yunfeng.huang [☺] tu-dortmund.de
mikail.yayla [☺] tu-dortmund.de

Hardwarepraktikum
Wintersemester 20/21

Übungsblatt 3

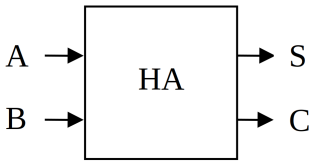
Bearbeitung ab Samstag, 28. November 2020

3.1 Halbaddierer und Volladdierer

3.1.1 Halbaddierer

Der bürgerliche Algorithmus des schriftlichen Addierens zerlegt die binäre Addition in die folgenden elementaren Additionen. Es ergibt sich eine Summe S und ein Carry C (Übertrag) mit der zugehörigen Funktiontabelle:
$A+B=S,\ C$
$0+0=0,\ C=0$
$0+1=1,\ C=0$
$1+0=1,\ C=0$
$1+1=0,\ C=1$

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



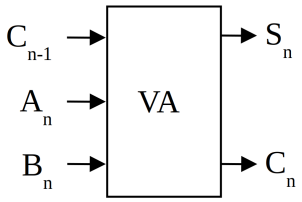
Aufgaben:

a. (1 Punkte) Zeichnet die Rechenschaltung des Halbaddierers und die Funktionstabelle. Gebt desweiteren die Berechnungsvorschrift des HAs mit XOR und AND Operationen an.

b. (2 Punkte) Vervollständigt die Implementierung des Halbaddierers (mit XOR und AND Operationen). Nutzt dazu die bereitgestelle Vorlage ($ha.vhdl$). Testet den HA mit der Testbench ($ha\_tb.vhdl$) mit allen Inputkombinationen.

3.1.2 Volladdierer

Für die Addition zweier mehrstelliger Binärzahlen müssen drei Binärziffern addiert werden können: der Übertrag von der vorhergehenden Stelle und die beiden Summanden. Nur in der niederwertigsten Stelle (LSB) gibt es keinen Übertrag. Es ergibt sich folgende Funktiontabelle:
$A_n+B_n+C_{n-1}=S_n,\ C_n$
$0+0+0=0,\ C_n=0$
$0+0+1=1,\ C_n=0$
$0+1+0=1,\ C_n=0$
$0+1+1=0,\ C_n=1$
$1+0+0=1,\ C_n=0$
$1+0+1=0,\ C_n=1$
$1+1+0=0,\ C_n=1$
$1+1+1=1,\ C_n=1$
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:



Es gilt $S_n = A_n \oplus B_n \oplus C_{n-1}$.

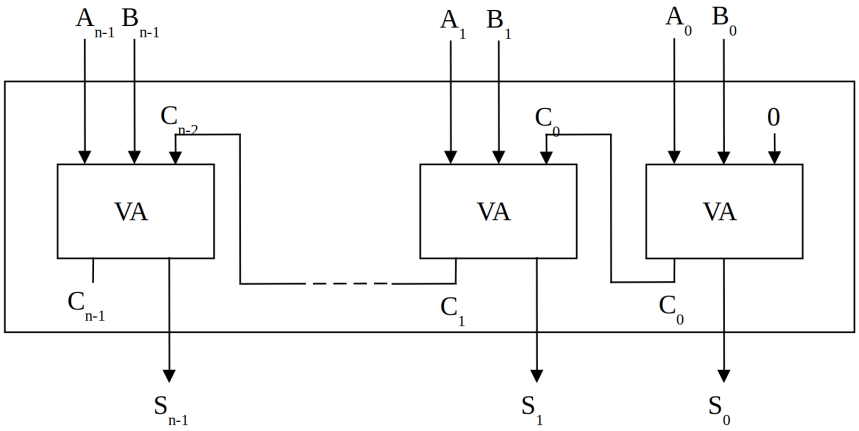
Aufgaben:

a. (1 Punkt) Zeichnet die Rechenschaltung des Volladdierers und gebt die Funktionstabelle an.

b. (2 Punkte) Vervollständigt die Implementierung des Volladdierers. Füllt dazu die bereitgestelle Vorlage aus ($fa.vhdl$). Testet einige Inputkombinationen mit der Testbench ($fa\_tb.vhdl$).

3.2 Ripple-Carry-Addierwerk

Mit Volladdierern und Halbaddierern 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 mit dem höchsten Gewicht ($2^{(n-1)}$) den Übertrag C_{n-1} erhalten hat. Die Überträge entstehen also nacheinander. Erst wenn der Übertrag C_{n-1} vorliegt, steht das Ergebnis zur Verfügung. In diesem Sinn arbeitet der Ripple-Carry-Adder seriell.

Aufgaben:

a. (2 Punkte) Vervollständigt die Implementierung eines 8 bit Ripple-Carry-Addierwerk in der bereitgestellten Vorlage ($rca.vhdl$). Nutzt dazu die HA und VA-Bausteine die ihr in den vorherigen Aufgaben implementiert habt. Testet eure Implementierung in einer Testbench mit einigen Inputkombinationen.

b. (2 Punkte) Erweitert den Addierer um eine Subtrahierfunktion. Der erweiterte RCA soll für eine der beiden Funktionen (Addition oder Subtraktion) konfigurierbar sein. Implementiert die Konfigierbarkeit durch einen Select-Eingang. Testet die Subtrahierfunktion in einer Testbench.

3.3 Carry-Look-Ahead Addierwerk

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 gewonnen 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 weitergeleitet werden.
Das Addierwerk berechnet folgende Summen und Carry-Werte:

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

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

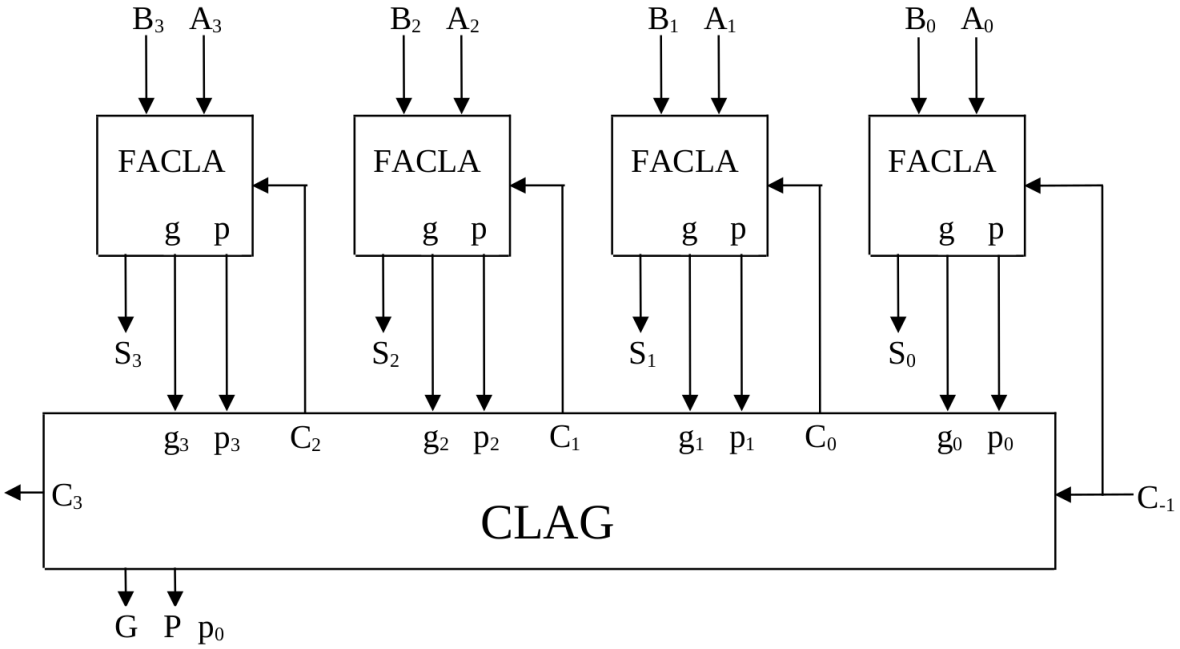
- g_n heißt Carry generate, weil ein Übertrag C_n gebildet wird, wenn sowohl A_n als auch B_n den Binärzustand 1 haben.
- p_n heißt Carry propagate, weil der Übertrag C_{n-1} weitergeleitet wird, wenn $p_n = 1$ und $g_n = 0$ ist.

Setzt man g_n und p_n ein ergibt sich für die C_n: (ohne Klammerung bindet $\wedge$ stärker als $\vee$)
$C_0 = g_0 \vee p_0 \wedge C_{-1}$
$C_1 = g_1 \vee p_1 \wedge C_0$
$C_2 = g_2 \vee p_2 \wedge C_1$
$C_3 = g_3 \vee p_3 \wedge C_2$

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) erhalten 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:
$C_3 = g_3 \vee p_3 \wedge g_2 \vee p_3 \wedge p_2 \wedge g_1 \vee p_3 \wedge p_2 \wedge p_1 \wedge g_0 \vee p_3 \wedge p_2 \wedge p_1 \wedge p_0 \wedge C_{-1}$ mit
$G = g_3 \vee p_3 \wedge g_2 \vee p_3 \wedge p_2 \wedge g_1 \vee p_3 \wedge p_2 \wedge p_1 \wedge g_0$ und
$P = p_3 \wedge p_2 \wedge p_1 \wedge p_0$.
Eine Kaskadierung mit C_3,G, P sowie C_{-1} wird bei unseren Versuchen nicht benötigt. Daher ist $C_{-1} = 0$.

Aufgaben:

a. (1 Punkt) Gebt zwei Rechenbeispiele 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) Implementiert 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 ausgefüllt werden werden, welche mit dem Blatt hochgeladen worden sind: $clag.vhdl$, $facla.vhdl$, $cla.vhdl$. Schreibt zudem eine Testbench und testet euren Addierer mit einigen Inputkombinationen. *Hinweis:* Die Fehlersuche fällt leichter, wenn ihr erst $facla.vhdl$ und $clag.vhdl$ implementiert und die Korrektheit dieser Bausteine überprüft, bevor ihr sie in $cla.vhdl$ verwendet.

c. (1 Punkt) Welche Vor-und Nachteile haben die jeweiligen Addierwerke RCA und CLA? Haltet diese in einer Tabelle fest.

d. (1 Punkt) Welche Arten von Addierern werden in neueren Desktop-CPUs verwendet?