Übung-03

3.1 Halbaddierer und Volladdierer (6 Punkte)

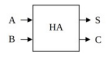
Hier werden Sie sich zunächst mit den grundlegenden Bausteinen von Addierern vertraut machen.

3.1.1 Halbaddierer

Der Algorithmus des schriftlichen Addierens zerlegt die binäre Addition in die folgenden elementaren Additionen. Es ergibt sich für die Eingaben A und B eine Summe S und ein Übertrag C (Carry) mit der zugehörigen Funktionstabelle:

$A+B=S,\ C$
$0+0=0,\ C=0$
$0+1=1,\ C=0$
$1+0=1,\ C=0$
$1+1=0,\ C=1$

Eine Digitalschaltung, die diese Funktion rechnen soll, habe die Eingänge A und B und die Ausgänge S und C:

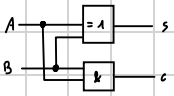


Aufgaben:

a. (1 Punkte) Zeichnen Sie die Rechenschaltung des Halbaddierers. Geben Sie desweiteren die Berechnungsvorschrift des Halbaddierers mit XOR und AND Operationen an. ($S = (R \wedge T) \vee L$ wäre zum Beispiel eine Berechnungsvorschrift.)

b. (2 Punkte) Vervollständigen Sie die Implementierung des Halbaddierers (mit XOR und AND Operationen). Nutzen Sie dazu die bereitgestelle Vorlage ($ha.vhdl$). Testen Sie den Halbaddierer in der Testbench ($ha\_tb.vhdl$) mit allen Inputkombinationen von 0 und 1.

a.



A ∧ B = C

A ⊕ B = S bzw. ¬(A ↔ B) = S

b. Quellcode wurde hier exakt, wie in der gegebenen Struktur implementiert.

C_{n-1} / C_{alt}	a	b	c	s	
0	0	0	0	0	✓
0	0	1	0	1	✓
0	1	0	0	1	✓
0	1	1	1	0	✓
1	0	0	0	1	✓
1	0	1	1	0	
1	1	0	1	0	
1	1	1	1	1	

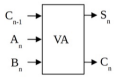
⇒

3.1.2 Volladdierer

Für die Addition zweier mehrstelliger Binärzahlen müssen drei Binärziffern addiert werden können: die beiden Summanden und der Übertrag von der vorhergehenden Stelle. Nur in der niederwertigsten Stelle (LSB = least significant bit) gibt es keinen Übertrag. Es gilt $S_n = A_n \oplus B_n \oplus C_{n-1}$. Es ergibt sich folgende Funktionstabelle:

$A_n+B_n+C_{n-1}=S_n,\ C_n$
$0+0+0=0,\ C_n=0$
$0+0+1=1,\ C_n=0$
$0+1+0=1,\ C_n=0$
$0+1+1=0,\ C_n=1$
$1+0+0=1,\ C_n=0$
$1+0+1=0,\ C_n=1$
$1+1+0=0,\ C_n=1$
$1+1+1=1,\ C_n=1$

Darin bedeuten die Indizes, dass die Schaltung bei der Addition zweier mehrstelliger Binärzahlen die Addition für die n-te Stelle durchführen soll. Das Blockdiagramm des Volladdierers sieht wie folgt aus:



Aufgaben:

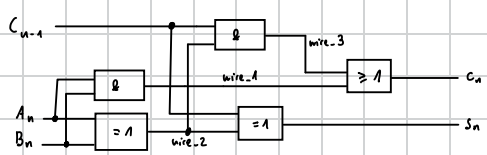
a. (1 Punkt) Zeichnet die Rechenschaltung des Volladdierers.

b. (2 Punkte) Vervollständigen Sie die Implementierung des Volladdierers. Füllen Sie dazu die bereitgestelle Vorlage aus ($fa.vhdl$). Testen Sie alle Inputkombinationen von 0 und 1 mit der Testbench ($fa\_tb.vhdl$).
Hinweis: Sie können den Volladdierer wahlweise direkt implementieren oder mit Hilfe der in Aufgabenteil 3.1.1 erstellten Halbaddierer.

a.

$S = C_{n-1} \oplus A_n \oplus B_n$

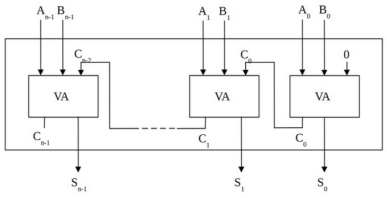
$C = \left(C_{n-1} \wedge (A_n \oplus B_n)\right) \vee (A_n \wedge B_n)$



b. Quellcode wurde hier exakt, wie in der gegebenen Struktur implementiert.

3.2 Ripple-Carry-Addierwerk (3 Punkte)

Mit n Volladdierern (alternative mit $n-1$ Volladdierern und einem Halbaddierer) kann man eine Digitalschaltung aufbauen, die zwei n-stellige Binärzahlen $A_{n-1},\ldots,A_0$ und $B_{n-1},\ldots,B_0$ addiert:

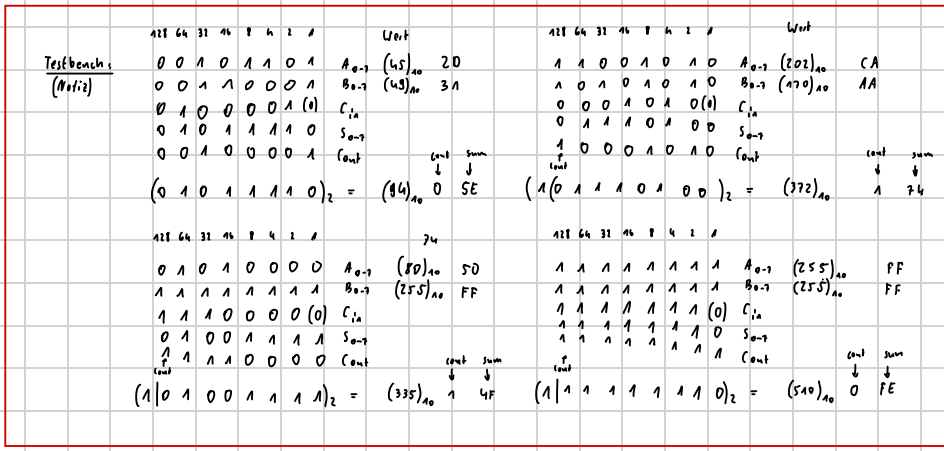


Im Addierwerk des Ripple-Carry-Adders arbeiten die Volladdierer parallel, d.h. gleichzeitig. Die von den Volladdierern berechneten Summen stehen aber nicht zur gleichen Zeit zur Verfügung, weil jeder der Volladdierer einen Übertrag von der nächstniedrigeren Stelle erhält. Die Summenwerte an den Ausgängen des Addierwerks sind erst dann gültig, wenn der Volladdierer der Stelle m den Übertrag C_{m-1} erhalten hat. Die Überträge entstehen also nacheinander. Erst wenn der Übertrag C_{n-2} vorliegt, steht das Ergebnis zur Verfügung. In diesem Sinne arbeitet der Ripple-Carry-Adder seriell.

Aufgabe: (3 Punkte) Vervollständigen Sie die Implementierung eines 8 bit Ripple-Carry-Addierwerk in der bereitgestellten Vorlage (*rca.vhdl*). Nutzen Sie dazu die HA und VA-Bausteine die Sie in den vorherigen Aufgaben implementiert haben. Testen Sie Ihre Implementierung in einer Testbench mit (mindestens) 4 Inputkombinationen.

Quellcode wurde hier exakt, wie in der gegebenen Struktur implementiert.

Testbench: (Notiz)



3.3 Carry-Look-Ahead Addierwerk (8 Punkte)

Die Idee des Carry-Look-Ahead-Adders ist es, die Carry-Signale nicht mehr von Adder-Modul zu Adder-Modul weiterzureichen, sondern in einer zusätzlichen kombinatorischen Schaltung direkt aus den Eingangsgrößen A_n und B_n zu erzeugen. Dabei sollen die Signale parallel über möglichst wenige Gatter laufen und alle Carry-Signale nach der selben Verzögerungszeit berechnet werden.

Als Beispiel wählen wir ein vierstelliges Addierwerk, das kaskadierbar ist. Dadurch kann ein Carry C_{-1} von einem Addierwerk übernommen werden und es kann ein Carry C_3 an ein anderes Addierwerk weitergeleitet werden.

Das Addierwerk berechnet folgende Summen und Carry-Werte:

$$S_0 = A_0 \oplus B_0 \oplus C_{-1},\ C_0 = (A_0 \wedge B_0) \vee ((A_0 \vee B_0) \wedge C_{-1})$$
$$S_1 = A_1 \oplus B_1 \oplus C_0,\ \ C_1 = (A_1 \wedge B_1) \vee ((A_1 \vee B_1) \wedge C_0)$$
$$S_2 = A_2 \oplus B_2 \oplus C_1,\ \ C_2 = (A_2 \wedge B_2) \vee ((A_2 \vee B_2) \wedge C_1)$$
$$S_3 = A_3 \oplus B_3 \oplus C_2,\ \ C_3 = (A_3 \wedge B_3) \vee ((A_3 \vee B_3) \wedge C_2)$$

Hinweis: Die Klammerung bindet stärker als $\wedge$ was wiederum stärker bindet als $\vee$. Daher sind, z.B., die Klammern um $(A_0 \vee B_0) \wedge C_{-1}$ in $(A_0 \wedge B_0) \vee ((A_0 \vee B_0) \wedge C_{-1})$ nicht nötig, wurden aber zur Erhöhung der Übersichtlichkeit gesetzt.

Wir führen zwei Hilfsvariablen g_n und p_n ein:
$g_n = A_n \wedge B_n,\ p_n = A_n \vee B_n$

- g_n heißt Carry generate, weil ein Übertrag C_n gebildet wird, wenn sowohl A_n als auch B_n 1 sind.
- p_n heißt Carry propagate, weil der Übertrag C_{n-1} weitergeleitet wird, wenn $p_n = 1$ und $g_n = 0$ ist.

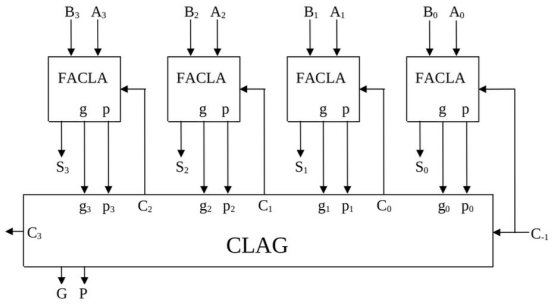
Setzt man g_n und p_n ein ergibt sich für die C_n:

$$C_0 = g_0 \vee (p_0 \wedge C_{-1})$$
$$C_1 = g_1 \vee (p_1 \wedge C_0)$$
$$C_2 = g_2 \vee (p_2 \wedge C_1)$$
$$C_3 = g_3 \vee (p_3 \wedge C_2)$$

Nach Ersetzen von C_1, C_2 und C_3 auf den rechten Seiten ergibt sich:

$$C_0 = g_0 \vee (p_0 \wedge C_{-1})$$
$$C_1 = g_1 \vee (p_1 \wedge g_0) \vee (p_1 \wedge p_0 \wedge C_{-1})$$
$$C_2 = g_2 \vee (p_2 \wedge g_1) \vee (p_2 \wedge p_1 \wedge g_0) \vee (p_2 \wedge p_1 \wedge p_0 \wedge C_{-1})$$
$$C_3 = g_3 \vee (p_3 \wedge g_2) \vee (p_3 \wedge p_2 \wedge g_1) \vee (p_3 \wedge p_2 \wedge p_1 \wedge g_0) \vee (p_3 \wedge p_2 \wedge p_1 \wedge p_0 \wedge C_{-1})$$

Wenn man nun die Volladdiererschaltung so umbaut, dass sie neben der Summe S_n auch die Hilfsvariablen g_n und p_n liefert, kann man einen n-stelligen Carry-Look-Ahead-Adder bauen:



Die Blackboxes FACLA (Full-Adder-Carry-Look-Ahead) enthalten die umgebaute Volladdiererschaltung. Der CLAG (Carry-Look-Ahead-Generator) erzeugt aus den g- und p-Hilfsvariablen die Überträge C_n. Man beachte: Die in dieser Schaltung benutzten Volladdierer in den FACLAs erzeugen keine Überträge.

Über die Ein/Ausgänge C_{-1}, C_3, G und P können mit mehreren CLAGs mehrstufige Carry-Look-Ahead-Generatoren erzeugt werden. Dabei werden G und P wie folgt berechnet:

$$G = g_3 \vee (p_3 \wedge g_2) \vee (p_3 \wedge p_2 \wedge g_1) \vee (p_3 \wedge p_2 \wedge p_1 \wedge g_0)$$
$$P = p_3 \wedge p_2 \wedge p_1 \wedge p_0$$

Eine Kaskadierung mit C_3,G, P sowie C_{-1} wird bei unseren Versuchen nicht benötigt. Daher ist $C_{-1} = 0$.

Aufgaben:

a. (2 Punkt) Geben Sie ein Rechenbeispiel für einen CLA-Durchlauf an bei dem zwei 4-bit Werte addiert werden, mit den zugehörigen A, B, g, p, C_n, S.

b. (4 Punkte) Implementieren Sie einen 4-bit CLA in VHDL. Dabei sollen der CLAG- und der FACLA-Baustein als seperate Komponenten implementiert werden. Der CLAG und FACLA sollen in einem CLA-Baustein als *component* genutzt werden. Es sollen folgende Dateien erweitert werden, welche mit dem Blatt hochgeladen worden sind: *clag.vhdl*, *facla.vhdl*, *cla.vhdl*. Schreiben Sie zudem eine Testbench und testen Sie Ihren Addierer mit (mindestens) 4 Inputkombinationen.
Hinweis: Die Fehlersuche fällt leichter, wenn Sie erst *facla.vhdl* und *clag.vhdl* implementieren und die Korrektheit dieser Bausteine mit Hilfe einer Testbench überprüfen, bevor diese in *cla.vhdl* verwendet werden.

c. (2 Punkt) Welche Vor-und Nachteile haben die jeweiligen Addierwerke RCA und CLA? Halten Sie diese in einer Tabelle fest.

a. zu addieren: $A = (1\,0\,1\,0)_2$ ($A_3 A_2 A_1 A_0$), $B = (1\,1\,0\,1)_2$ ($B_3 B_2 B_1 B_0$), $C_{-1} = 0$ — ● = Signal im CLA ● = Signal aus CLA

n=0:
$S_0 = A_0 \oplus B_0 \oplus C_{-1} = 0 \oplus 1 \oplus 0 = \underline{1}$
$g_0 = A_0 \wedge B_0 = 0 \wedge 1 = \underline{0}$ $p_0 = A_0 \vee B_0 = 0 \vee 1 = \underline{1}$ } in FACLA
$C_0 = g_0 \vee (p_0 \wedge C_{-1}) = 0 \vee (1 \wedge 0) = \underline{0}$ } in CLAG
von CLAG liegt C_0-Signal zu FACLA mit $n>1$ an

n=1:
$S_1 = A_1 \oplus B_1 \oplus C_0 = 1 \oplus 0 \oplus 0 = \underline{1}$ } in FACLA
$g_1 = A_1 \wedge B_1 = 1 \wedge 0 = \underline{0}$ $p_1 = A_1 \vee B_1 = 1 \vee 0 = \underline{1}$
$C_1 = g_1 \vee (p_1 \wedge C_0) = 0 \vee (1 \wedge 0) = \underline{0}$ } in CLAG
$= g_1 \vee (p_1 \wedge g_0) \vee (p_1 \wedge p_0 \wedge C_{-1}) = 0 \vee (1 \wedge 0) \vee (1 \wedge 0 \wedge 0) = \underline{0}$
von CLAG liegt C_1-Signal zu FACLA mit $n>2$ an

n=2:
$S_2 = A_2 \oplus B_2 \oplus C_1 = 0 \oplus 1 \oplus 0 = \underline{1}$ } in FACLA
$g_2 = A_2 \wedge B_2 = 0 \wedge 1 = \underline{0}$ $p_2 = A_2 \vee B_2 = 0 \vee 1 = \underline{1}$
$C_2 = g_2 \vee (p_2 \wedge C_1) = 0 \vee (1 \wedge 0) = \underline{0}$ } in CLAG
$= g_2 \vee (p_2 \wedge g_1) \vee (p_2 \wedge p_1 \wedge g_0) \vee (p_2 \wedge p_1 \wedge p_0 \wedge C_{-1}) = \underline{0}$
von CLAG liegt C_2-Signal zu FACLA mit $n>3$ an

n=3:
$S_3 = A_3 \oplus B_3 \oplus C_2 = 1 \oplus 1 \oplus 0 = \underline{0}$ } in FACLA
$g_3 = A_3 \wedge B_3 = 1 \wedge 1 = \underline{1}$ $p_3 = A_3 \vee B_3 = 1 \vee 1 = \underline{1}$
$C_3 = g_3 \vee (p_3 \wedge C_2) = 1 \vee (1 \wedge 0) = \underline{1}$ } in CLAG
$= g_3 \vee (p_3 \wedge g_2) \vee (p_3 \wedge p_2 \wedge g_1) \vee (p_3 \wedge p_2 \wedge p_1 \wedge g_0) \vee (p_3 \wedge p_2 \wedge p_1 \wedge p_0 \wedge C_{-1})$

Notiz für Test:

Signal-Output von CLA ist: (C_3 S_3 S_2 S_1 S_0) $= (1\,|\,0\,1\,1\,1)$

$g_0 = 0$ $g_1 = 0$ $g_2 = 0$ $g_3 = 1$

$p_0 = 1$ $p_1 = 1$ $p_2 = 1$ $p_3 = 1$

$C_0 = 0$ $C_1 = 0$ $C_2 = 0$ $C_3 = 1$

$P = 1 = 1 \wedge 1 \wedge 1 \wedge 1$

$G = 1$, da $g_3 = 1$

b. Quellcode wurde hier erneut, wie in der gegebenen Struktur implementiert.

C.

	Vorteile	Nachteile
RCA	• weniger Ressourcen / sehr klein ⇒ Größe eines RCA ist $5n-3$, wobei n ein logisches Gatter darstellt • einfach zu implementieren ⇒ der Carry eines VA muss einfach nur mit dem nächsten VA verbunden werden.	• Schnelligkeit ist geringer, da VA aufeinander "warten" müssen ⇒ sehr tiefe Schaltung • Ineffizienz wächst linear mit seiner Größe ⇒ für Bitvektoren der Größe n, wächst die Verzögerung linear mit durch proportional gleich viel eingesetzte VA
CLA	• hohe Geschwindigkeit ⇒ durch eine parallele Berechnung der Carry Überträge • gleich bleibende Tiefe von 4 von beliebig langen Bit-Vektoren ⇒ Verzögerung entspricht einer sub-linearen Entwicklung, statt linear wie beim RCA	• großer Ressourcenverbrauch ⇒ Verknüpfung für C, G, P Signale stark anwächst, somit eine erhöhte Anzahl von insbesondere &-Gattern und Verknüpfung aller Gatter realisiert werden muss (höherer Energieverbrauch auf physikalischer Ebene) • Komplexität in der Realisierung ⇒ bedingt durch sich vergrößernde Logik-Formeln (insb. für G und P-Signal) und Verknüpfungen der Signale