====== Laboratorul 07 - Sumatorul Carry-Lookahead ======

===== Obiective =====

Scopul acestui laborator este proiectarea unui sumator Carry-lookahead, care are performanțe mai bune decât sumatorul Ripple-Carry studiat în laboratorul trecut.

==== 1. Recapitulare ====

==== a. Sumatoare multi-bit ====

În viața reală noi lucrăm cu numere reprezentabile pe mai mulți biți. Totuși, sumatoarele de tip Half-Adder și Full-Adder pot aduca cel mult un bit. Pentru depășirea acestei limitări, laboratorul [[cn1:laboratoare:06|trecut]] am implementat un circuit capabil să facă suma numerelor reprezentate pe mai mulți biți.

=== Sumatorul Ripple-Carry ===

Cel mai simplu circuit este sumatorul ripple-carry care folosind N sumatoare full-adder poate însuma numere pe N biți.

{{ :cn1:laboratoare:06:ripple-carry4bit.png?400 | Ripple-carry pe 4 biți }}

Conectarea lor, prezentată în figura de mai sus, este una foarte simplă: ieșirea C<sub>out</sub> a modulului de rang //i// va fi conectată la intrarea C<sub>in</sub> a modulului de rang //i+1//.

=== Limitări ===

Implementarea acestuia este facilă, însă performanța acestuia este scăzută. \*\*De ce?\*\* Deoarece transportul de intrare pentru fiecare sumator complet este dependent de semnalul carry-out din full-adder-ul anterior. Prin structura sa internă: modulul de rang //i// trebuie să aştepte modulul de rang //i-1// sa îşi termine execuția pentru a afla cât este C<sub>in\_i</sub>.

=== Îmbunătățiri ===

Cum putem implementa un circuit mai rapid? Eliminând dependența între rang-uri în ceea ce privește bitul de carry. Pentru început aflăm expresia în logică booleană pornind de la tabelul de adevăr al bitului de carry C<sub>1</sub>.

^ A<sub>0</sub> ^ B<sub>0</sub> ^ C<sub>0</sub> ^ C<sub>1</sub> ^

| 0 | 0 | 0 | 0 |

| 0 | 0 | 1 | 0 |

| 0 | 1 | 0 | 0 |

| 0 | 1 | 1 | 1 |

| 1 | 0 | 0 | 0 |

| 1 | 0 | 1 | 1 |

| 1 | 1 | 0 | 1 |

| 1 | 1 | 1 | 1 |

\* C<sub>1</sub> = A<sub>0</sub> \* B<sub>0</sub> \* C<sub>0</sub> + A<sub>0</sub> \* B<sub>0</sub> \* ~C<sub>0</sub> + ~A<sub>0</sub> \* B<sub>0</sub> \* C<sub>0</sub> + A<sub>0</sub> \* ~B<sub>0</sub> \* C<sub>0</sub>

\* C<sub>1</sub> = A<sub>0</sub> \* B<sub>0</sub> \* (C<sub>0</sub> + ~C<sub>0</sub>) + C<sub>0</sub> \* (~A<sub>0</sub> \* B<sub>0</sub> + A<sub>0</sub> \* ~B<sub>0</sub>)

\* C<sub>1</sub> = A<sub>0</sub> \* B<sub>0</sub> + C<sub>0</sub> \* (A<sub>0</sub> ⊕ B<sub>0</sub>)

Dacă generalizăm formula, pentru //i = 0//:

\* C<sub>i+1</sub> = A<sub>i</sub> \* B<sub>i</sub> + C<sub>i</sub> \* (A<sub>i</sub> ⊕ B<sub>i</sub>)

Prin urmare, pentru fiecare modul de full-adder, am putea calcula bitul carry de intrare cu cost constant egal cu 1, fără să mai așteptăm propagarea acestuia prin toate celelalte full-adder-e de rang mai mic.

\* C<sub>out</sub> = //Generează\_biții\_carry? \*\*SAU\*\* Propagă\_bitul\_de\_carry\_de\_la\_rangul\_inferior\_dacă\_nu\_e\_generat?//

==== 2. Prezentare generală ====

Pentru a reduce timpul necesar unui calcul, proiectanții folosesc \*\*sumatorul carry-lookahead\*\*.

{{ cn1:laboratoare:07:4-bit\_carry\_lookahead\_adder.svg.png | Sumator Carry-lookahead pe 4 biti }}

În sumatorul cu transport anticipat, fiecare adunare pe bit elimină dependenţa de semnalul carry-out anterior şi impune în schimb utilizarea valorilor celor doi operanzi de intrare. El functioneaza prin generarea a două noi semnale (P si G) pentru fiecare rang binar în funcție de starea intrărilor.\\

\* \*\*G\*\* - reprezintă un semnal de \*\*generare\*\* a unui bit de carry.\\

\* \*\*P\*\* - reprezintă un semnal de \*\*propagare\*\* a unui bit de carry.\\

Deci pentru suma a două numere \*\*A\*\* și \*\*B\*\*, dacă bitul \*\*n\*\* din \*\*A\*\*, împreună cu bitul \*\*n\*\* din \*\*B\*\*

generează un semnal \*\*G\*\*, înseamnă că acești biți generează un bit de carry, iar în cazul unui semnal \*\*P\*\*, bitul de carry de la \*\*n-1\*\* (dacă există) este \*\*propagat\*\* de către biții \*\*n\*\* pe poziția \*\*n+1\*\*.

Pornind de la un sumator \*\*ripple-carry\*\*, unde pentru fiecare full-adder (i), semnalul carry-out, c(i+1), este setat la 1 dacă una dintre următoarele condiții este adevărată:

\* A<sub>i</sub> = 1 AND B<sub>i</sub> = 1

sau

\* (A<sub>i</sub> = 1 OR B<sub>i</sub> = 1) AND c<sub>i</sub> = 1

am observat că putem calcula semnalul carry-out folosind doar semnalele rangului curent:

\* c<sub>i+1</sub> = A<sub>i</sub> & B<sub>i</sub> | c<sub>i</sub> & (A<sub>i</sub> | B<sub>i</sub>) (1)

Pentru un sumator \*\*carry-lookahead\*\* definim G<sub>i</sub>(A<sub>i</sub>, B<sub>i</sub>) ca fiind predicatul logic care este adevărat atunci când A<sub>i</sub> + B<sub>i</sub> generează carry (primul caz)

\* \*\*G<sub>i</sub>(A<sub>i</sub>, B<sub>i</sub>) = A<sub>i</sub> & B<sub>i</sub>\*\*

Si P<sub>i</sub>(A<sub>i</sub>, B<sub>i</sub>) ca fiind predicatul logic care este adevărat atunci cand A<sub>i</sub> + B<sub>i</sub> propagă carry (al doilea caz)

\* \*\*P<sub>i</sub>(A<sub>i</sub>, B<sub>i</sub>) = A<sub>i</sub> | B<sub>i</sub>\*\*

Uneori (dar nu și in laboratorul de față) se folosește și o altă definiție a propagării. Conform acesteia, A<sub>i</sub> + B<sub>i</sub> va propaga dacă există carry anterior, dar nu va propaga dacă nu există acest carry. Astfel, propagagarea o putem exprima:

\* P'<sub>i</sub>(A<sub>i</sub>, B<sub>i</sub>) = A<sub>i</sub> ⊕ B<sub>i</sub>

Desi OR(prima variantă) e mai rapid decât XOR(a doua variantă), pentru un sumator carry-lookahead pe mai multe nivele e mai avantajos să folosim P'<sub>i</sub>(A<sub>i</sub>, B<sub>i</sub>).

Rescriind ecuația (1) cu P<sub>i</sub> si G<sub>i</sub>, rezultă:

\* \*\*c<sub>i+1</sub> = G<sub>i</sub> | P<sub>i</sub> & c<sub>i</sub>\*\*

Pentru o comparație mai tangibilă a întârzierilor celor 2 sumatoare (ripple-carry și carry-lookahead), puteți folosi aceste 2 simulatoare cu întârziere setabilă pentru porțile folosite (se pot observa întârzieri semnificative ale lui ripple-carry față de carry-lookahead pentru numere reprezentabile pe 32 de biți):

\* [[http://www.ecs.umass.edu/ece/koren/arith/simulator/Add/ripple/ripple.html|Ripple-carry]]

\* [[http://www.ecs.umass.edu/ece/koren/arith/simulator/Add/lookahead/|Carry-lookahead]] ==== 3. Detalii de implementare ====

Pentru fiecare rang dintr-o secvență binară, sumatorul va determina dacă perechea de biți care trebuie adunată poate genera sau propaga carry. Acest lucru permite circuitului sa “pre-proceseze” cei doi termeni ai adunării pentru a determina transportul înainte de a efectua adunarea propriu-zisă. Apoi, când aceasta are loc, nu va exista nici o întârziere de propagare a carry-ului, ca în cazul unui Ripple Adder. Mai jos aveți un exemplu de calcul al termenilor de propagare și, generare pentru un sumator de 2 biți:

\* c<sub>1</sub> = G<sub>0</sub> | P<sub>0</sub> & c<sub>0</sub>

\* c<sub>2</sub> = G<sub>1</sub> | P<sub>1</sub> & c<sub>1</sub>

Înlocuind c<sub>1</sub> în expresia lui c<sub>2</sub>, vom obtine urmatoarele ecuații:

\* c<sub>1</sub> = G<sub>0</sub> | P<sub>0</sub> & c<sub>0</sub>

\* c<sub>2</sub> = G<sub>1</sub> | P<sub>1</sub> & G<sub>0</sub> | P<sub>1</sub> & P<sub>0</sub> & c<sub>0</sub>

Similar, putem generaliza pentru un sumator de 4 biți:

\* c<sub>3</sub> = G<sub>2</sub> | P<sub>2</sub> & G<sub>1</sub> | P<sub>2</sub> & P<sub>1</sub> & G<sub>0</sub> | P<sub>2</sub> & P<sub>1</sub> & P<sub>0</sub> & c<sub>0</sub>

\* c<sub>4</sub> = G<sub>3</sub> | P<sub>3</sub> & G<sub>2</sub> | P<sub>3</sub> & P<sub>2</sub> & G<sub>1</sub> | P<sub>3</sub> & P<sub>2</sub> & P<sub>1</sub> & G<sub>0</sub> | P<sub>3</sub> & P<sub>2</sub> & P<sub>1</sub> & P<sub>0</sub> & c<sub>0</sub>

Putem observa că pentru un sumator \*\*carry-lookahead\*\* cu 4 biți (cel de mai sus), propagarea semnalului de carry până la ieșirea ultimului nivel de adunare necesită trecerea prin 3 porți logice, pe când un sumator ripple-carry necesită trecerea prin 6 porți.

{{ cn1:laboratoare:07:adders\_comparison.jpg?600x600 | Ripple-Carry vs. Carry-Lookahead }}

===== TL;DR =====

Pentru fiecare rang dintr-o secvență binară, sumatorul va determina dacă perechea de biți care trebuie adunată poate genera sau propaga carry. Acest lucru permite circuitului sa “pre-proceseze” cei doi termeni ai adunării pentru a determina transportul înainte de a efectua adunarea propriu-zisă. Apoi, când aceasta are loc, nu va exista nici o întârziere de propagare a carry-ului, ca în cazul unui Ripple Adder. Mai jos aveți un exemplu de calcul al termenilor de propagare și, generare pentru un sumator de 2 biți:

\* c<sub>1</sub> = G<sub>0</sub> | P<sub>0</sub> & c<sub>0</sub>

\* c<sub>2</sub> = G<sub>1</sub> | P<sub>1</sub> & c<sub>1</sub>

Înlocuind c<sub>1</sub> în expresia lui c<sub>2</sub>, vom obtine urmatoarele ecuații:

\* c<sub>1</sub> = G<sub>0</sub> | P<sub>0</sub> & c<sub>0</sub>

\* c<sub>2</sub> = G<sub>1</sub> | P<sub>1</sub> & G<sub>0</sub> | P<sub>1</sub> & P<sub>0</sub> & c<sub>0</sub>

Similar, putem generaliza pentru un sumator de 4 biți:

\* c<sub>3</sub> = G<sub>2</sub> | P<sub>2</sub> & G<sub>1</sub> | P<sub>2</sub> & P<sub>1</sub> & G<sub>0</sub> | P<sub>2</sub> & P<sub>1</sub> & P<sub>0</sub> & c<sub>0</sub>

\* c<sub>4</sub> = G<sub>3</sub> | P<sub>3</sub> & G<sub>2</sub> | P<sub>3</sub> & P<sub>2</sub> & G<sub>1</sub> | P<sub>3</sub> & P<sub>2</sub> & P<sub>1</sub> & G<sub>0</sub> | P<sub>3</sub> & P<sub>2</sub> & P<sub>1</sub> & P<sub>0</sub> & c<sub>0</sub>

Putem observa că pentru un sumator \*\*carry-lookahead\*\* cu 4 biți (cel de mai sus), propagarea semnalului de carry până la ieșirea ultimului nivel de adunare necesită trecerea prin 3 porți logice, pe când un sumator ripple-carry necesită trecerea prin 6 porți.

{{ cn1:laboratoare:07:adders\_comparison.jpg?600x600 | Ripple-Carry vs. Carry-Lookahead }}

===== Exerciții =====

Pentru fiecare dintre exerciții va trebui să faceți atât \*\*implementarea\*\* cât și \*\*simularea\*\* cu seturi de date relevante. (\*\*minim 3 cazuri de test\*\*)

\*\*Task 01\*\* (3p) Implementați un sumator carry-lookahead pe 4 biți.

\*\*Task 02\*\* (1p) Implementați un scăzător carry-lookahead pe 4 biți, pornind de la punctul precedent.

\*\*Task 03\*\* (2.5p) Implementați un sumator pe 16 de biți folosind modulul carry-lookahead deja implementat și logica sumatorului ripple-carry.

\*\*Task 04\*\* (1.5p) Implementați un sumator ripple-carry pe 4 biți folosind modulul \*\*full\_adder\*\* dat.

\*\*Task 05\*\* (2p) Găsiți 2 teste relevante în care să fie vizibilă optimizarea adusă de sumatorul carry-lookahead față de cel ripple-carry.

<note>

Creați module de test pentru //cla\_adder// și //rpc\_adder//, apoi comparați simulările obținute.

Ce se întâmplă cu semnalele: //sum// și //c\_out//?

</note>

<hidden>

\*\*Task 03\*\* (4p) Implementati un sumator/scazator carry-lookahead pe 4 biti cu ajutorul [[http://www.digilentinc.com/Data/Products/NEXYS/Nexys\_rm.pdf|placii de laborator]]. Interactiunea cu modulul se va face astfel:

\* Intrari:

\* operandul A -> 4 switch-uri

\* operandul B -> 4 switch-uri

\* buton1 ->buton pentru validarea operanzilor

\* buton2 -> buton pentru operatia de adunare

\* buton3 -> buton pentru operatia de scadere

\* buton reset

\* Iesiri:

\* cele 8 leduri -> pentru afisarea operanzilor

\* afisajul cu 7 segmente -> pentru afisarea rezultatului

\* Mod de operare:

\* trebuie intodusi operanzii cu ajutorul switch-urilor si afisari cu ajutorul led-urilor(cate 4 pentru fiecare operand

\* pentru a valida opernazii trebuie apasat butonul "buton1"

\* dupa validare:

\* la apasarea butonul "buton2" se va afisa cu ajutorul afisajului cu 7 segmente rezultatul adunarii celor doi operanzi

\* la apasarea butonul "buton3" se va afisa cu ajutorul afisajului cu 7 segmente rezultatul adunarii celor doi operanzi

\* dupa apasarea unui buton, va ramane afisata aceeasi valoare pana la apasarea urmatorului buton

\* pentru introducerea altor valori pentru operanzi se va apasa butonul de reset

\* pentru a realiza operațiile veți folosi modulele create la exercițiile anterioare

</hidden>

===== Resurse =====

\* {{:cn1:laboratoare:07:lab7bis\_skel.zip|Scheletul de laborator}}