Nume: Marin Vanessa-Ramona

Implementarea modulului **sensors_input**:

Am verificat daca cel putin un senzor dintr-o pereche este egal cu 0, atunci nici perechea lui nu va fi luata in considerare in calculul sumei. Asadar, pentru a obtine inaltimea la care se afla aeronava, am impartit suma la numarul de senzori luati in considerare, adica am facut media aritmetica. Deoarece ne dorim ca rezultatul final sa fie la cel mai apropiat numar intreg, vom aduna la suma inaltimilor funizate de senzori 1. Acest artificiu are loc pentru ca Verilogul, aproximeaza numerele cu X.5 in jos.

Spre exemplu: 5/2 = 2.5 => dpdv matematic, aproximam la 3, dar verilogul imi aproximeaza la 2. Pentru a rezolva aceasta problema, m-am gandit sa urmaresc cand obtin numere cu 0.5.

2/2 = 1 -> aproximat in Verilog => 1

3/2 = 1,5; -> aproximat in Verilog => 1, dar imi doresc 2

4/2 =2 -> aproximat in Verilog => 2

5/2 =2,5 -> aproximat in Verilog => 2, dar imi doresc 3

Numerele care contin 0.5, sunt rezultate atunci cand impartirea la 2 are restu' 1.

Deci am adunat +1 la acestea, tinand cont de cum rotunjeste Verilogul in spate:

(2+1)/2 = 1,5 -> aproximat in Verilog => 1

(3+1)/2 = 2 -> aproximat in Verilog => 2

(4+1)/2 = 2,5 -> aproximat in Verilog => 2

(5+1)/2 = 3 -> aproximat in Verilog => 3

Apoi am luat cazul in care toti senzorii sunt disponibili. Iar la suma inaltimilor furnizate de acestia am adaugat 2.

Am ales sa adaug 2 la suma inaltimilor furnizate de senzori, luandu-mi pe foaie cateva numere pe care le-am impartit la 4 si am comparat cu rezultatul pe care mi-l ofera Verilog-ul.

4/4 = 1

5/4 = 1,25 -> aproximat in Verilog => 1

6/4 = 1,5 -> aproximat in Verilog => 1, dar imi doresc 2

7/4 = 1,75 -> aproximat in Verilog => 1, dar imi doresc 2

Deci am adunat +2 la acestea, tinand cont de cum rotunjeste Verilogul:

(4+2)/4 = 1.5 -> aproximat in Verilog => 1

(5+2) / 4 -> aproximat in Verilog => 1

(6+2)/4 -> aproximat in Verilog => 2

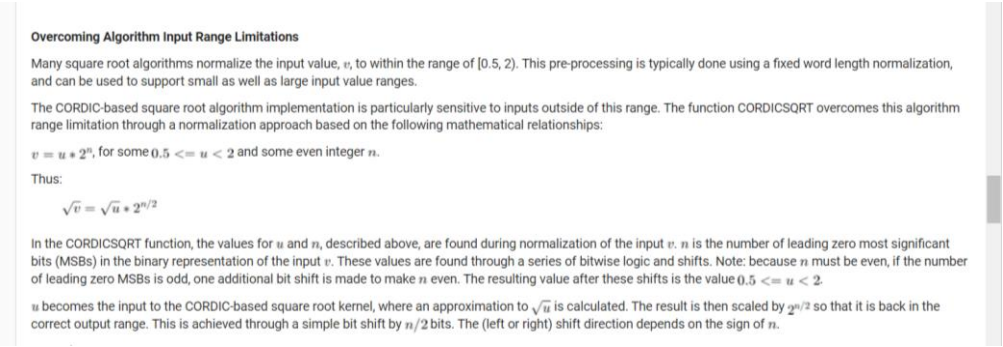
(7+2)/4 -> aproximat in Verilog => 2

Implementarea modulului **square_root**:

Am folosit Algoritmul CORDIC, oferit in resursele din cerinta temei.

In exemplu din resurse si-au dorit o precizie de 8, dar pentru cazul meu imi doresc una de 16, deci

L meu va fi egal cu 15, iar baza va fi egala cu 2 la puterea 15.



Overcoming Algorithm Input Range Limitations

Many square root algorithms normalize the input value, v, to within the range of [0.5, 2). This pre-processing is typically done using a fixed word length normalization, and can be used to support small as well as large input value ranges.

The CORDIC-based square root algorithm implementation is particularly sensitive to inputs outside of this range. The function CORDICSQRT overcomes this algorithm range limitation through a normalization approach based on the following mathematical relationships:

$v = u * 2^n$, for some $0.5 <= u < 2$ and some even integer n.

Thus:

$$\sqrt{v} = \sqrt{u} * 2^{n/2}$$

In the CORDICSQRT function, the values for u and n, described above, are found during normalization of the input v. n is the number of leading zero most significant bits (MSBs) in the binary representation of the input v. These values are found through a series of bitwise logic and shifts. Note: because n must be even, if the number of leading zero MSBs is odd, one additional bit shift is made to make n even. The resulting value after these shifts is the value $0.5 <= u < 2$.

u becomes the input to the CORDIC-based square root kernel, where an approximation to $\sqrt{u}$ is calculated. The result is then scaled by $2^{n/2}$ so that it is back in the correct output range. This is achieved through a simple bit shift by $n/2$ bits. The (left or right) shift direction depends on the sign of n.

Deci avand in vedere, documentatia disponibila in matlab Help Center, aflam ca intervalul in care se afla inputul este foarte sensibil si il modificam corespunzator. (mi-am modificat inputul inmultindu-l cu 2 la puterea n).

De asemenea, observam din exemplu aflat la resurse + putin calcul ca inputul este reprezentat pe 14 biti, iar outputul este reprezentat pe 7 biti.

Pentru ca am modificat inputul, am avut nevoie sa-i maresc si reprezentarea acestuia, asa ca mi-am declarant un nou input pe 32 de biti, in care am salvat inputul modificat.

Am inceput sa rulez algoritmul pentru inputul meu initial inmultit cu diferite puteri a lui 2. Deoarece, marind inputul conditia de if va avea mai multe raspunsuri fluctuante, outputul va creste si el.

Astfel, am inmultesc inputul cu 2 la puterea 16, 16 fiind precizia pe care mi-o doresc.

Luam exemplul din resurse: 139

139 pe 8 biti = 1000_1011

Dupa ce il inmultim cu 2^16:

139 pe 32 biti = 1000_1011_0000_0000_0000_0000, deci prin inmultirea cu 2^16, nu facem altceva decat sa shiftam stanga cu 16 pozitii. Astfel, in output ajungem sa avem si partea fractionala, care va fi delimitata ulterior.

Aplic CORDIC:

L (precizia)	2^L (baza)	output	input=	139 *2^16
15	32768	32768	32768x32768 > in	Do nothing

14	16384	16384	16384x16384 > in	Do nothing
13	8192	8192	8192x8192 > in	Do nothing
12	4096	4096	4096x4096 > in	Do nothing
11	2048	2048	2048x2048 < in	+ 2048 to output
10	1024	2048	(1024+2048)^2 > in	Do nothing
9	512	2048	(512+2048)^2 < in	+ 512 to output
8	256	2560	(256+2560)^2 < in	+256 to output
7	128	2816	(2816+128)^2 < in	+128 to output
6	64	2944	(2944+64)^2 < in	+64 to output
5	32	3008	(3008+32)^2 > in	Do nothing
4	16	3008	(3008+16)^2 >in	Do nothing
3	8	3008	(3008+8)^2 < in	+8 to output
2	4	3016	(3016+4)^2 > in	Do nothing
1	2	3016	(3016+2)^2 < in	+2 to output
0	1	3018	(3018+1) ^2 > in	Do nothing

Deci obtinem ca output = 0000_1011_1100_1010

In concluzie, daca marim precizia -> marim si baza, iar pentru a obtine un raspuns dorit, trebuie sa marim si inputul odata cu baza.

Ideea succinta: Observand in exemplu din resurse faptul ca outputul(109) este reprezentat pe 7 biti, iar inputul(12056) pe 14 biti. Pentru cazul din tema in care stiu ca am un output pe 16 biti, trebuie sa fac inputul pe 32 de biti, deci shiftez inputul inainte de a face algoritmul cu 16 pozitii la stanga, adica inmultesc inputul cu 2 la puterea 16, iar dupa parcurgem algoritmul CORDIC.

Implementarea modulului **display_and_drop**:

Am verificat cele 3 cazuri pe rand:

1. **drop_en = 0** , nu este activata posibilitatea lansarii si **t_act < t_lim**, timpul current posibil de coborare a pachetului este mai mic decat timpul limita de coborare a pachetului
2. **drop_en = 0** , nu este activate posibilitatea lansarii si **t_act > t_lim,** timpul current posibil de coborare a pachetului este mai mare decat timpul limita de coborare a pachetului
3. **drop_en = 1** , este activata posibilitatea lansarii si **t_act > t_lim**, timpul current posibil de coborare a pachetului este mai mic sau egal decat timpul limita de coborare a pachetului

Pentru fiecare caz in parte am afisat un mesaj corespunzator, folosindu-ma de 4 module 7Seg. Pentru reprezentarea corecta a mesajului, m-am folosit de Anexa din Laboratorul 3 (Seven-segment display). De asemenea, pe langa mesaj, am furnizat si informatia conform careia semnalul de alarma este activat sau nu (1 sau 0).

Implementarea top modulului **baggage_drop**:

Am urmarit schema de implementare furnizata pe site-ul temei.

Am observant ca legatura intre modulele din interiorul top modulului se face folosir wire-uri.

Astfel, in top module mi-am declarat cate un wire pentru fiecare legatura:

- **my_height** -> legatura dintre iesirea lui sensors_input si intrarea lui square_root
- **my_sqrt** -> legatura dintre iesirea lui square_root , acestei legaturi trebuie sa-i facem o prelucrarea intermediara, deci mi-am mai luat un wire **out_my_sqrt**, caruia ii dau valoarea lui **my_sqrt** prelucrata. Acum **out_my_sqrt** va face legatura dintre valoarea prelucrata a iesirii lui square_root si intrarea modului display_and_drop

De asemenea, mi-am instatiat fiecare modul implementat anterior, in top module.

In final, am folosit instantele si legaturile(wire-urile) pentru a-mi construi intreg circuitul conform schemei din cerinta temei.