Documentație proiect - !Shazam

Contribuție proiect:

- Bodrogean Nadira: fixed point arithmetic system, Radix 2 butterfly, detectorul de maxime
- Ghimpău Mihai-Vladimir: PISO, FIFO, SPI, Arduino
- Rață Sorin-Gabriel: coordonator, ADC, FFT input controller, calcularea magnitudinilor, python training/recognizing

1. Bodrogean Nadira

După ce am descoperit un modul de FFT (yoonisi/R2FFT) care demonstra corectitudine teoretică prin testele realizate cu ajutorul simulărilor funcționale în QuestaSIM, am cotribuit la portarea modulului matematic din virgulă flotantă în virgulă fixă și în înlocuirea semnalului *BFP_EXP* (care ar trebui să reprezinte exponentul calculat dinamic la fiecare pas) cu un exponent static prin rescrierea modulului Radix 2 butterfly (unde am învățat de asemenea utilitatea unei arithecturi de tip *pipeline*)

Pentru început cele două module matematice importante sunt:

- *FIXED_POINT_ADDER*
- *FIXED_POINT_MULTIPLIER*

Operațiile sunt effectuate în complement de 2 (deci scăderea reprezintă doar negare și adunare cu 1), iar împărțirea poate fi implementată prin înmulțire cu fracții: 0.5, 0.25, 0.125, etc.

1. FIXED_POINT_ADDER

Modulul pentru adunare folosește operatorul sintetizabil + și salvează rezultatul unei adunări a două numere pe 16 biți într-un număr pe 17 biți (16 biți *temp_sum* și 1 bit *extra*). Folosind algoritmul de detecție a overflow-ului descris aici: Signed multiplication overflow detection in Verilog - Stack Overflow, vom trunca rezultatul superior sau inferior. Arhitectura presupune un *pipeline* cu 2 stadii:

- Începerea este dată când semnalul *enable este HIGH*, moment în care se realizează adunarea pe 17 biți
- După un clock semnalul *compute_sum_and_overflow* este *HIGH*, iar algoritmul va trunca rezultatul dacă depășește valorile maxime pe 16 biți
- După încă un clock, rezultatul pe ieșirea *sum* este corect, iar semnalul *done* este *HIGH* indicând faptul că output-ul va fi mai fi corect timp de 1 clock

Arhitectura este extrem de sensibilă la semnalul *enable*. Dacă acesta devine *HIGH* incorect va provoca o cascadă de valori greșite și poate compromite extrem de ușor sistemul. Testul pentru acest modul poate fi găsit aici: Fixed Point Adder - functional simulation, iar codul este disponibil aici: Fixed Point Adder - code.

2. FIXED_POINT_MULTIPLIER

Modulul pentru înmulțtire folosește operatorul sintetizabil * și salvează rezultatul înmulțirii a două numere pe 16 biți într-un număr pe 32 de biți. Folosind o notație standard

pentru a reprezenta numerele din virgulă fixă: Q n.m (unde *n* este numărul de biți pentru partea întreagă, iar m este numărul de biți pentru partea fracționară), înmulțirea:
Q n.m * Q a.b va produce un număr de 32 de biți de forma Q n+a.m+b, iar virgula se află acum la poziția „n + a" de la dreapta la stânga, iar numărul începe de la poziția „n" și se termină la poziția „n + 16". Pentru detectarea overflow-ului, se va lua noul număr (de la poziția „n" la poziția „n + 16") și se va verifica dacă MSB respectă logica reală:

- dacă Q1 – pozitiv și Q2 – pozitiv, atunci MSB (Q1 * Q2) = 1 reprezintă overflow, iar MSB (Q1 * Q2) = 0 este corect
- dacă Q1 – negativ și Q2 – negativ, atunci MSB (Q1 * Q2) = 1 reprezintă overflow, iar MSB (Q1 * Q2) = 0 este corect
- dacă Q1 – negativ/pozitiv și Q2 – pozitiv/negativ, atunci MSB (Q1 * Q2) = 0 reprezintă overflow, iar MSB(Q1 * Q2) = 1 este corect

Arhitectura este de asemenea bazată pe un *pipeline* cu două etape, care începe cu semnalul *enable* pe *HIGH*, iar rezultatul corect va apărea pe wire după două perioade de clk, alături de semnalul *done* care va fi *HIGH* o perioadă. Testul pentru acest modul poate fi găsit aici: Fixed Point Multiplier - functional test, iar codul este disponibil aici: Fixed Point Multiplier - code.

3. RADIX 2 BUTTERFLY

Pentru a evita nevoia de a calcula poziția virgulei în mod dinamic (prin intermediul parametrului *BFP_EXP*) vom exploata faptul că biții proveniți de la ADC pot fi reinterpretați. Butterfly-ul conceput de creator folosea algoritmul DIF (Decimation in Frequency) pentru implementarea operațiilor, astfel:

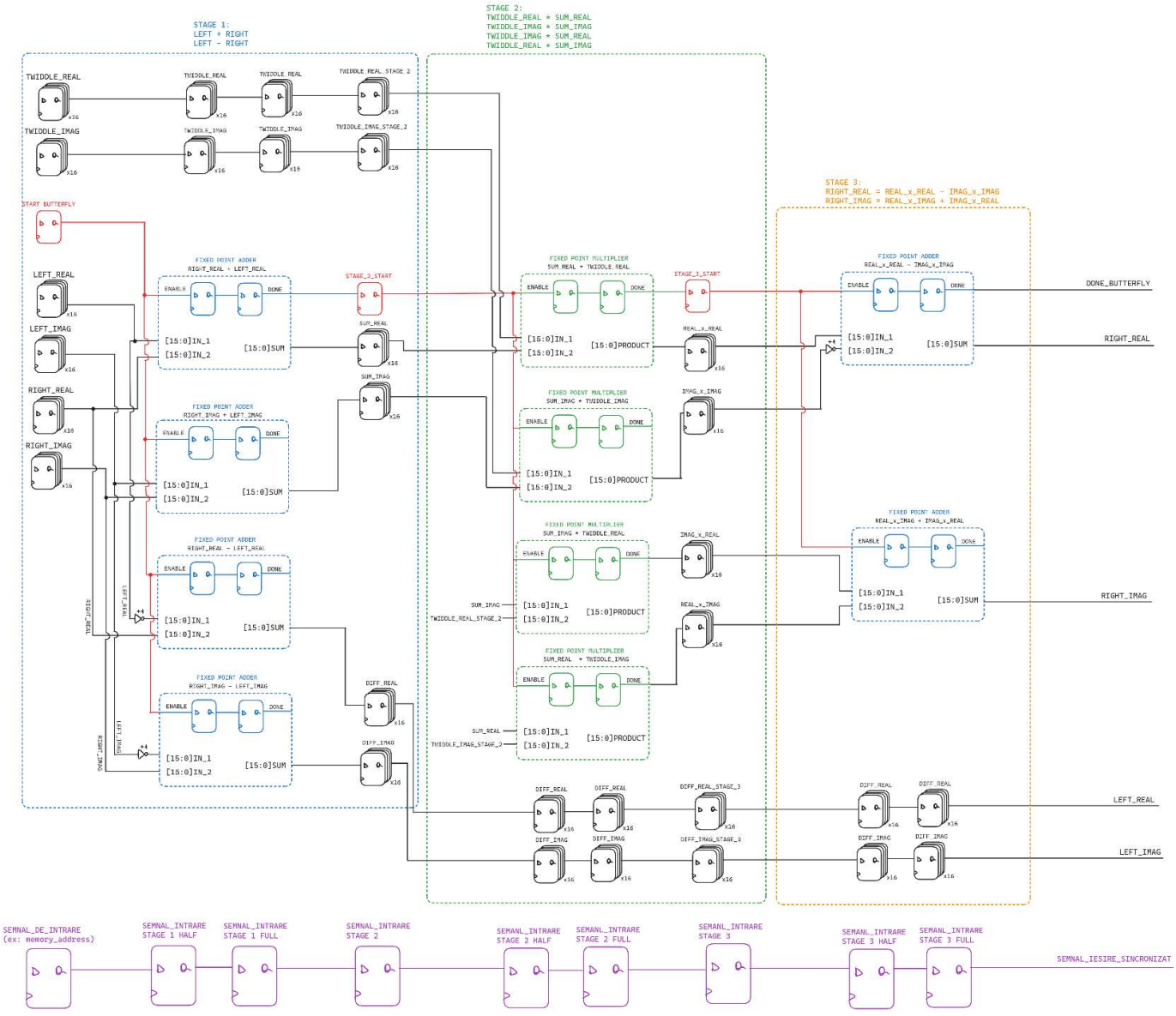
```
assign new_left = left + right;
assign new_right = (left - right) * twiddle;
```

Știind că *twiddle* ∈ [0;1] putem demonsta ușor că, dacă considerăm *left* și *right* inițiale ca aparținând [0;1], valoarea maximă a lui *new_left* și *new_right* poate fi doar între [0; 2^{level}], unde *level* reprezintă nivelul curent din cadrul FFT-ului, iar înmulțirile pot considera că:

- la primul nivel numerele sunt în format *Q2.14 * Q.1.15 -> Q2.14*
- la al doilea nivel în format *Q3.13 * Q1.15* -> Q3.13
- ...
- iar la al zecelea nivel sunt în format *Q12.4 * Q1.15 –> Q12.4*

unde primul operand este (*left – right)*, al doilea este *twiddle*, iar rezultatul este *new_right*. Deoarece avem de aface cu numere complexe, trebuie să înmulțim independent părțile reale, cele imaginare, apoi real cu imaginar și imaginar cu real, iar la final scădem (respectiv adunăm) rezultatele pentru a obține partea reală și cea imaginară a lui *new_right*. Ca și la modulele de adunare/înmulțire, vom pune într-un *pipeline* toate semnalele ce țin de infrastructură (de exemplu semnalele de activare, pozițiile în RAM, etc.) pentru ca la finalul modulului, toate semnalele de ieșire să corespundă celor inițiale, ținând cont de timpul de procesare pentru operațiile de adunare și înmulțire. Testul pentru radix2butterfly se poate găsi aici: Radix2Butterfly – functional test

Mai jos este atașată diagrama bloc pentru Radix 2 Butterfly, unde se observă design-ul de tip *pipeline*:



4. DETECTORUL DE MAXIME

Detectorul de maxime folosește tot o arhitectură de tip pipeline pentru a înjumătății la fiecare pas numărul de elemente al unui array de magnitudini, folosind o funcție trivială de comparare:

```
module comparator(
  input reset,
  input [24:0] a,
  input [24:0] b,
  output reg [24:0] max
);

// we are using unsigned numbers:
always_comb begin
  if(reset) max = 0;
  else begin
    max = (a[15:0] > b[15:0]) ? a : b;
  end
end

endmodule
```

Numerele sunt pe 25 de biți deoarece cei mai semnificativi 9 biți (24:16) reprezintă valoarea frecvența în sine (ex: 14(Hz), 155(Hz), 343(Hz), etc), iar următorii 16 biți (15:0) reprezintă magnitudinea acestei frecvențe (ex: 100023, 1454452, 313131345, etc.). Noi comparăm magnitudinile, dar vrem să știm și ce frecvență a avut magnitudinea maximă.

Pentru a reduce numărul de elemente la fiecare pas, folosim un modul de „reduction” care primește un array cu *SIZE* elemente și returnează unul cu *SIZE/2* cele mai mari elemente, folosind o construcție de tipul *generate for* pentru a evita instanțierea manuală a moduleleor de comparare:

```
module reduction #(parameter SIZE = 511)(
  input reset,
  input [24:0] in [SIZE - 1: 0],
  output [24:0] out [SIZE / 2 - 1: 0]
);

genvar i;
generate
for(i = 0; i <= SIZE/2 - 1; i = i + 1) begin: gen_comparator_stage_2
  comparator comp(
    .reset(reset),
    .a(in[i]),
    .b(in[SIZE - i - 1]),
    .max(out[i])
  );
end
endgenerate
endmodule
```

Atunci când găsim o maximă o înlocuim în array-ul inițial cu valoarea 0 pentru a nu fi detectată din nou.

```
reg [24:0] data [511:0];

find_1_peak PEAK_FINDER (
  // pipeline start trigger
  .start(start_find_peak),

  // input
  .data_in(data),

  // outputs
  .peak(peak),
  .peak_index(peak_index),

  // pipeline finished, peak found
  .output_active(peak_output_active)
);

if(peak_output_active) begin
  data[peak_index] <= {25{1'b0}};
end
```

Testul pentu acest modul poate fi găsit aici: Comparator - functional test, iar codul aici: Comparator - code

2. Ghimpău Mihai-Vladimir

Segmentul de care am fost responsabil în proiect este trimiterea datelor de la modulul FFT la Arduino, prin intermediul protocolului SPI (Serial Peripheral Interface).

Modulul SPI gestioneaza transferul de date intre un master si unul sau mai multe dispozitive slave prin intermediul a 3 iesiri si o intrare:

1. *SCLK* (Serial Clock - output): clock-ul serial generat de master pentru a sincroniza transferul de date
2. *MOSI* (Master Out Slave In - output): linia pe care master-ul trimite date către slave
3. *MISO* (Master In Slave Out - input): linia pe care slave-ul trimite date către master
4. *CS* (Chip Select – output): linia folosită de master pentru a selecta un dispozitiv slave specific.

Un detaliu important in funcționarea corectă a transferului de date este că linia MOSI funcționează pe tactul negativ al SCLK-ului.

În aceste condiții am considerat ca date de intrare:

- *SCLK* este un clock de 4 MHz generat cu un PLL (frecventa de lucru optimă pentru Arduino: How do you use SPI on an Arduino? - Arduino Stack Exchange)
- *RESET* este un reset universal
- *[8:0] DATA_IN* reprezintă magnitudinea unei frecvențe semnificative păstrate în FIFO
- *FIFO_EMPTY* este un semnal care împiedică citirea dintr-o coadă goală (care produce rezultate impredictibile)

Ca date de iesire avem:

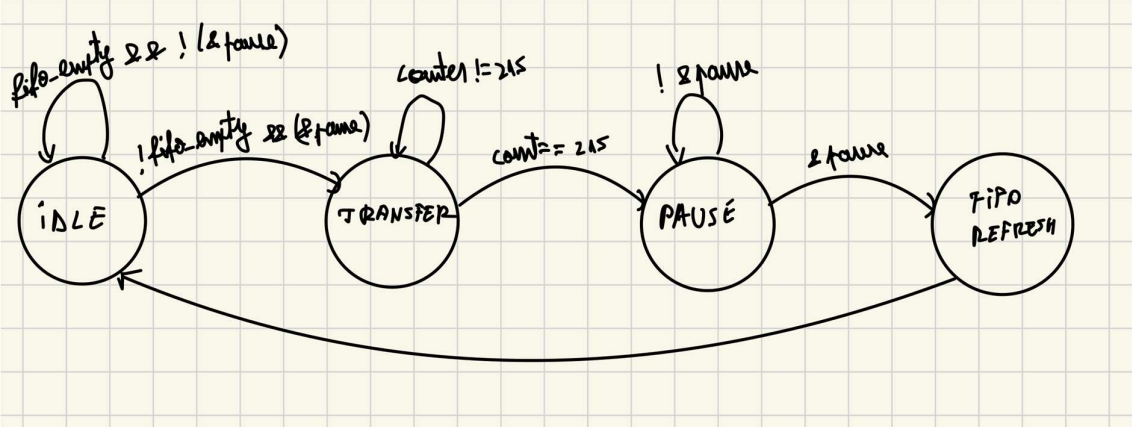
- *FIFO_REFRESH_DATA* este un semnal care anunță că ultima magnitudine a fost trimisă cu succes (cei mai semnificativi 8 biți au ajuns la Arduino), semnalând disponibilitatea modulului SPI de a trimite o nouă frecvență
- *MOSI* este semnalul prin care se transmit datele serial
- *CS* este semnalul prin care activăm slave-ul (Arduino)

Detalii legate de functionarea modulului (FSM-ul modulului):

- Starea *IDLE* este starea în care modulul așteaptă să primeasca date pentru a începe transferul. În această stare, se resetează contorul de biți, se dezactivează refresh-ul FIFO-ului, se resetează outputul *MOSI*, se dezactivează *CS* și se citesc date de la FIFO în registru
- În starea *TRANSFER* se transmit serial datele la Arduino prin *MOSI*. Aici se resetează pauza, se activează CS, și se contorizează numărul de biți trimiși.
- Starea *PAUSE* este o folosită pentru a așepta ca Arduino să trimită datele mai departe spre Python prin afișarea frecvenței în Serial Console (operație care durează 6 µs: worst case scenario, ceea ce înseamnă aproximativ 32 *SCLK*). În această stare doar se dezactivează ieșirile SPI-ului: *CS* si *MOSI*.

- Starea *FIFO_REFRESH* este ultima stare a automatului, unde se transmite mai departe sistemului că SPI-ul a transmis datele și că este disponibil pentru a citi date noi.

Diagrama automatului este atașată mai jos:



Deoarece cele mai semnificative 10 frecvențe vin sub forma unui array de numere în format binar, am implementat un modul PISO care să transforme matricea de biți într-un array de biți, care să poată fi stocat în FIFO ca un *cuvânt*:

```
// a matrix of bits, representing an array of 10 elements, numbers from 0 -> 511
wire [9:0] most_significant_magnitudes [8:0];

// an array of bits, representing a signle number from 0 -> 511
wire [8:0] single_magnitude;

// the PISO will output a number on every positive clock edge
PISO matrix_to_array (
    // ...
  .data_in(most_significant_magnitudes),
  .serial_out(single_magnitude),
   // ...
);

FIFO f (
   // ...
  .data_in(signle_magnitude),
   // ...
)
```

Ca date de intrare avem:

- *CLK* este clock-ul placii de 50 MHz
- *RESET* este reset-ul global
- *LOAD* este semnalul când datele de intrare sunt valide
- *DATA_IN* sunt numerele primite de la FFT

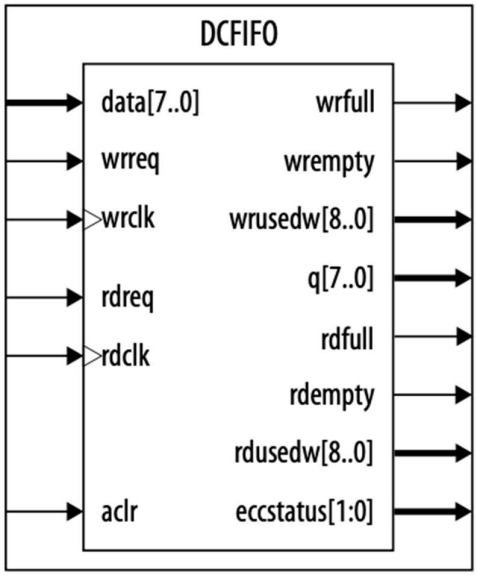
Iar ca date de iesire avem:

- *SERIAL_OUT* este ieșirea serială care intră în FIFO
- *OUTPUT_ACTIVE* este semnalul pentru încărcarea lui *SERIAL_OUT* în FIFO

Funcționarea se bazează pe un FSM simplu cu două stări: *IDLE* si *LOAD*. În *IDLE*, automatul așteaptă să primească semnalul *LOAD* pentru a începe încărcarea registrului *SERIAL_OUT*. În starea *LOAD*, se transmit frecvențele, câte una pe rând pe frontul pozitiv al *CLK*-ului.

O problemă ce apare în proiectarea circuitelor digitale este nevoia ca semnalele să traverseaza de la un domeniu de clock la altul. Acesta este fenomenul de Clock Domain Crossing (CDC). Pentru a evita pierderea datelor (atunci când trecem de la un clock mai rapid la unul mai încet) sau starea de metastabilitate se utilizează tehnici speciale precum dublarea bistabilelor sau FIFO-uri cu numărătoare Gray.

În acest design, singurul loc pentru a realiza încrucișarea domeniilor de clock este între modulele PISO și SPI. Am adus aici un Dual Clock FIFO (implementat cu numărătoare Gray) care să mențină integritatea datelor pentru a le putea transmite mai departe la Arduino (dpretet/async_fifo). Urmărind documentația, semnalele pe care le-am folosit sunt:



- Pentru READ:
  - *RDCLK* este *SCLK*-ul de 4MHz generat prin PLL,
  - *[8:0] Q* este frecvența care va fi trimisă spre Arduino
  - *RDEMPTY* este semnalul care anunță lipsa datelor de trimis
  - *RDREQ* este samnalul permite citirea primei frecvențe din FIFO
- Pentru WRITE:
  - *WRCLK* este *CLK*-ul de 50MHz al plăcii
  - *[8:0] DATA* este frecvența care va fi adăugată la finalul cozii, provenită din modulul SPI
  - *WRFULL* este semnalul care anunță imposibilitatea de a mai adăuga frecvențe în coadă
  - *WRREQ* este semnalul care permite adăugarea intrării *DATA* la finalul FIFO-ului

```
assign WRREQ = PISO_output_active && !fifo_full;
```

Testul care demonstrează corectitudinea sistemelor de trimitere a datelor (integrat cu primirea acestora de la FFT) poate fi găsit aici: PISO_FIFO_SPI - functional test, PISO_FIFO_SPI with FFT Integration - functional test

Pentru a putea trimite datele la Arduino, am legat următoarele fire:
- de la pinul *Arduino_IO13* (FPGA) la *pinul 13* (Arduino),
- de la pinul *Arduino_IO11* (FPGA) la *pinul 11* (Arduino)
- de la pinul *Arduino_IO10* (FPGA) la *pinul 10* (Arduino).

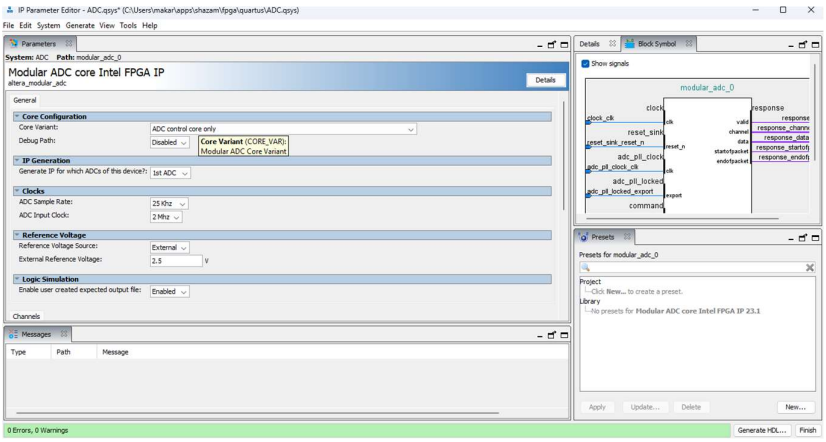
Codul Arduino citește de la MISO și scrie la consola seriala configurată cu un Baud Rate de *2000000*. Această cofigurație este esențială pentru a asigura că toate datele primite prin FPGA ajung mai departe la scriptul de python pentru a continua procesul de antrenare/recunoaștere.

3. Rață Sorin-Gabriel

Responsabilitățile pe care le-am avut au fost coordonarea coechipierilor mei și integrarea soluțiilor implementate de ei pentru a obține produsul final. Astfel, după ce ne-am documentat cu privire la algorimul din spatele proiectului original Shazam, am aflat că piesa cea mai importantă pe care o vom implementa pe FPGA este FFT-ul cu date primite de la ADC-ul plăcii FPGA, conectat la un microfon. Articolele pe care le-am citit sugerau utilizarea a două metode pentru îmbunătățirea acurateții analizei spectrale. Prima metodă este aplicarea procedeului: Window function, pentru a elemina artefactele cauzate de un semnal neperiodic (cum este cel obținut prin eșantionarea a 1024 de puncte dintr-un semnal auditiv). Însă acest procedeu produce o pierdere a informației la marginile ferestrei (Window period(overlap) and FFT - Signal Processing Stack Exchange). A doua metodă sugerată (pentru a combate insuficiențele primei metode) este de a suprapune eșantioanele astfel încât marginile unui set de date să se regăsească în mijlocul următorului set de date. Astfel nu vom pierde frecvențe care ar apărea la marginea ferestrei (un exemplu ilustrativ apare și în acest blog: Understanding Windowing And Overlapping Analysis - Noise & Vibration Blog (prosig.com)). Deși nu am reușit să implementăm procedeul de Windowing la predarea proiectului, am implementat procedeul de suprapunere cu un coeficient de 50%. Astfel, atunci când funcționalitatea de Windowing va fi implementată, vom obține o analiza spectrală îmbunătățită, fără nevoia de îngrijorare pentru o pierdere de informații.

1. ADC

ADC-ul este responsabil pentru achiziția datelor de la microfon. Am folosit un modul preexistent din catalogul Intel, pentru care am utilizat următoarele configurații:



Am ales o frecvență de eșantionare de 25k samples pentru a obține o rezoltuție decentă (12.2 Hz) în „intervalul de frecvență” (frequency bin). Deoarece sunetele sunt concepute pentru urechea umană (care poate auzi doar sunete doar până la 22.1kHz), detectarea unei frecvențe maxime de 12.5kHz, având o rezoluție de 12Hz garantează că vom putea înregistra chiar și tonalitățile de „Presence” și puțin din „Air” (care sunt frecvențe responsabile de claritate și sunetele pătrunzătoare/stridente - [The Audio Frequency Spectrum Explained | Headphonesty]), păstrând intacte calități ca vocea umană sau bass-ul. Modulul care furnizează „sample”-urile pentru FFT este *ADC_samples*, pentru care cele mai importante câmpuri sunt *adc_data* și *adc_data_valid* prin care se furnizează magnitudinea vibrației detectate și când este această magnitudine poate fi folosită.

```
module ADC_samples (
    input clk,
    input reset,
    output [11:0] adc_data,
    output adc_data_valid
);

   wire clk_2MHz_wire, clk_2MHz_locked;
   clk_2MHz clk_2MHz_INSTANCE (
       .inclk0(clk),
       .c0(clk_2MHz_wire),
       .locked(clk_2MHz_locked)
   );

   ADC ADC_INSTANCE (
      .adc_pll_clock_clk(clk_2MHz_wire),
      .adc_pll_locked_export(clk_2MHz_locked),
      .clock_clk(clk),
      .command_valid(1),
      .command_channel(1),
      .command_startofpacket(1),
      .command_endofpacket(1),
      .command_ready(),
      .reset_sink_reset_n(~reset),
     .response_valid(adc_data_valid),
      .response_channel(),
     .response_data(adc_data),
      .response_startofpacket(),
      .response_endofpacket()
   );

endmodule
```

Un test folosit pentru a verifica funcționarea modulului ADC poate fi găsit aici: [ADC show readings on FPGA HEX display].

2. Paralelizarea FFT-urilor:

Deoarece durata unui FFT este de aproximativ 2151409ns după achiziționarea datelor, lipsa unui al doilea FFT ar duce la pierderea a 53 puncte din 1024 (la un sampling rate de 25k), ceea ce ar putea afecta acuratețea sistemului. Cu toate acestea, în calculul inițial frecvența de sampling era de 50k, iar un benchmark inițial indica faptul că FFT-ul ar putea dura chiar și 20163381ns fără arhitectura de tip pipeline, ceea ce ar fi condus la pierderea a 1113 date la așteptarea completării fiecărui FFT (adică 1113 puncte pierdute la fiecare 1024 citite, data loss: 108.69%) . Pentru a combate această pieredere ne-am gândit în primă fază să utilizăm 3 FFT-uri care să ruleze în paralel (cu 50% data overlap). Astfel, am fi pierdut doar 261 puncte la fiecare 1536 citite (data loss: 17%). Așa că am instanțiat 3 module FFT:

După completare, fiecare FFT se resetează singur. Arhitectura internă a FFT-ului cere ca input-uri (pe lângă cele de sistem: clk, reset):

```
FFT_IMPLEMENTATION FFT_0 (
        .clk(clk),
        .reset(reset_FFT_0),
        .done_all_processing(FFT_0_all_done),
        .input_stream_active_i(write_active_FFT_0),
        .input_real_i({fft_input, 1'b0, 1'b0, 1'b0, 1'b0}),
        .input_imaginary_i(0),
        .index(index),
        .magnitude(magnitude_FFT_0),
        .magnitude_ready(magnitude_FFT_0_ready)
    );

    reg reset_FFT_1 = 0;
    always @(posedge clk) begin
        if(reset || !start) reset_FFT_1 <= 1;
        else if(FFT_1_all_done) begin
            reset_FFT_1 <= 1;
        end
        else begin
            reset_FFT_1 <= 0;
        end
    end

    FFT_IMPLEMENTATION FFT_1 (
        .clk(clk),
        .reset(reset_FFT_1),
        .done_all_processing(FFT_1_all_done),
        .input_stream_active_i(write_active_FFT_1),
        .input_real_i({fft_input, 1'b0, 1'b0, 1'b0, 1'b0}),
        .input_imaginary_i(0),
        .index(index),
        .magnitude(magnitude_FFT_1),
        .magnitude_ready(magnitude_FFT_1_ready)
    );


    reg reset_FFT_2 = 0;
    always @(posedge clk) begin
        if(reset || !start) reset_FFT_2 <= 1;
        else if(FFT_2_all_done) begin
            reset_FFT_2 <= 1;
        end
        else begin
            reset_FFT_2 <= 0;
        end
    end

    FFT_IMPLEMENTATION FFT_2 (
        .clk(clk),
        .reset(reset_FFT_2),
        .done_all_processing(FFT_2_all_done),
        .input_stream_active_i(write_active_FFT_2),
        .input_real_i({fft_input, 1'b0, 1'b0, 1'b0, 1'b0}),
        .input_imaginary_i(0),
        .index(index),
        .magnitude(magnitude_FFT_2),
        .magnitude_ready(magnitude_FFT_2_ready)
    );
```

- *INPUT_STREAM_ACTIVE_I*: introduce FFT-ul în starea de *READING*, până când un numărător ajunge la 1024, după care automatul intră în starea de *PROCESSING*. Este important ca acest semnal să fie readus în starea 0 atunci când ADC-ul are *adc_data_valid* inactiv, altfel output-ul FFT-ului va fi incorect.
- *INPUT_REAL_I*: magnitudea semnalului audio
- *INPUT_IMAGINARY_I:* trebuie să fie mereu 0 (nu există)

Ieșirile sunt folosite pentru a trece mai departe la comparator:
- *DONE_ALL_PROCESSING:* anunță sistemul că acest FFT este gata
- *MAGNITUDE:* trimite câte o magnitudine la sistem pentru a construi matricea *data* pentru comparator:

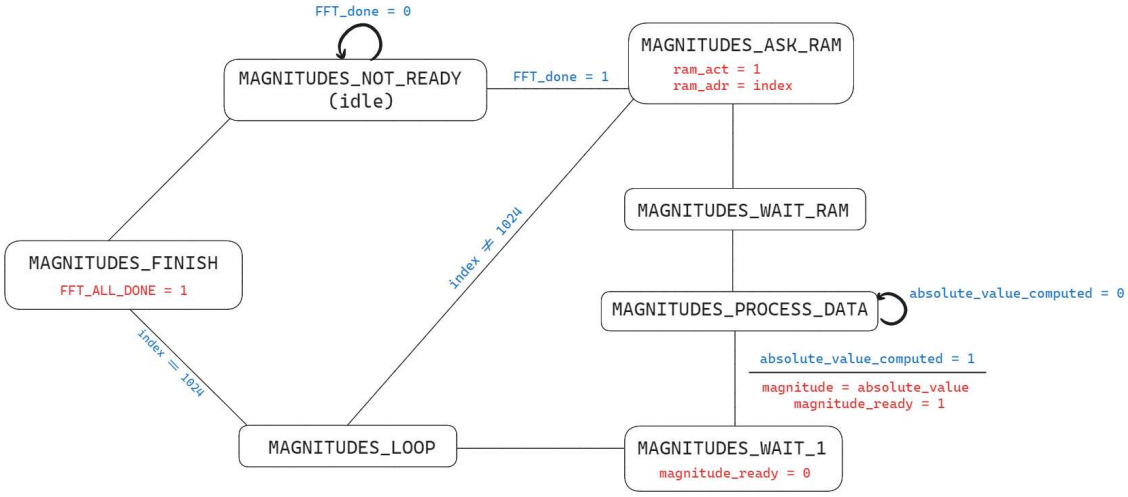
```
assign load = magnitude_FFT_0_ready | magnitude_FFT_1_ready |
magnitude_FFT_2_ready;

always @ (posedge clk) begin
  if (load) begin
      data[index] <= magnitude;
  end
end
```

- *INDEX:* generează index-ul folosit mai sus
- *MAGNITUDE_READY*: semnalează când magnitudinea este disponibilă pentru a fi folosită

Cu ajutorul Nadirei am eficientizat modulul FFT, iar acum doar 2 din cele 3 module ar mai fi necesare (iar dacă am considera că pierderea a 53 „sample”-uri este acceptabilă, unul ar fi suficient). În orice caz, controller-ul pentru semnalele: WRITE_FFT_0, WRITE_FFT_1, WRITE_FFT_2 se poate găsi aici: FFT input controller - code. În linii mari, este doar un numărător până la 1535 care dă toggle la cele 3 semnale depinzând care ar trebui să fie active
- 0:511 – FFT_0 și FFT_1
- 512:1023 – FFT_1 și FFT_2
- 1024: 1535 – FFT_2 și FFT_0

3. Calcularea Magnitudinilor

Pentru calcularea magnitudinilor așteptăm semnalul de completare a calculelor din FFT, care pornește următorul automat:



În codul pentru calcularea magnitudinii se observă că produsul *Real * Real* și *Imaginar * Imaginar* se începe doar când automatul se află în starea *MAGNITUDES_PROCESS_DATA*. În plus, datorită trucului matematic observat de Nadira, știm că partea reală și imaginară a numerelor se întorc în format Q12.4, și vom considera produsul tot în același format.

```
wire product_done, absolute_value_done;
wire signed [15: 0] P1, P2, absolute_value;

   fixed_point_multiplier #(.EXP_WIDTH_A(4), .EXP_WIDTH_B(4),
.EXP_WIDTH_PRODUCT(4)) MULTIPLY_REAL (
      .clk(clk),
      .enable(current_state == MAGNITUDES_PROCESS_DATA),
      .reset(reset),
      .A(dmadr_real),
      .B(dmadr_real),
      .done(product_done),
      .product(P1)
   );

   fixed_point_multiplier #(.EXP_WIDTH_A(4), .EXP_WIDTH_B(4),
.EXP_WIDTH_PRODUCT(4)) MULTIPLY_IMAG (
      .clk(clk),
      .reset(reset),
      .enable(current_state == MAGNITUDES_PROCESS_DATA),
      .A(dmadr_imag),
      .B(dmadr_imag),
      .product(P2)
   );

   fixed_point_truncation_adder ADD (
      .clk(clk),
      .enable(product_done),
      .A(P1),
      .B(P2),
      .sum(absolute_value),
      .done(absolute_value_done)
   );
```

Testul pentru corectitudinea implementării magnitudinilor FFT-ului se poate găsi aici: FFT with computed Magnitudes - functional test

4. Coordonator

Ca și coordonator am încercat să le ofer colegilor mei task-uri concrete și feedback pentru munca depusă, fapt pentru care m-am documentat extensiv și am testat multe variante de arhitecturi, folosind ceea ce ei mi-au oferit ca punct de plecare și extinzând împreună capacitățile sistemului. Am încercat cu toții să menținem moralul echipei ridicat, chiar și atunci când șansele de a reuși păreau foarte mici.

5. Python

Codul de python este o reinterpretare a sursei: How Shazam Works - An explanation in Python | Michael Strauss pentru a citi frecvențele de pe COM5, scriind în manieră asincronă în baza de date pentru a nu bloca citirea.