Boriceanu Ioana-Roxana, grupa 331AB

Voi încerca să descriu în continuare modul de implementare al fiecărui modul.

Modulul sensors_input:
Implementarea acestui modul a fost relativ simplă, am creeat două variabile de tip reg pe 17 biti pentru a calcula suma valorilor celor 4 senzori și media(înălțimea). Motivul pentru care le-am făcut pe 17 biți și nu pe 16 e pentru că trebuia luat în considerare și bitul de carry.

O observație importantă la acest modul e că atunci când am calculat suma a trebuit să adun 1 sau 2 pentru că altfel nu-mi trecea testele fiindcă nu făcea aproximare prin adaos, ci doar prin lipsă. În felul ăsta dacă suma noastră era 91 de exemplu, rezultatul mediei, după împărțirea la 2, ar fi 45.5, deci aproximarea trebuia făcută la 46, Verilog nu face asta, din păcate, și atunci am mai adunat acel 1 la sumă, în cazul în care luam în considerare doar doi senzori(sau 2 în cazul în care luăm în considerare 4 senzori), ca rezultatul să fie 92, iar media să fie 46 așa cum trebuia. Ca să mă asigur că nu le stric pe cele la care trebuia făcută aproximare prin lipsă am efectuat aceeași operație și am observat că de obicei dă ceva cu 0.5 și îl ignoră, deci este în regulă.

Modulul display_and_drop

Aici nu sunt foarte multe de explicat, am folosit un bloc always@ în care am verificat toate situațiile descrise în cerința temei și am afișat mesajele corespunzătoare pe cele 4 display-uri ținând cont că variabilele sunt definite cu [6:0], nu [0:6] ca să scriu corect reprezentările pe cei 7 biți ale caracterelor.

Modulul baggage_drop

Aici am creeat legăturile dintre celelalte trei module și am mai făcut o operație suplimentară pentru a-l calcula pe t_act după formula t_act=sqrt(height)/2, am folosit operatorul de shiftare la dreapta în locul operatorului de împărțire deoarece dă același rezultat și am vrut să păstrez pattern-ul din restul temei(pattern-ul fiind că am încercat să folosesc peste tot operatorul de shiftare în loc de ‚/’).

Modulul square_root

A fost amuzant de implementat, dar nu mai vreau. Aici am încercat mai multe metode de calcul și până la urmă am rămas la CORDIC pentru că aveam deja o bucățică de cod(aka tot algoritmul) în resursele date. Marea provocare aici a fost să îmi dau seama cum anume fac rezultatul să fie reprezentat în virgulă fixă așa cum era specificat în cerință. 3 zile mai târziu, s-a întâmplat un miracol și a trecut toate testele.

Aici am început prin a implementa algoritmul exact ca pe site în încercarea de a înțelege ce efect are asupra valorilor noastre și am observat că el primește ca date de intare valoarea pe care aplicăm algoritmul, în cazul nostru in, și o valoare base, care în cazul nostru este 2^(n-1), unde n este numărul de biți pe care este reprezentată variabila in, iar ca date de ieșire îl are doar pe out pe care îl inițializăm la începutul programului(al blocului always@) cu 0.

După care, într-o structură repetitivă cu număr fix de pași, calculăm valoarea lui out. La fiecare iterație se adaugă în out valoarea lui base și se verifică dacă pătratul valorii obținute este mai mare decât valoarea inițială, dacă da, atunci scădem din out valoarea lui base pentru că înseamnă că am obținut o valoare care nu e rădăcina pătrată a lui in, și se trece mai departe la shiftarea lui base cu 1, care este echivalentă cu împărțirea la 2. De ce am ales base=128? Mi-am dat seama, după multe încercări eșuate, că dacă avem 8 biți de 1 de exemplu 11111111 asta e echivalent cu 2^0+2^1+2^2+2^3+2^4+2^5+2^6+2^7. Deci, în reprezentarea pe 8 biți , bitul cel mai din stânga e echivalent cu 2^7, și noi practic în algoritm nu facem altceva decât să modificăm biți în 0 sau 1 ca să ajungem la valoarea dorită. Adică, la nivel de bit în algoritm pe out=0000_0000 îl adunăm cu base=128=1000_0000 și o să devină out=1000_0000, verificăm condiția și decidem dacă revenim la valoarea inițială sau rămânem la aceasta, base este shiftat cu 1, ceea ce este echivalent cu 128/2=64=2^6 și la următoarea iterație se repetă procedura de mai sus, out=out+base ⇔ out=1000_0000 +0100_0000= 1100_0000, out ia noua valoare se verifică condiția etc etc. Practic, modificăm câte un bit din 0 în 1, de la dreapta la stânga, sau nu-l modificăm, la fiecare iterație.

În cazul meu, primul for calculează valoarea primilor 8 biți, iar al doilea for valoarea ultimilor 8 biți. Pentru al doilea for, am avut nevoie de două variabile auxiliare, aux pe 24 de biți și o variabilă in2 pe care am folosit-o să păstrăm variabila in shiftată la stânga cu 8 biți. Algoritmul e exact la fel, adică modifică pe rând fiecare din cei 8 biți „de după virgulă”. Problema aici e că noi când o să calculăm pătratul lui out el o să fie pe 24 de biți nu pe maxim 16 ca înainte și nu îl putem compara cu in. Deci luam in2 și îl egalăm cu in shiftat la stânga cu 8, un mic artificiu să trecem de la in pe 8 biți la un in pe 16 biți. În for pe lângă operațiile de mai sus, de data asta îl punem out*out în aux și îl shiftăm cu 8. Shiftarea o facem ca să aducem rezultatul de pe biții [23:8] pe biții [15:0] și mai apoi să putem compara rezultatul obținut cu in2 care e și el la rândul lui pe 16 biți. Restul calculelor au loc în out la fel ca la primul for, pe base îl reinițializăm cu 128 înainte de a începe calculele și îl shiftăm la fiecare iterație.

Numărul de iterații este egal cu numărul de biți pe care trebuie să îi modificăm în fiecare etapă, adică 8.