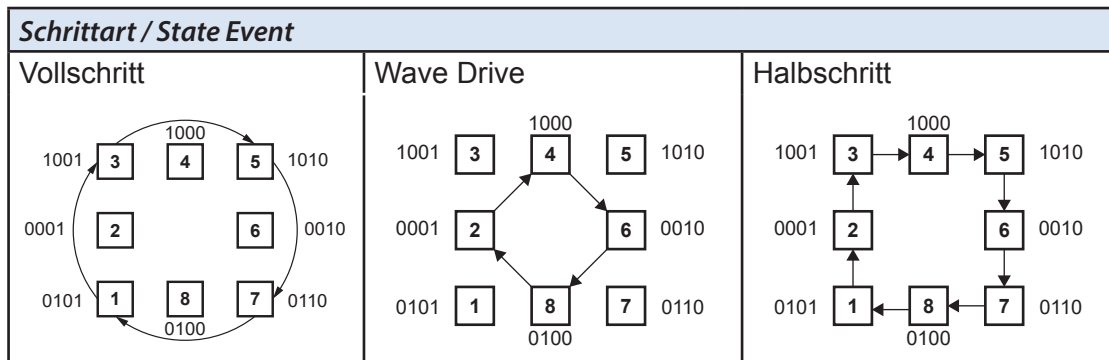
Jetzt haben wir für alle 3 Schritarten einen Sequenzer gebaut. Als nächstes werden wir einen umschaltbaren Sequenzer aufbauen.

Wir haben für jeden Sequenzer einen anderen Zähler benutzt. Mit D-FF und mit JK-FF kann man relativ einfach Up- oder Down-Zähler aufbauen. Umschaltbare Zähler werden aber recht aufwändig. Wir haben kein T-Flip-Flop benutzt (Toggle), beim T-FF wechselt der Ausgangszustand bei jedem Clock, solange der Eingang T aktiv ist. Sobald der Eingang T inaktiv ist, passiert nicht mehr. Genau genommen ist das T-FF ein JK-FF, bei dem die Eingänge J und K verbunden sind und den Namen T erhalten.



### 7.4.5 Universeller-Sequenzer mit Addierwerk

Beim jetzigen Projekt benötigen wir einen Zähler der im Halbschrittmodus bei jedem Takt um eins hochzählt und im Wave-Drive oder im Vollschrittmodus um 2 hochzählt. Das können wir mit normalen Zähler nicht machen. Dafür werden wir ein Addierwerk benutzen, bei dem wir mit jedem Takt 1 oder 2 addieren. Um die Drehrichtung zu ändern, müssen wir aber auch rückwärts zählen können. Das bedeutet, unser Addierwerk auch subtrahieren können.



- Beim Umschalten von Vollschritt auf Halbschritt und beim Umschalten von Wave-Drive auf Halbschritt können wir die States belassen wie sie sind.
- Beim Umschalten von Halbschritt auf Vollschritt gibt es zwei Möglichkeiten:
  - Steht der Motor bei Schritt 1, 3, 5 oder 7 können wir die States belassen wie sie sind.
  - Steht der Motor bei Schritt 2, 4, 6 oder 8 müssen wir uns entscheiden in welche Richtung wir beim Umschalten einen Halbschritt einfügen.
- Beim Umschalten von Halbschritt auf Wave Drive gibt es wieder zwei Möglichkeiten:
  - Steht der Motor bei Schritt 1, 3, 5 oder 7 müssen wir uns entscheiden in welche Richtung wir beim Umschalten einen Halbschritt einfügen.
  - Steht der Motor bei Schritt 2, 4, 6 oder 8 können wir die States belassen wie sie sind.

Hier ist unsere Anforderungsliste:

- Wir arbeiten mit 3 Eingängen für die Schrittartauswahl (Vollschritt = Prio 1, Halbschritt = Prio 2 und Wave-Drive = Prio 3)
- Ein Eingang dient zur Steuerung CW/CCW (Im- oder Gegenuhrzeigersinn)
- Wir erzeugen einen Zähler der folgende Additionen ausführen kann: +1, +2, -1 und -2
- Die binären Ausgangssignale des Zählers werden wir genau gleich verknüpfen wie im vorangehenden Halbschritt-Sequenzer

Wir verzichten hier auf eine weitere Analyse, wir entwickeln weder Top-Down, noch Bottom-Up, wir beginnen einfach mal in der Mitte. Wir schauen uns einen Addierer an und entwickeln daraus unsere gewünschte Funktion.

Am Ende wollen wir eine Schaltung die nur aus Logik-Gattern und Flip-Flops aufgebaut ist. Also keine herstellerabhängigen Funktionsblöcke.

### 7.4.5.1 Addierer aus der Bibliothek

Zuerst entnehmen wir der Bibliothek einen 4-Bit-Addierer mit dem unser Werk starten, später werden wir den Addierer noch aus Einzelgattern aufbauen.

Ressourcen  
Logic elements:  
6/1270  
Total pins:  
14/212

7

Arbeits-schritt	Beschreibung
<b>1</b>	<p><b>Schema des Addierers aus der Bibliothek:</b></p>
<b>2</b>	<p><b>Simulation :</b></p> <p>Daraus lesen wir:</p> <ul style="list-style-type: none"> <li>• 0 + 0 = 0</li> <li>• 1 + 1 = 2</li> <li>• 2 + 1 = 3</li> <li>• 3 + 2 = 5</li> <li>• Carry + 5 + 2 = 8</li> <li>• Carry + 6 + 0 = 7</li> <li>• 8 + 12 = 4 + CarryOut(16) = 4 + 16 = 20</li> <li>• 9 + 8 = 1 + CarryOut(16) = 1 + 16 = 17</li> <li>• usw.</li> </ul>

Das funktioniert schon mal. Wie funktioniert jetzt aber das Subtrahieren.

Subtrahend – Minuend = Differenz

Mit einem Addierer können wir auch subtrahieren, indem wir das Zweier-Komplement des Minuenden als Summand2 eingeben.

Dies entspricht dann der Operation:

$$\text{Summand 1} + (- \text{Summand 2}) = \text{Summe}$$

Beispiel 1:  $9 - 5 = 4$

Exponent		$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
Wertigkeit		128	64	32	16	8	4	2	1
Erster Summand		0	0	0	0	1	0	0	1
Invertierter zweiter Summand	+	1	1	1	1	1	0	1	0
Zweierkomplement (Carry in)	+								1
Übertrag (Carry out)	+	1	1	1	1	0	1	1	
Summe	=	0	0	0	0	0	1	0	0

Beispiel 2:  $4 - 1 = 3$

Erster Summand		0	0	0	0	0	1	0	0
Invertierter zweiter Summand	+	1	1	1	1	1	1	1	0
Zweierkomplement (Carry in)	+								1
Übertrag (Carry out)	+	1	1	1	1	1	0	0	
Summe	=	0	0	0	0	0	0	1	1

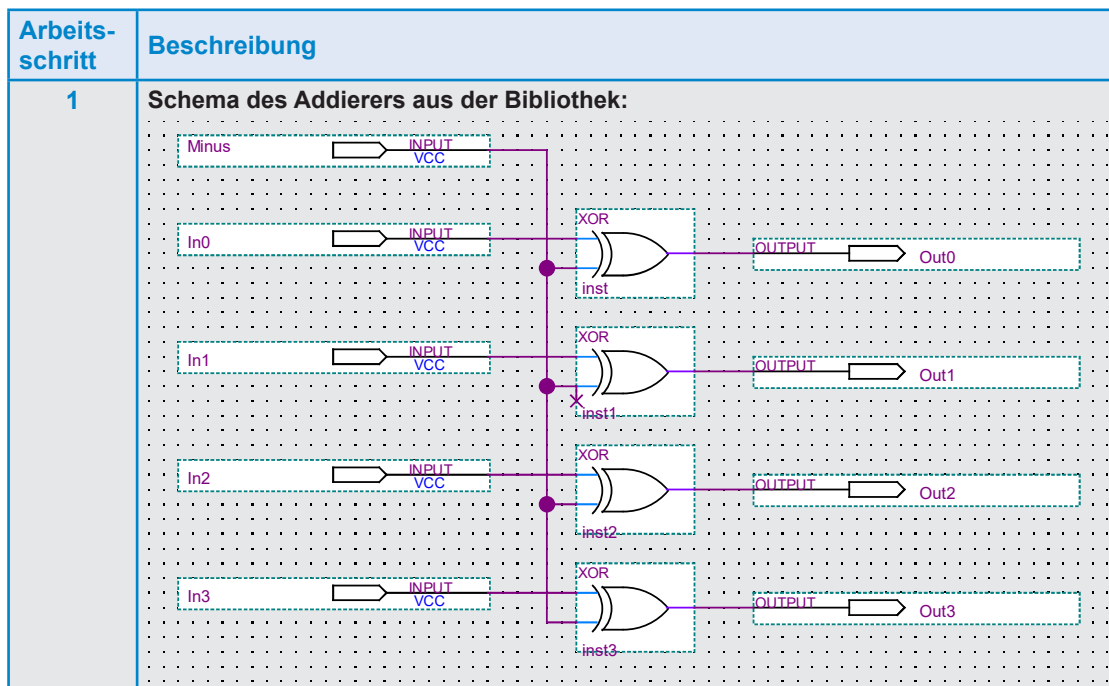
Das wollen wir jetzt mal in unser Schema einbauen:

Dafür benötigen wir aber noch einen steuerbaren Invertierer für unseren zweiten Summanden. Wir schauen die Anforderung für ein Bit in einer Wahrheitstabelle an und erkennen sofort die Lösung:

Gesteuerter Inverter		
Eingänge		Ausgang
Steuereingang	A	nA
0 = addieren	0	0
0 = addieren	1	1
1 = subtrahieren	0	1
1 = subtrahieren	1	0

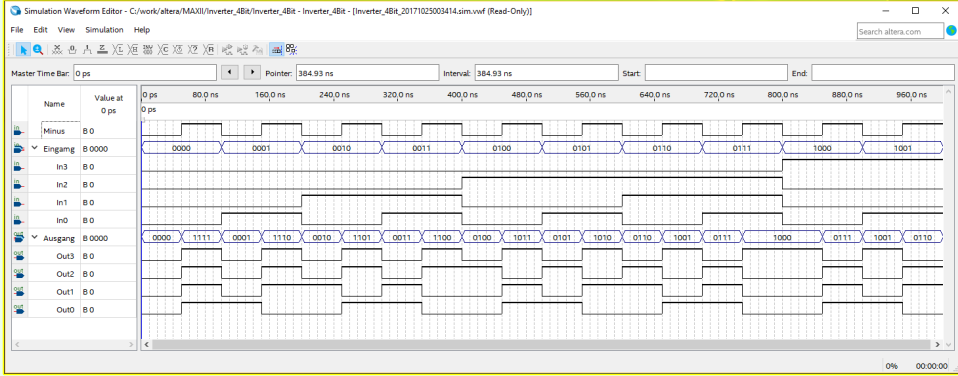
Tabelle 7.6

Das ist wieder einmal unser EXOR. Mit 4 EXOR-Gattern können wir 4 Bits gesteuert invertieren:



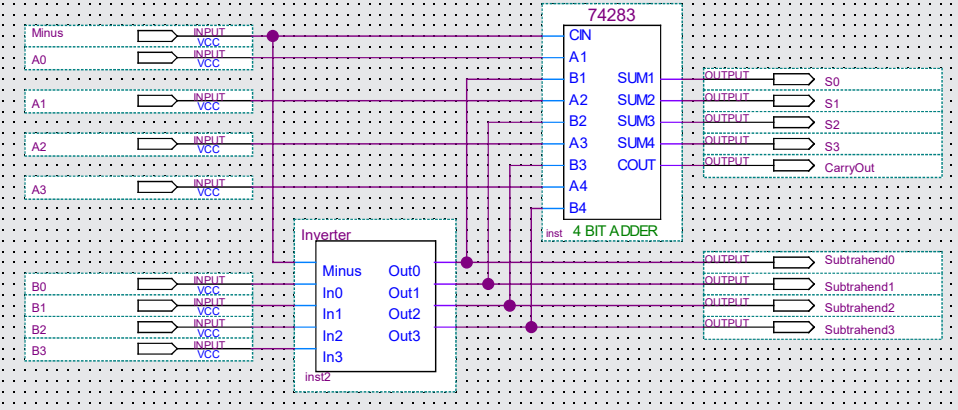
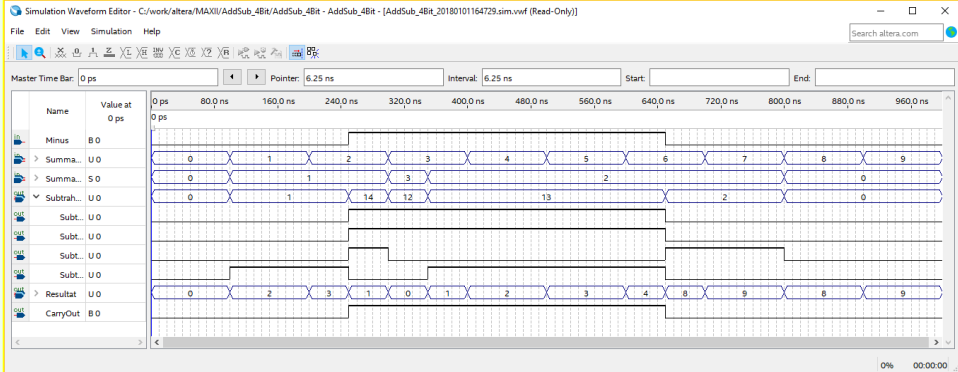
Ressourcen  
 Logic elements:  
 4/1270  
 Total pins:  
 9/212

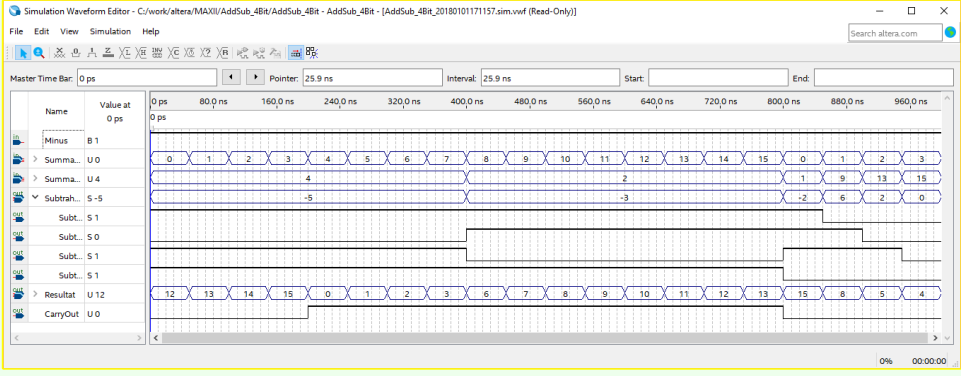
7

Arbeits-schritt	Beschreibung
<b>2</b>	<p><b>Simulation :</b></p>  <p>• Dies entspricht dem Einerkomplement. Das Zweierkomplement, erzeugen wir mit dem Carry des Addierers.</p>

Jetzt erweitern wir den Addierer mit dem Invertierer und haben ein Addierer/Subtrahierer in einem:

Ressourcen  
 Logic elements:  
 10/1270  
 Total pins:  
 18/212

Arbeits-schritt	Beschreibung		
<b>1</b>	<p><b>Schema des Addierers aus der Bibliothek:</b></p> 		
<b>2a</b>	<p><b>Simulation Addition und Subtraktion:</b></p>  <p><b>Berechnungen herauslesen:</b></p> <table border="0" style="width: 100%;"> <tr> <td style="vertical-align: top;"> <ul style="list-style-type: none"> <li>• 0 + 0 = 0</li> <li>• 1 + 0 = 1</li> <li>• 1 + 1 = 2</li> <li>• 2 + 1 = 3</li> <li>• 2 - 1 = 1</li> <li>• 3 - 1 = 2</li> </ul> </td> <td style="vertical-align: top;"> <ul style="list-style-type: none"> <li>• 4 - 3 = 1</li> <li>• 4 - 2 = 2</li> <li>• 5 - 2 = 3</li> <li>• 6 - 2 = 4</li> <li>• 6 + 2 = 8</li> <li>• usw.</li> </ul> </td> </tr> </table>	<ul style="list-style-type: none"> <li>• 0 + 0 = 0</li> <li>• 1 + 0 = 1</li> <li>• 1 + 1 = 2</li> <li>• 2 + 1 = 3</li> <li>• 2 - 1 = 1</li> <li>• 3 - 1 = 2</li> </ul>	<ul style="list-style-type: none"> <li>• 4 - 3 = 1</li> <li>• 4 - 2 = 2</li> <li>• 5 - 2 = 3</li> <li>• 6 - 2 = 4</li> <li>• 6 + 2 = 8</li> <li>• usw.</li> </ul>
<ul style="list-style-type: none"> <li>• 0 + 0 = 0</li> <li>• 1 + 0 = 1</li> <li>• 1 + 1 = 2</li> <li>• 2 + 1 = 3</li> <li>• 2 - 1 = 1</li> <li>• 3 - 1 = 2</li> </ul>	<ul style="list-style-type: none"> <li>• 4 - 3 = 1</li> <li>• 4 - 2 = 2</li> <li>• 5 - 2 = 3</li> <li>• 6 - 2 = 4</li> <li>• 6 + 2 = 8</li> <li>• usw.</li> </ul>		

Arbeits-schritt	Beschreibung
<b>3</b>	<p><b>Simulation nur Subtraktion:</b></p>  <p><b>Berechnungen herauslesen:</b></p> <ul style="list-style-type: none"> <li>• <math>0(+16) - 4 = 12(-16) = -4</math></li> <li>• <math>1(+16) - 4 = 13(-16) = -3</math></li> <li>• <math>2(+16) - 4 = 14(-16) = -2</math></li> <li>• <math>3(+16) - 4 = 15(-16) = -1</math></li> <li>• <math>4 - 4 = 0</math></li> <li>• <math>5 - 4 = 1</math></li> <li>• <math>6 - 4 = 2</math></li> <li>• <math>7 - 4 = 3</math></li> <li>• <math>8 - 2 = 6</math></li> <li>• <math>9 - 2 = 7</math></li> <li>• <math>10 - 2 = 8</math></li> <li>• <math>11 - 2 = 9</math></li> <li>• <math>12 - 2 = 10</math></li> <li>• <math>13 - 2 = 11</math></li> <li>• <math>14 - 2 = 12</math></li> <li>• <math>15 - 2 = 13</math></li> <li>• <math>0(+16) - 2 = 14(-16) = -2</math></li> <li>• <math>1(+16) - 9 = 8(-16) = -8</math></li> <li>• <math>2(+16) - 13 = 5(-16) = -9</math></li> <li>• <math>3(+16) - 15 = 4(-16) = -12</math></li> </ul>

7

Solange das CarryOut = null ist, kann vom Resultat 16 abgezogen werden um die korrekten mathematischen Resultate zu erhalten.

Im Detail am Beispiel:  $1 - 9 = 11$

Erster Summand		0	0	0	1	0	0	0	1
Invertierter zweiter Summand	+	1	1	1	1	0	1	1	0
Übertrag (Carry)	+	1	1	1	0	1	1	1	1
Summe	=	0	0	0	0	1	0	0	0

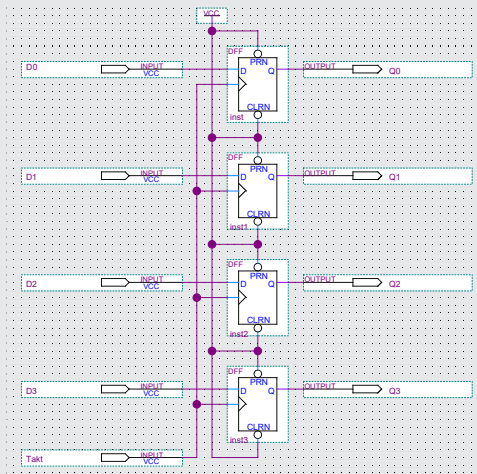
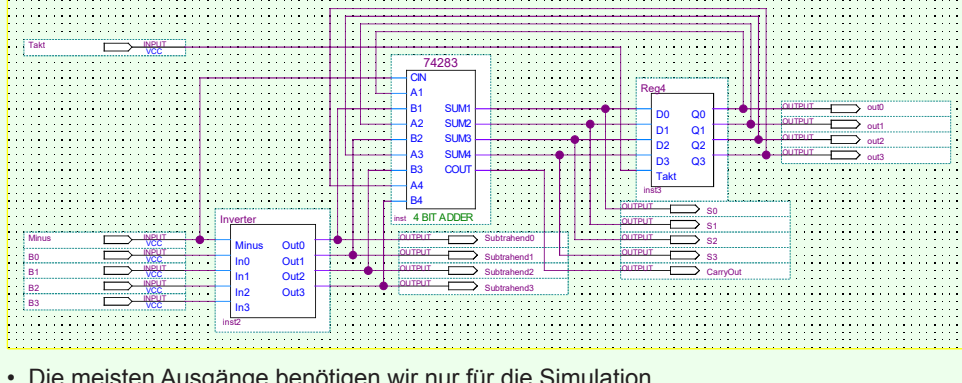
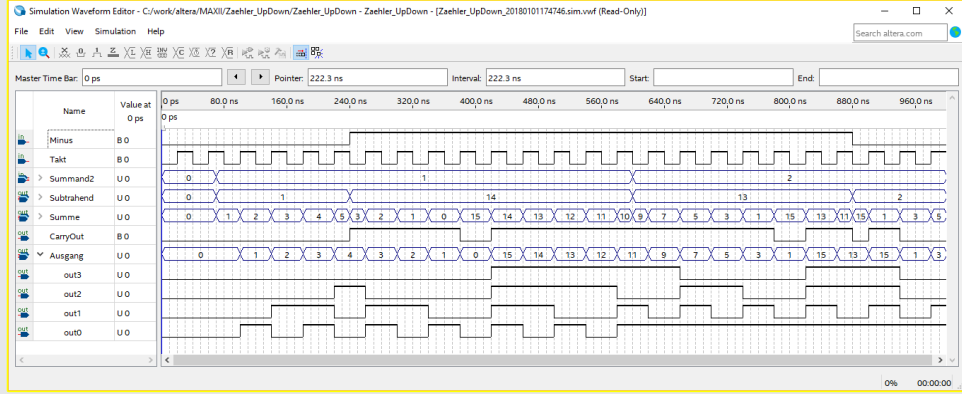
Uns interessieren aber nur die 4 Bits S0...S3. Wir können also jetzt addieren und subtrahieren. Eigentlich wollten wir aber einen Zähler bauen. Dafür erweitern wir die bestehende Schaltung mit einem Register. Über dieses Register führen wir das Resultat der Berechnung zurück an den Summanden1/Minuenden

7.4.5.2 Zähler mit Addierer aus der Bibliothek

Ressourcen  
Logic elements:  
4/1270  
Total pins:  
9/212

7

Ressourcen  
Logic elements:  
14/1270  
Total pins:  
19/212

Arbeits-schritt	Beschreibung	
1	<p><b>Schema des 4-Bit-Latch:</b></p> 	<p>Zuerst erzeugen wir ein 4-Bit-Latch. Das bauen wir mit 4 D-Flip-Flops auf. Der Latch leitet mit jeder positiven Flanke wird das 4-Bit Eingangssignal an den Ausgang. In unserem Fall wird die damit Summe an den Summanden 1 geleitet und daher wieder eine neue Summe = Summand 1(alte Summe) + Summand 2 gebildet.</p>
2	<p><b>Neues Schema Zähler mit rückgeführter Summe:</b></p>  <ul style="list-style-type: none"> <li>• Die meisten Ausgänge benötigen wir nur für die Simulation</li> </ul>	
3	<p><b>Simulation :</b></p>  <ul style="list-style-type: none"> <li>• Das Addieren von 1, 2, -2 und -1 funktionieren einwandfrei.</li> <li>• Mit jeder positiven Flanke erhöht oder erniedrigt sich der Wert des Ausgangs um den Wert am Eingang B</li> </ul>	

Jetzt haben wir den von uns benötigten Zähler. Was wir noch brauchen ist eine Steuerung die für die verschiedenen Schritarten.

### 7.4.5.3 Zuweisung der Schrittarten zum Summanden des Zählers

VS = Vollschritt mit Prio 1; HS = Halbschritt mit Prio 2; WD = Wave Drive mit Prio 3

Prioritätstabelle						
Schrittart				Ausgänge (Summand)		
Richtung (RITG)	WD	VS	HS	B1	B0	Dezimal
0	0	0	0	0	0	0
0	0	0	1	0	1	1
0	0	1	0	1	0	2
0	0	1	1	1	0	2
0	1	0	0	1	0	2
0	1	0	1	0	1	1
0	1	1	0	1	0	2
0	1	1	1	1	0	2
1	0	0	0	0	0	-0
1	0	0	1	0	1	-1
1	0	1	0	1	0	-2
1	0	1	1	1	0	-2
1	1	0	0	1	0	-2
1	1	0	1	0	1	-1
1	1	1	0	1	0	-2
1	1	1	1	1	0	-2

7

Die notwendigen Verknüpfungen lesen wir aus der Wahrheitstabelle heraus:

$$B0 = \overline{RITG} \cdot \overline{WD} \cdot \overline{VS} \cdot HS + \overline{RITG} \cdot WD \cdot \overline{VS} \cdot HS + RITG \cdot \overline{WD} \cdot \overline{VS} \cdot HS + RITG \cdot WD \cdot \overline{VS} \cdot HS$$

$(\overline{VS} \cdot HS)$  ausklammern:

$$B0 = (\overline{VS} \cdot HS) \cdot (\overline{RITG} \cdot \overline{WD} + \overline{RITG} \cdot WD + RITG \cdot \overline{WD} + RITG \cdot WD)$$

RITG und  $\overline{RITG}$  ausklammern:

$$B0 = (\overline{VS} \cdot HS) \cdot (\overline{RITG} \cdot (\overline{WD} + WD) + RITG \cdot (\overline{WD} + WD))$$

Wir wissen, dass  $\overline{WD} + WD = 1$ :

$$B0 = (\overline{VS} \cdot HS) \cdot (\overline{RITG} \cdot 1 + RITG \cdot 1) = (\overline{VS} \cdot HS) \cdot (\overline{RITG} + RITG)$$

Und konsequenterweise dass  $\overline{RITG} + RITG = 1$

$$B0 = (\overline{VS} \cdot HS) \cdot 1 = (\overline{VS} \cdot HS)$$

$$\underline{\underline{B0 = \overline{VS} \cdot HS}}$$

Und jetzt suchen wir die Verknüpfung für B1:

$$B1 = \overline{RITG} \cdot \overline{WD} \cdot VS \cdot \overline{HS} + \overline{RITG} \cdot \overline{WD} \cdot VS \cdot HS + \overline{RITG} \cdot WD \cdot \overline{VS} \cdot \overline{HS} + \overline{RITG} \cdot WD \cdot \overline{VS} \cdot HS + \overline{RITG} \cdot WD \cdot VS \cdot \overline{HS} + \overline{RITG} \cdot WD \cdot VS \cdot HS + RITG \cdot \overline{WD} \cdot VS \cdot \overline{HS} + RITG \cdot \overline{WD} \cdot VS \cdot HS + RITG \cdot WD \cdot \overline{VS} \cdot \overline{HS} + RITG \cdot WD \cdot \overline{VS} \cdot HS + RITG \cdot WD \cdot VS \cdot \overline{HS} + RITG \cdot WD \cdot VS \cdot HS$$

RITG und  $\overline{RITG}$  ausklammern:

$$B1 = \overline{RITG} \cdot (\overline{WD} \cdot VS \cdot \overline{HS} + \overline{WD} \cdot VS \cdot HS + WD \cdot \overline{VS} \cdot \overline{HS} + WD \cdot \overline{VS} \cdot HS + WD \cdot VS \cdot \overline{HS} + WD \cdot VS \cdot HS) + RITG \cdot (\overline{WD} \cdot VS \cdot \overline{HS} + \overline{WD} \cdot VS \cdot HS + WD \cdot \overline{VS} \cdot \overline{HS} + WD \cdot \overline{VS} \cdot HS + WD \cdot VS \cdot \overline{HS} + WD \cdot VS \cdot HS)$$

Die beiden Klammerinhalte sind gleich und können somit zusammengefasst werden:

$$B1 = (\overline{RITG} + RITG) \cdot (\overline{WD} \cdot VS \cdot \overline{HS} + \overline{WD} \cdot VS \cdot HS + WD \cdot \overline{VS} \cdot \overline{HS} + WD \cdot \overline{VS} \cdot HS + WD \cdot VS \cdot \overline{HS} + WD \cdot VS \cdot HS)$$

7

Weglassen von  $\overline{RITG} + RITG = 1$

$$B1 = \overline{WD} \cdot VS \cdot \overline{HS} + \overline{WD} \cdot VS \cdot HS + WD \cdot \overline{VS} \cdot \overline{HS} + WD \cdot \overline{VS} \cdot HS + WD \cdot VS \cdot \overline{HS} + WD \cdot VS \cdot HS$$

VS ausklammern:

$$B1 = VS \cdot (\overline{WD} \cdot \overline{HS} + \overline{WD} \cdot HS + WD \cdot \overline{HS} + WD \cdot HS) + WD \cdot \overline{VS} \cdot \overline{HS}$$

Entweder sieht man, dass  $\overline{WD} \cdot \overline{HS} + \overline{WD} \cdot HS + WD \cdot \overline{HS} + WD \cdot HS = 1$  ist (alle vier kombinatorischen Möglichkeiten von WD und HS kommen darin verODERT vor) oder kürzt schrittweise weiter. Wir kürzen jetzt weiter und klammern als nächstes  $\overline{WD}$  und WD aus:

$$B1 = VS \cdot (\overline{WD} \cdot (\overline{HS} + HS) + WD \cdot (\overline{HS} + HS)) + WD \cdot \overline{VS} \cdot \overline{HS}$$

Jetzt können wir  $\overline{HS} + HS = 1$  wegekürzen:

$$B1 = VS \cdot (\overline{WD} + WD) + WD \cdot \overline{VS} \cdot \overline{HS}$$

Jetzt können wir  $\overline{WD} + WD = 1$  wegekürzen:

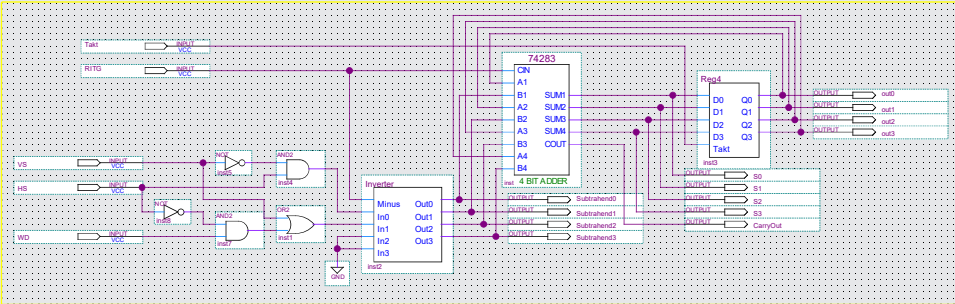
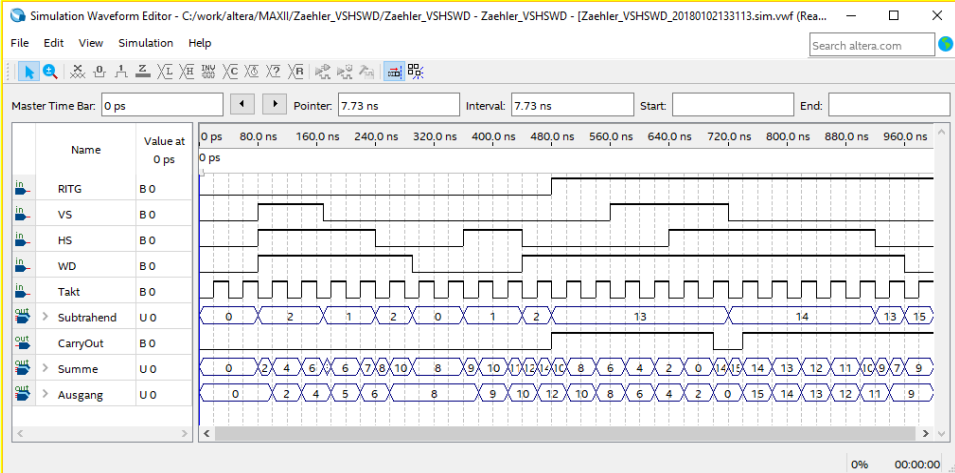
$$B1 = VS + WD \cdot \overline{VS} \cdot \overline{HS}$$

Jetzt meinen wir, dass wir nicht mehr weiter kürzen können. Wir können aber tatsächlich  $\overline{VS}$  weglassen, da die linke Seite des ODER bedeutet, dass die Funktion immer 0 EINS ist wenn VS EINS ist, hat  $\overline{VS}$  auf der rechten Seite des ODER keine Einfluss auf den Ausgang B1:

$$\underline{\underline{B1 = VS + WD \cdot \overline{HS}}}$$



Somit haben wir die beiden notwendigen Funktionen für die Prioritätssteuerung und Selektion des Summanden von +1 oder +2 oder -1 und -2 und können dies in das Schema der Zähler einbauen:

Arbeits-schritt	Beschreibung
1	<p><b>Neues Schema Zähler mit Bestimmung des 2'ten Summanden/Subtrahenden:</b></p>  <ul style="list-style-type: none"> <li>Die meisten Ausgänge benötigen wir nur für die Simulation</li> </ul>
2	<p><b>Simulation :</b></p>  <ul style="list-style-type: none"> <li>Die Prioritätssteuerung funktioniert</li> <li>Das Umschalten von Vorwärts auf Rückwärtszählen (Up/Down) funktioniert auch</li> </ul>

Jetzt ist es an der Zeit den Zähler aus der Bibliothek zu ersetzen. Das ist nicht zwingend notwendig, aber wir haben ja am Anfang entschieden, dass wir die komplette Schaltung mit Logik-Gattern aufbauen und keine Bauteile aus einer speziellen Bibliothek benutzen.

### 7.4.5.3 Das Addierwerk

Addierwerk tönt so hochstehend, eigentlich erzeugen wir eine Schaltung die 1 + 1 mit «CARRY IN» und «CARRY OUT» berechnen kann. Das nennt man dann einen Volladdierer. Den Halbaddierer kennen wir ja bereits bestens aus diesem und dem vorderen Kapitel. Ohne grosse Herleitung schreiben wir die Wahrheitstabelle auf und suchen die notwendigen Verknüpfungen.

#### Definition der Variablen

Erster Summand		<b>A</b>
Zweiter Summand	+	<b>B</b>
Carry in	+	<b>Cin</b>
Carry out	+	<b>Cout</b>
Summe	=	<b>S</b>

**Wahrheitstabelle Halb-Addierer**

Eingang/Ausgang	Carry in	Summand 2	Summand 1	Carry out	Summe
	Cin	B	A	Cout	S
WT-Zeile_1	0	0	0	0	0
WT-Zeile_2	0	0	1	0	1
WT-Zeile_3	0	1	0	0	1
WT-Zeile_4	0	1	1	1	0
WT-Zeile_5	1	0	0	0	1
WT-Zeile_6	1	0	1	1	0
WT-Zeile_7	1	1	0	1	0
WT-Zeile_8	1	1	1	1	1

Die notwendige Verknüpfung für den Ausgang Summe erkennen wir inzwischen schnell. Er ist immer dann EINS, wenn die Parität der Eingänge EINS ist (Ungerade).  
Also  $S = A \oplus B \oplus Cin$

7

Der Ausgang «Carry out» ist immer dann EINS, wenn zwei der Eingänge EINS sind. Dafür haben kennen wir aber keine Verknüpfung. Wir können aber die Bedingungen aus der Wahrheitstabelle herauslesen und die notwendigen Verknüpfungen aufschreiben:

$$Cout = A \cdot B \cdot \overline{Cin} + A \cdot \overline{B} \cdot Cin + \overline{A} \cdot B \cdot Cin + A \cdot B \cdot Cin$$

Neu gruppieren, so dass die A • B beieinander sind und intern umstellen:

$$Cout = \overline{Cin} \cdot A \cdot B + Cin \cdot A \cdot B + Cin \cdot A \cdot \overline{B} + Cin \cdot \overline{A} \cdot B$$

Ausklammern von A • B und Cin:

$$Cout = (\overline{Cin} + Cin) \cdot A \cdot B + Cin \cdot (A \cdot \overline{B} + \overline{A} \cdot B)$$

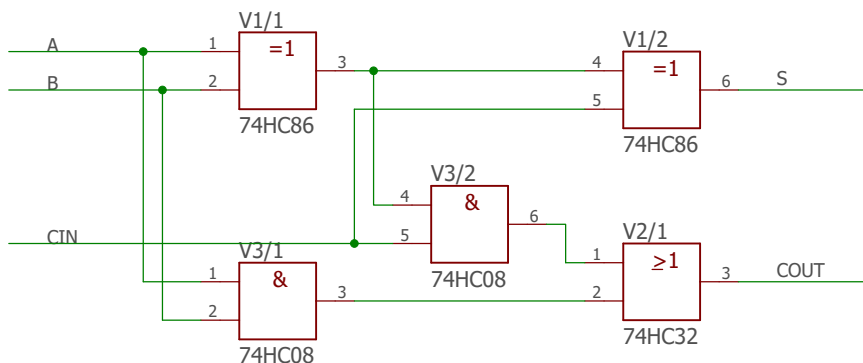
Weglassen eines Terms:  $(\overline{Cin} + Cin) = 1$

$$Cout = A \cdot B + Cin \cdot (A \cdot \overline{B} + \overline{A} \cdot B)$$

EXOR-Verknüpfung erkennen:  $A \cdot \overline{B} + \overline{A} \cdot B = A \oplus B$ ; diese Verknüpfung haben wir schon als Teilresultat bei der Erzeugung der Summe S und können diese abgreifen.

$$\underline{\underline{Cout = A \cdot B + Cin \cdot (A \oplus B)}}$$

Und hier ist das Schema (Diesmal mit IEC-Symbolen):



Der Volladdierer in Quartus:

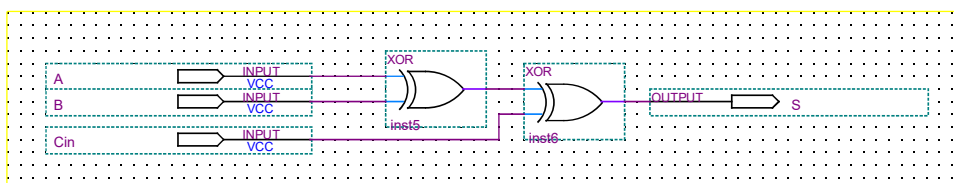
Arbeits-schritt	Beschreibung														
<b>1</b>	<p><b>Zuerst erzeugen wir den Volladdierer:</b></p>														
<b>2</b>	<p><b>Simulation :</b></p> <table border="1" style="margin-top: 10px;"> <thead> <tr> <th>Name</th> <th>Value at 0 ps</th> </tr> </thead> <tbody> <tr> <td>Inputs</td> <td>B 000</td> </tr> <tr> <td>Cin</td> <td>B 0</td> </tr> <tr> <td>B</td> <td>B 0</td> </tr> <tr> <td>A</td> <td>B 0</td> </tr> <tr> <td>S</td> <td>B 0</td> </tr> <tr> <td>Cout</td> <td>B 0</td> </tr> </tbody> </table> <ul style="list-style-type: none"> <li>Die Simulation entspricht der Wahrheitstabelle</li> </ul>	Name	Value at 0 ps	Inputs	B 000	Cin	B 0	B	B 0	A	B 0	S	B 0	Cout	B 0
Name	Value at 0 ps														
Inputs	B 000														
Cin	B 0														
B	B 0														
A	B 0														
S	B 0														
Cout	B 0														

Ressourcen  
 Logic elements:  
 2/1270  
 Total pins:  
 5/212

Effektiv benötigen wir nur einen 3-bit-Addierer und schalten deshalb nur 3 «Volladdierer» hintereinander.

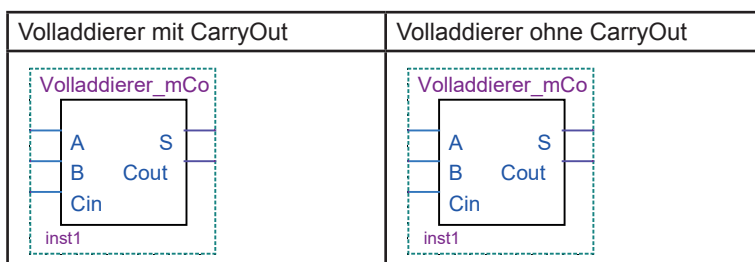
Wir addieren und Subtrahieren ja nur die Werte 1 und 2 (nur 2 Bit für den Summanden 2), da wir beim Subtrahieren aber mit dem Zweierkomplement arbeiten, müssen wir alle 3 Bit für den Summanden 2 und zusätzlich den «Carry In» Eingang benutzen.

Den letzten Carry-Ausgang benötigen wir nicht, dafür erzeugen wir noch einen Funktionsblock «Volladdierer ohne CarryOut». Ein Volladdierer ohne Carry-Ausgang ist ein simpler Paritätstester mit 3 Eingängen:



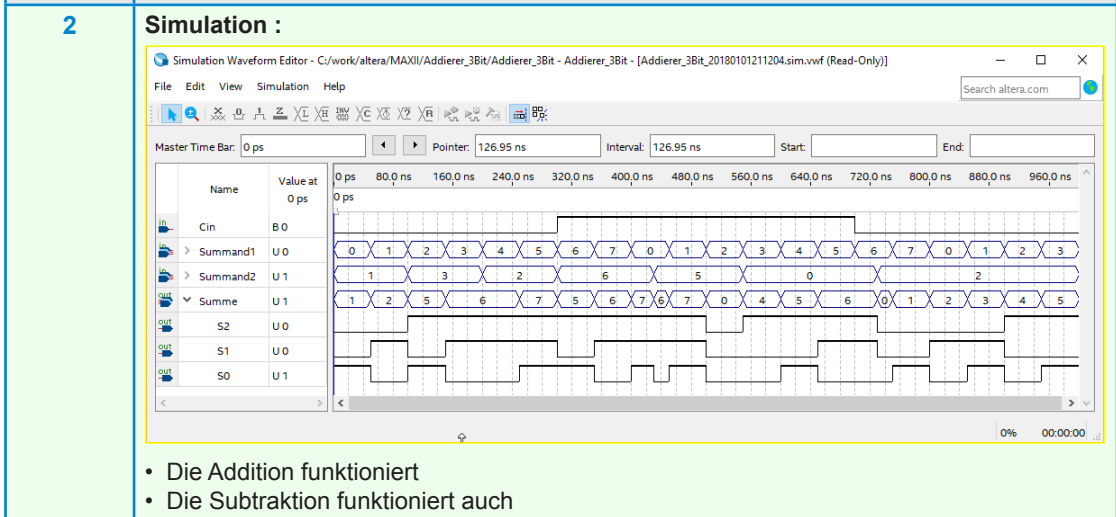
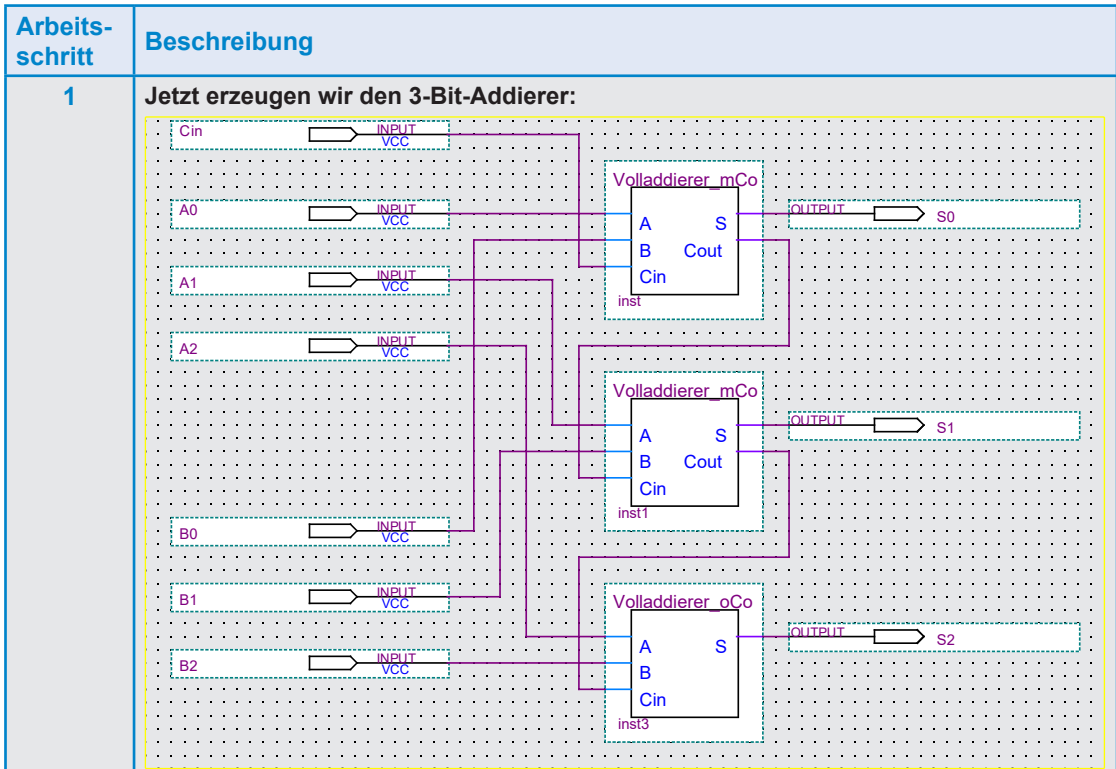
Ressourcen  
 Logic elements:  
 1/1270  
 Total pins:  
 4/212

Wir generieren für beide Volladdierer je einen Funktionsblock:



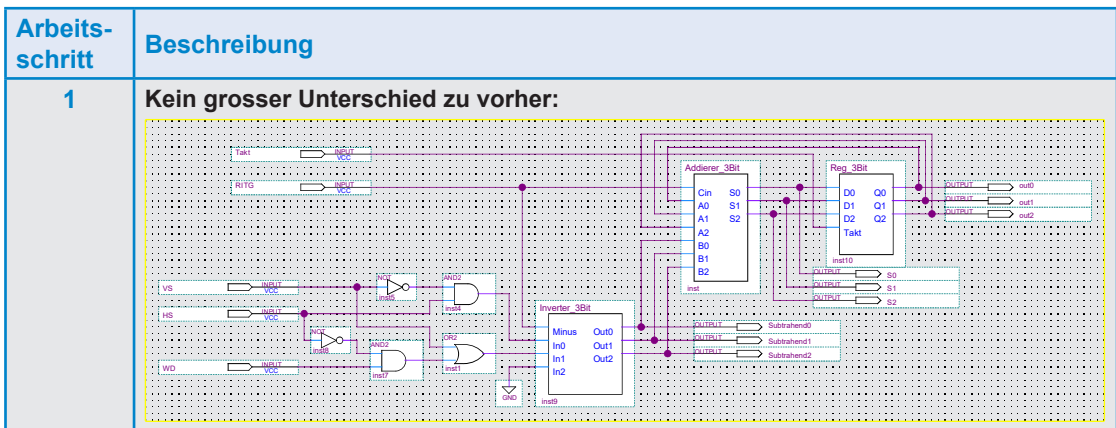
Ressourcen  
Logic elements:  
5/1270  
Total pins:  
10/212

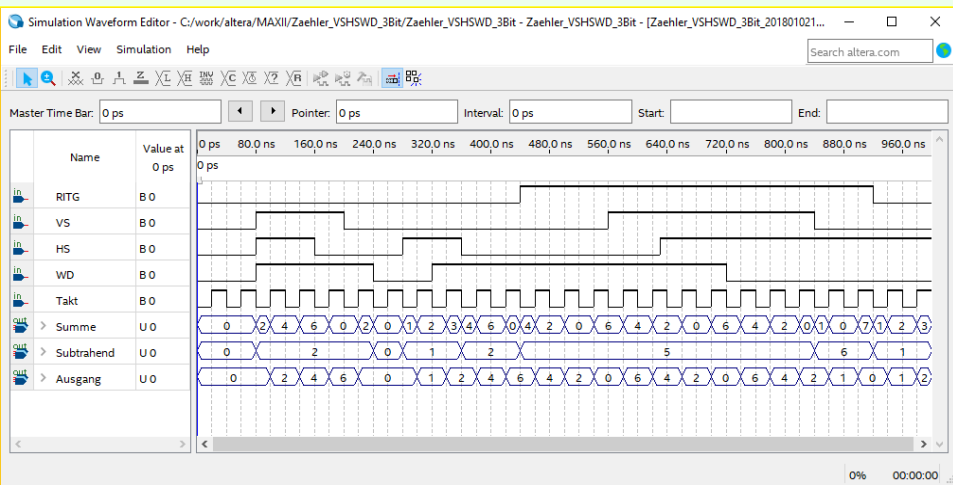
7



Und jetzt bauen wir den Addierer in die vorangehend entwickelte Schaltung ein (Mit Vereinfachung der Blöcke Inverter und Register/Latch von 4 Bit → 3 Bit)

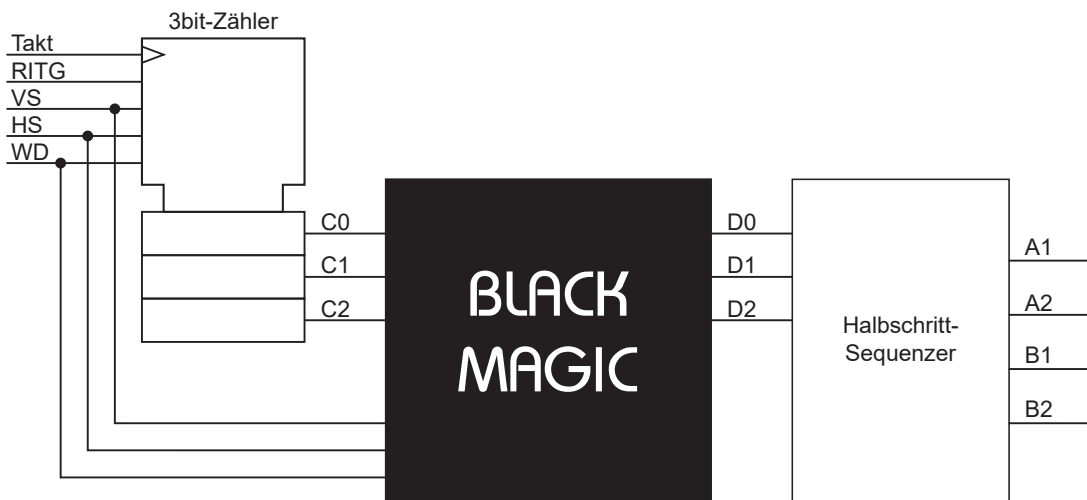
Ressourcen  
Logic elements:  
8/1270  
Total pins:  
14/212



Arbeits-schritt	Beschreibung
<b>2</b>	<p><b>Simulation:</b></p>  <p>• Auch wie vorher aber nur noch mit 3 Bit.</p>

Jetzt könnten wir die Sequenzerschaltung aus «7.4.4 Halbschritt-Sequenzer mit Zähler» am Ausgang anfügen und hätten einen Sequenzer der fast funktioniert. Unser Problem ist noch die richtige Schrittfolge für «Halbschritt» und «Wave Drive». Je nachdem wo der Zähler gerade steht, zählt er nur noch gerade oder nur noch ungerade Schritte. Am Anfang haben wir gesagt, dass wir später entscheiden werden wie dieses Problem lösen. Ich habe 15 Jahre lang versucht in den Zähler selber einzugreifen um die Schrittweiten passend zu verändern und ohne Schrittverluste hin- und herzuschalten. Ich habe in den ganzen 15 Jahren keine zufrieden stellende Lösung gefunden (Zwischenspeichern der Schrittkorrekturen, wieder zurück korrigieren und das in diversen Variationen, direktes Eingreifen in den Sequenzer, usw.). Die Lösung ist mir dann mal während einer Autofahrt in den Sinn gekommen. Einfach so, während ich über eine Autobahnbrücke fuhr. Und hier ist sie:

### 7.4.5.3 Die Korrekturschaltung «BLACK MAGIC»



7

Die Lösung liegt eigentlich nicht im Block «BLACK MAGIC» sondern darin, dass man ihn hat und dass man ihn zwischen den Zähler und den Schaltwerk für die Spulensteuerung schaltet. Dadurch wird der Zähler vom Sequenzer entkoppelt. Die Korrektur wird vorgenommen, ohne dass der Wert des Zähler verändert wird. Das bedeutet, dass sich sein Wert nicht während dem Umschalten der Schrittartern nicht verändert und er beim Zurückschalten wieder in den ursprünglichen Zustand geht. Somit entstehen keine Schrittverluste und auch keine Doppelschritte wie bei anderen Ansätzen.

Dieses Konzept ist auch anwendbar für die Realisation eines Sequenzer in einem Mikrokontroller. Egal ob Assembler, C, C++, Python oder andere Programmiersprachen, diese oder an die Anforderungen angepasste Logik kann zwischen den Zähler und den Sequenzer geschaltet werden.

Die Schaltung selber haben wir schnell hergeleitet

**Wahrheitstabelle «BLACK MAGIC»:**

WT-Zeile	Halbschritt						Vollschritt						Wave Drive					
	Eingang			Ausgang			Eingang			Ausgang			Eingang			Ausgang		
	C2	C1	C0	D2	D1	D0	C2	C1	C0	D2	D1	D0	C2	C1	C0	D2	D1	D0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
2	0	0	1	0	0	1	0	0	1	0	1	0	0	0	1	0	0	1
3	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	1
4	0	1	1	0	1	1	0	1	1	0	0	0	0	1	1	0	1	1
5	1	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	1	1
6	1	0	1	1	0	1	1	0	1	1	0	0	0	1	0	1	0	1
7	1	1	0	1	1	0	1	1	0	1	1	0	0	1	1	1	1	1
8	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1

Daraus sehen wir:

- Bei Halbschritt ändern wir nichts
- Bei Vollschritt benötigen wir nur gerade Zahlen und addieren deshalb immer eins zum Eingang der Wert ungerade ist
- Bei Wave Drive benötigen wir nur ungerade Zahlen und addieren deshalb immer eins zum Eingang der Wert gerade ist

Wir testen deshalb den Ausgangswert des Zählers auf gerade und ungerade Parität und addieren entsprechend der Prioritäten der Schrittartsteuerung den Wert 1 zum Ausgangswert des Zählers.

Dafür modifizieren wir die bestehende Wahrheitstabelle aus «7.4.5.3 Zuweisung der Schrittarten zum Zähler»:

VS = Vollschritt mit Prio 1; HS = Halbschritt mit Prio 2; WD = Wave Drive mit Prio 3

Prioritätstabelle			
Schrittart			Korrektur
WD	VS	HS	Dezimal
0	0	0	keine Korrektur = 0
0	0	1	keine Korrektur = 0
0	1	0	+1 wenn ungerade
0	1	1	+1 wenn ungerade
1	0	0	+1 wenn gerade
1	0	1	keine Korrektur = 0
1	1	0	+1 wenn ungerade
1	1	1	+1 wenn ungerade

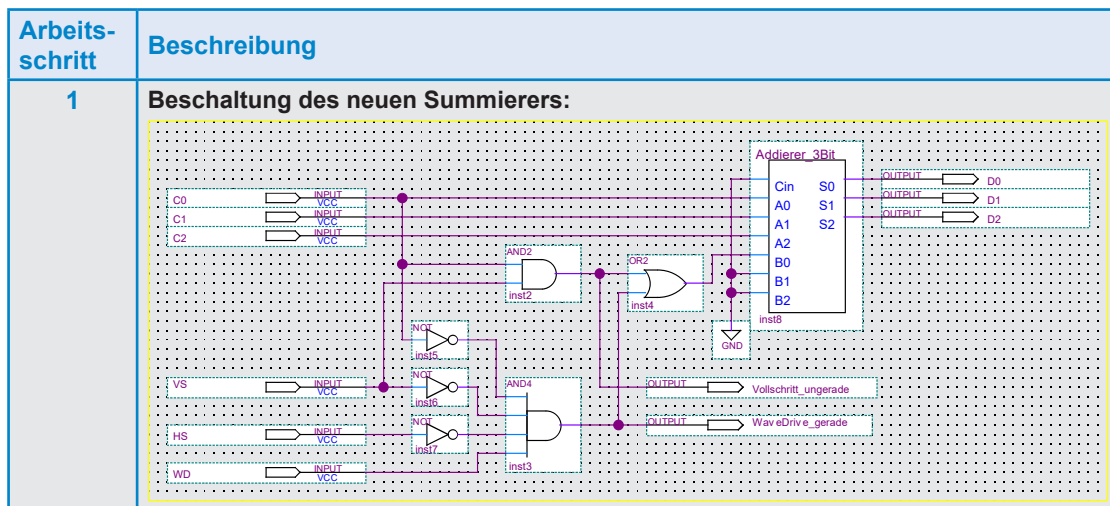
Damit können wir die Funktion des Korrekturbits herleiten:

$$\text{Korrekturbit} = \text{VS} \bullet \text{ungerade Zahl} + \text{WD} \bullet \overline{\text{VS}} \bullet \overline{\text{HS}} \bullet \text{gerade Zahl}$$

$$\text{Korrekturbit} = \text{VS} \bullet \text{C0} + \text{WD} \bullet \overline{\text{VS}} \bullet \overline{\text{HS}} \bullet \overline{\text{C0}}$$

Jetzt fügen wir nach dem Zähler einen Addierer hinzu bei dem wir immer nur «plus 1» rechnen wenn es notwendig ist.

Umsetzung in programmierbarer Logik:



Ressourcen  
 Logic elements:  
 5/1270  
 Total pins:  
 11/212

Arbeits-schritt	Beschreibung
<b>2</b>	<p><b>Simulation:</b></p> <p>• Genau was wir wollten</p>

Wir können aber immer noch etwas verbessern. Den Summierer können wir für diese Anwendung einfacher aufbauen. Die Eingänge «Bn» benötigen wir gar nicht. Muss +1 gerechnet werden setzen wir einfach das «Carry In» und können den 3-Bit-Addierer so mit 3 Halb-Addierern aufbauen.

Dafür modifizieren wir unseren Halbaddierer aus «7.3.1 Halbaddierer mit Schemaeingabe» indem wir die Eingänge umbenennen.

Ressourcen  
Logic elements:  
2/1270  
Total pins:  
4/212

Arbeits-schritt	Beschreibung
<b>1</b>	<p><b>Zuerst erzeugen wir den Carry-Halb-Addierer:</b></p>

Ressourcen  
Logic elements:  
3/1270  
Total pins:  
7/212

Arbeits-schritt	Beschreibung
<b>2</b>	<p><b>Jetzt erzeugen wir den Addierer «3-Bit-Plus-CarryIn» (vereinfachten 3-Bit-Addierer):</b></p>



<b>Arbeits-schritt</b>	<b>Beschreibung</b>
<b>3</b>	<p><b>Simulation :</b></p> <p>• Die Addition +1 funktioniert</p>

Jetzt ersetzen wir den 3-Bit-Volladdierer in der vorangehenden Schaltung durch den neuen vereinfachten Plus1-Addierer:

7

<b>Arbeits-schritt</b>	<b>Beschreibung</b>	<p>Ressourcen Logic elements: 5/1270 Total pins: 11/212</p>
<b>1</b>	<p><b>Schaltung BlaMa2: Variante 2 von BlackMagic</b></p>	
<b>2</b>	<p><b>Simulation:</b></p> <p>• Genau was wir wollten</p>	

Jetzt fügen wir diese Schaltung zwischen Zähler und Sequenzer ein:

Ressourcen  
Logic elements:  
14/1270  
Total pins:  
15/212

7

Arbeits-schritt	Beschreibung
<b>1</b>	<p><b>Komplette Schaltung:</b></p> <p>The schematic shows a logic circuit for a motor sequence controller. It features a counter (Zaehler) and a register (Reg) connected to a decoder (blama2). The decoder outputs four signals (A, B, C, D) which are connected to the motor's phase outputs. The circuit is implemented in a logic device with various logic elements like AND, OR, and NOT gates.</p>
<b>2</b>	<p><b>Simulation:</b></p> <p>The simulation waveform shows the timing of the motor sequence controller. The clock signal (Takt) is a square wave. The counter output (Zaehler) is a sequence of numbers from 0 to 10. The motor outputs (A, B, C, D) are square waves that change state at specific points in the counter sequence, corresponding to the motor's steps.</p> <ul style="list-style-type: none"> <li>• Genau was wir wollten</li> </ul>

Jetzt fehlt nur noch die Zeitsteuerung. Die Entwicklung eines «Numerisch Kontrollierten Oszillators» NCO würde den Rahmen dieses Buches sprengen, den werden wir dafür in Buch 3 behandeln und erzeugen die Schrittfrquenz vorläufig mit den in diesem Kapitel schon behandelten Zählern. Wir wählen eine Schrittfrquenz von 50 Hz.

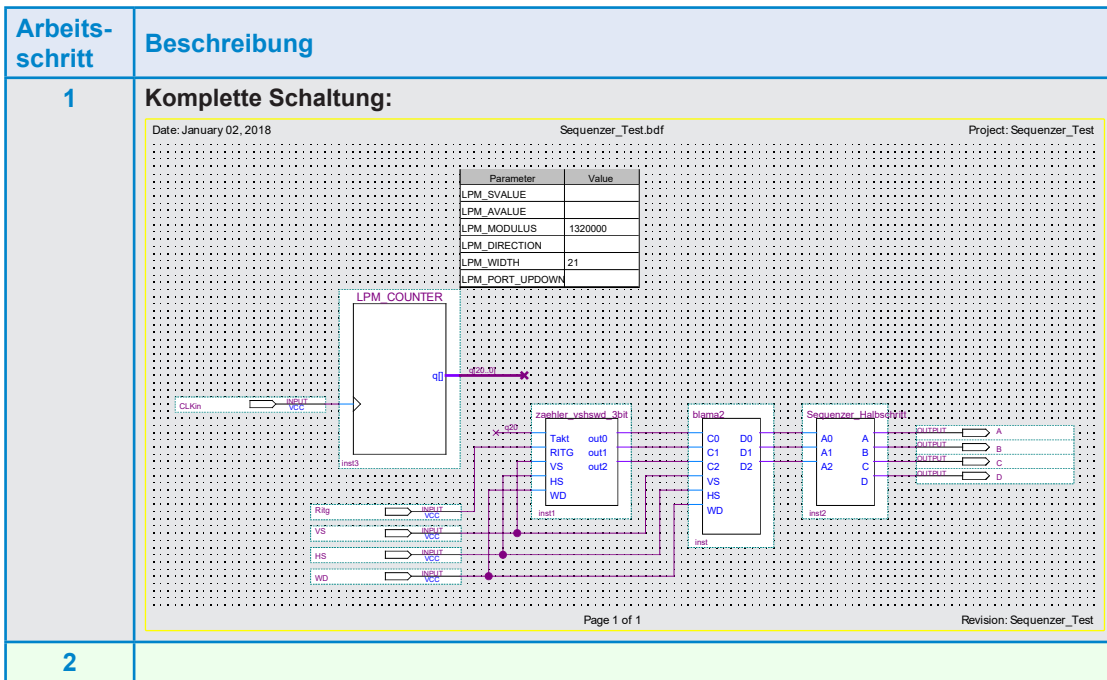
**Erzeugung des Taktsignals für den Test des Sequenzer:**

Wir benutzen wieder die Megafunktion «LPM\_COUNTER» und müssen wieder die Parameter an die Frequenz 50 Hz anpassen.

Dafür geben wir in «Properties» neue Werte ein

«LPM\_MODULUS» =  $66\,000\,000\text{ Hz} : 50\text{ Hz} = 1\,320\,000$

«LPM\_WITDH» = 21; aus  $\log 1\,320\,000 : \log 2 = 20.33$  aufgerundet



Ressourcen  
 Logic elements:  
 43/1270  
 Total pins:  
 9/212

Pegelwandler

