Politechnika Wrocławska
Wydział Elektroniki
Kierunek Informatyka

Architektura komputerów II Projekt

Całkowitoliczbowa transformata Fouriera

Autorzy:
Leszek Ostek (249439),
Jakub Labryszewski
(251001)

Termin zajęć:
Czwartek, 17:05-18:45 TN

Prowadzący:
Dr inż. Piotr Patronik

5 czerwca 2020

Spis treści

0.1 Wstęp 2
0.2 Podstawy teoretyczne 2
0.2.1 FFT 2
0.2.2 System RNS 4
0.3 Realizacja projektu 6
0.3.1 Implementacja całkowitoliczbowa 6
0.3.2 Implementacja w systemie RNS 8
0.4 Wyniki i dyskusja 10
0.4.1 Prezentacja działania FFT 10
0.4.2 Porównanie zwykłej FFT z wersją RNS 11
0.5 Wnioski 12

0.1 Wstęp

Celem naszego projektu było zapoznanie się z działaniem szybkiej całkowitoliczbowej transformaty Fouriera a następnie na podstawie uzyskanej wiedzy zaiplementować ją w komputerze za pomocą wybranego przez nas języka programowania. Omawianą FFT mieliśmy również zapisać w systemie RNS (system resztowy) i porównać jej działanie z podstawową wersją transformaty.

Aby zrealizować projekt na początku zapoznaliśmy się z dokumentem który szczegółowo opisuje ten problem. Kolejnym krokiem było wybranie języka programowania w którym zrealizujemy cel. Zdecydowaliśmy na język C gdyż uznaliśmy że nie będą nam potrzebne skąplikowane języki obiektowe gdyż samo FFT zajmie maksymalnie kilka metod. Również wchodziły w grę języki niższego poziomu na przykład Assembler lecz uznaliśmy że kod w tym zapisie może być mało czytelny. Przedostatnim krokiem była implementacja obu wersji FFT czyli całkowitoliczbowej oraz tej w RNS. Ważne było również oprócz prawidłowej kompliacji kodu to aby działał on poprawnie więc finalnym krokiem było dokładne sprawdzenie kodu oraz prawidłowe wyliczenie kilku przypadków widma i porównanie go z tym co zwracają nasze obie transformaty.

W poniższym wywodzie postaramy się szczegółowo opisać realizację projektu skupiając się głównie na podstawach teoretycznych jak i spróbujemy podać różnice obu implementacji jeżeli występują.

0.2 Podstawy teoretyczne

0.2.1 FFT

Na początku zaczniemy od wyjaśnienia czym jest transformata Fouriera. Nie wchodząc w szczegóły jest to funkcja przekształcona za pomocą wzoru:

$$F(f) = \int_{\infty}^{\infty} f(t) \cdot e^{-2j\pi ft} dt \tag{1}$$

Omawiana operacja przenosi sygnał w dziedzinę częstotliwości co pozwala nam to odczytać daną częstość sygnału, jest to bardzo przydatne na przykład przy obróbce dźwięku bądź przy kompresowaniu danych chociażby w postaci zdjęcia.
My natomiast będziemy omawiać dystkretną wersję tej transformaty operującej na liczbach całkowitych w dodatku z zastosowaniem pewnych sztuczek optymalizacyjnych dzięki którym złożoność obliczeniowa z $O(N^2)$ spada do $O(Nlog_2N)$ co w skrajnych przypadkach pozwala przyspieszyć obliczenia nawet o kilka lat. Będzie to FFT całkowitoliczbowa gdyż pozwoli nam to na obliczanie jej na urządzeniach które nie mają koprocesora. Niezoptymalizowana wersja dana jest wzorem:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \quad dla\ k = 0, 1, ..., N-1 \tag{2}$$

gdzie $W_N = e^{-j\pi/N}$

Stostjąc pewne sztuczki da się zauważyć pewną powtarzalność, wadą nowego rozwiązania będzie to że sygnał musi zawierać $N = 2^K$ próbek. Funkcja W_N ma pewne właściwości:

$$W_N^{k+N/2} = -W_N^k, \quad W_N^{k+N} = W_k^N \tag{3}$$

Stosując tą zależność możemy rozłożyć powyższy wzór na trzy mniejsze które będą używały powtarzających się czynników dzięki czemu oszczędzimy na czasie licząc je tylko raz. Wzory prezentują się następująco:

$$X(2k) = \sum_{n=0}^{N/2-1} [x(n) + x(n+N/2)]W_{N/2}^{kn} \tag{4}$$

$$\text{Dla } k = 0, 1, ..., N/2 - 1$$

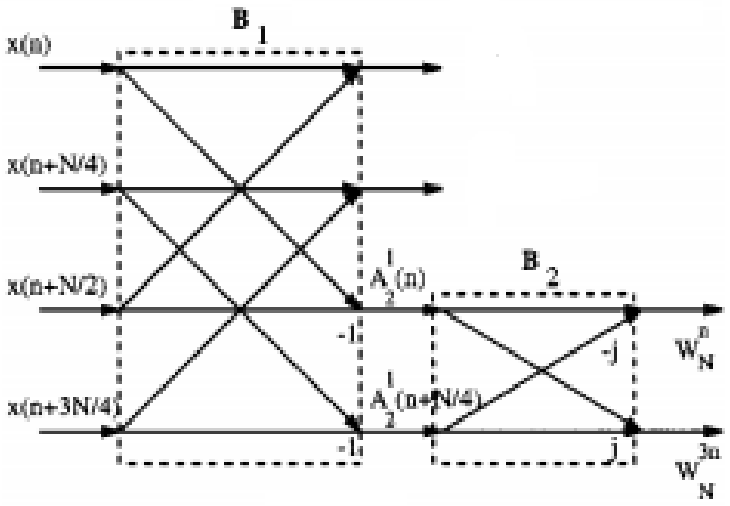
oraz

$$X(4k+1) = \sum_{n=0}^{N/4-1} [x(n) - jx(n+N/4) - x(n+N/2) + jx(n+3N/4)]W_N^n W_{N/4}^{kn} \tag{5}$$

$$X(4k+3) = \sum_{n=0}^{N/4-1} [x(n) + jx(n+N/4) - x(n+N/2) - jx(n+3N/4)]W_N^{3n} W_{N/4}^{kn} \tag{6}$$

$$\text{Dla } k = 0, 1, ..., N/4 - 1$$

Jak można teraz zauważyć wiele czynników dla odpowiednich n sie powtarza. Możemy ten fakt wykorzystać, pomaga nam w tym tworzenie tak zwanych struktur zwanych motylkami oraz rozbijanie transformaty na mniejsze elemanty. Nazwa motylek wzięła się z tąd że na schemacie struktury transformaty przypomina on motyla.



Rysunek 1: Czteropunktowa mała transformata

Powyższa struktura może przybierać większe rozmiary będące potęgą dwójki. Całą operacje którą wykonuje ten element można jeszcze przyspieszyć poprzez zoptymalizowanie mnożenia. Jak wiemy dana część sygnału $x(n)$ to liczba zespolona, jak i W_N^n to też liczba zespolona, mnożąc te dwie liczby otrzymujemy coś takiego $x(n) \cdot W_N^n = (x_r + jx_i)(c + js) = x_r c + x_r js + jx_i c - x_i s$. Daje to nam cztery sumy

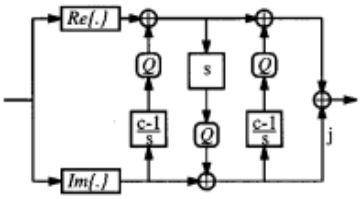
i cztery iloczyny, co ciekawe da się tą liczbę zmniejszyć. Na początku zapisujemy powyższe równanie w formie macierzowej:

$$y = [1 \quad j] \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} x_r \\ x_i \end{bmatrix} = [1 \quad j]R \begin{bmatrix} x_r \\ x_i \end{bmatrix} \tag{7}$$

Ową macierz R możemy rozpisać na trzy kolejne macierze.

$$R = \begin{bmatrix} 1 & \frac{c-1}{s} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix} \begin{bmatrix} 1 & \frac{c-1}{s} \\ 0 & 1 \end{bmatrix} \tag{8}$$

Gdy stablicujemy powyższe $\frac{c-1}{s} = \frac{cos(n)-1}{sin(n)}$ oraz $s = sin(n)$ zaoszczędzimy na jednym mnożeniu i jednym dodawaniu co przy wielu operacjach jest sporym przyspieszeniem. Poniższy schemat pokazuje jak będzie wyglądało mnożenie przez W_N^n.



Rysunek 2: Schemat mnożący

Ponieważ powyższe elementy przez które mnożymy nie są liczbami całkowitymi więc aby móc je poprawnie stablicować stosując nadal założenie że transformata ma być całkowitoliczbowa musimy pomnożyć je przez jakąś stałą. Postanowiliśmy zastosować mnożnik 2^{15} gdyż taki został zaproponowany w literaturze którą udostępnił nam prowadzący. Element Q to funkcja kwantująca przesuwająca wynik bitowo o 15 w prawo co jest równoważne z podzieleniem przez 2^{15}. Jest to operacja niezbędna aby zachować poprawność obliczeń. Element Q może również zastosować różnego rodzaju zaokrąglenia aby poprawić ostateczny wynik transformaty.

Ostatnią rzeczą na którą należy zwrócić uwagę to to że wyjścia transformaty są pozamieniane miejscami. Ta zamiana odbywa się w taki że w lini wejścia $x(k)$ znajduję się wyjście $X(K)$ gdzie K to liczba k z zastosowaniem pewnego odwrócenia bitowego. W matematycznych słowach jest to opisane w taki sposób że dla transformaty o długości N binarna wartość numeru indeksu wejścia zapisanego binarnie wynosi $L = Log_2 N$, więc liczba K to taka liczba że jeśli weźmiemy liczbę k i na bit na pozycji i wstawimy bit z pozycji $L - 1 - i$ to otrzymamy liczbę K.

0.2.2 System RNS

System resztowy jest nie wagowym systemem liczbowym. Dlatego znacznie się różni od systemów wagowych takich jak system binarny czy dziesiętny. Operacje arytmetyczne wykonywane w tym systemie takie jak dodawanie, odejmowanie czy mnożenie działają bez przeniesienia - każda liczba wynikiu jest funkcją tylko jednej liczby z każdego operandu, tym samym jest niezależna od innych liczb wyniku. Takie podejście do tematu jest stosowane w przetwarzaniu sygnałów - gdzie w większości wykonuje się tylkko operacje dodawania, odejmowania i mnożenia - co znacznie wpływa na czas przetwarzania. W tym systemie liczbę przedstawia się za pomocą reszt z dzielenia przez liczby całkowite zwane modułami, których zbiór nazywamy

bazą. Dla danej bazy $(m_1, m_2, ..., m_n)$ resztowa reprezentacja liczby całkowitej X jest zbiorem n liczb $(x_1, x_2, ..., x_n)$, gdzie x_i to liczby całkowite zdefiniowane przez zbiór n równań:

$$X = q_i * m_i + x_i... \tag{9}$$

$$\text{Dla } i = 1, 2, ..., n$$

i q_i to liczba całkowita wybrana w ten sposób aby $0 \leq x_i \leq m_i$. Resztowa reprezentacja liczby jest unikalna jeśli wartość liczby mieści się pomiędzy $-(M-1) \div 2$ do $(M-1) \div 2$ dla nieparzystych i od $-M \div 2$ do $(M \div 2 - 1)$ dla przarzystych, gdzie $M = m_1 * m_2 * ... * m_n$.

Dodawanie i odejmowanie w systemie RNS

Dla systemu resztowego złożonego z modułów: $m_1, m_2, ..., m_n$, niech X i Y będą liczbami w tym systemie resztowym. Dodawanie i odejmowanie X i Y:

$$X = (|X|_{m1}, |X|_{m2}, ..., |X|_{mn})$$

$$Y = (|Y|_{m1}, |Y|_{m2}, ..., |Y|_{mn})$$

Jest dane wzorem:

$$|X \pm Y| = (||X|_{m1} \pm |Y|_{m1}|_{m1}, ||X|_{m2} \pm |Y|_{m2}|_{m2}, ..., ||X|_{mn} \pm |Y|_{mn}|_{mn}) \tag{10}$$

Mnożenie w systemie RNS

Podobnie jak wyżej X i Y mają postać:

$$X = (|X|_{m1}, |X|_{m2}, ..., |X|_{mn})$$

$$Y = (|Y|_{m1}, |Y|_{m2}, ..., |Y|_{mn})$$

Dla systemu resztowego złożonego z modułów: $m_1, m_2, ..., m_n$, mnożenie będzie wyrażone wzorem:

$$|X * Y| = (||X|_{m1} * |Y|_{m1}|_{m1}, ||X|_{m2} * |Y|_{m2}|_{m2}, ..., ||X|_{mn} * |Y|_{mn}|_{mn}) \tag{11}$$

Zamiana z systemu RNS na system dziesiętny

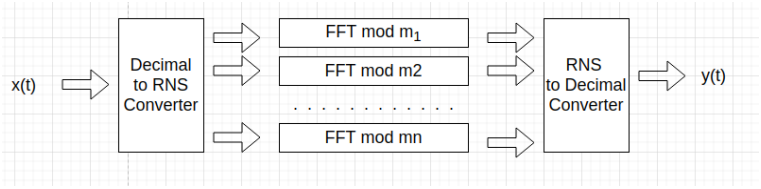
W naszej impelementacji Transformaty Fouriera z użyciem systemu RNS - gdzie liczba w tym systemie ma postać (x_1, x_2, x_3) - stosujemy zbiór modułów m_1, m_2, m_3 postaci $(2n+1, 2n, 2n-1)$. Dla takich zależności w zbiorze modułów liczba dziesiętna ma postać:
Dla $(x_1 + x_3)$ parzystego:

$$|X| = |\frac{m_2 * m_3 * x_1}{2} - m_1 * m_3 * x_2 + \frac{m_1 * m_2 * x_3}{2}|_M \tag{12}$$

Dla $(x_1 + x_3)$ nieparzystego:

$$|X| = |\frac{M}{2} + \frac{m_2 * m_3 * x_1}{2} - m_1 * m_3 * x_2 + \frac{m_1 * m_2 * x_3}{2}|_M \tag{13}$$

Schemat działania transformaty Fouriera w systemie resztowym



Rysunek 3: Schemat działania transformaty Fouriera w systemie RNS

0.3 Realizacja projektu

0.3.1 Implementacja całkowitoliczbowa

W tym kroku omówimy szczegółowo jak zrealizowaliśmy projekt. Zdecydowaliśmy się na 32 punktową FFT. Wielkość naszej implementacji ściśle zależy od stałych jakie zdefiniujemy, dzięki zastosowaniu rekurencji możemy powiększać naszą transformate do puki nie zacznie pojawiać się problem przepełnionego stosu. Aby policzyć mnożniki dla naszej 32 punktowej FFT użyliśmy narzędzia Excel, wyniki wyliczeń znajdują się w tabeli poniżej:

	angle		c-1/s	s
0	0		x	x
1	-0,196349541		3227	-6393
2	-0,392699082		6518	-12540
3	-0,589048623		9940	-18205
4	-0,785398163		13573	-23170
5	-0,981747704		17515	-27246
6	-1,178097245		21895	-30274
7	-1,374446786		26892	-32138
8	-1,570796327		32768	-32768
9	-1,767145868		39928	-32138
10	-1,963495408		49041	-30274
11	-2,159844949		61305	-27246
12	-2,35619449		79109	-23170
13	-2,552544031		108022	-18205
14	-2,748893572		164736	-12540
15	-2,945243113		332699	-6393
16	-3,141592654		x	x
17	-3,337942194		-332699	6393
18	-3,534291735		-164736	12540
19	-3,730641276		-108022	18205
20	-3,926990817		-79109	23170
21	-4,123340358		-61305	27246
22	-4,319689899		-49041	30274
23	-4,51603944		-39928	32138
24	-4,71238898		-32768	32768
25	-4,908738521		-26892	32138
26	-5,105088062		-21895	30274
27	-5,301437603		-17515	27246
28	-5,497787144		-13573	23170
29	-5,694136685		-9940	18205
30	-5,890486225		-6518	12540
31	-6,086835766		-3227	6393

Rysunek 4: Stałe pomnożone przez 2^{15}

Co ciekawe tabele z literatury udostępnionej przez prowadzącego [1], prawdopodobnie mają źle stablicowane stałe dla 16 punktowej FFT. Gdy wykorzystaliśmy stałe podane w tym dokumęcie otrzymywaliśmy błędne wyniki, również po wyliczeniu ich zauwazyliśmy że nasze stałe się różnią.

Teraz przejdziemy do omówienia kodu. Transformate Fouriera oblicza u nas powtarzana rekurencyjnie funkcja. Zapętla ona do skutku strukturę z *Rysunek 1* [1].

```
for(int i=start;i<start+len2;i++){
    signal->re[i]=re[i]+re[i+len2];
    signal->im[i]=im[i]+im[i+len2];
    signal->re[i+len2]=re[i]-re[i+len2];
    signal->im[i+len2]=im[i]-im[i+len2];
}
```

Rysunek 5: Czesc B_1

Ta część kodu wykonuje dokładnie sumowanie i odejmowanie oznaczone B_1 na *Rysunek 1* [1].

```
    signal->re[start+len2]=re[start+len2]+im[start+len2+len4];
    signal->im[start+len2]=im[start+len2]-re[start+len2+len4];
    signal->re[start+len2+len4]=re[start+len2]-im[start+len2+len4];
    signal->im[start+len2+len4]=im[start+len2]+re[start+len2+len4];
for(int i=start+len2+1;i<start+len2+len4;i++){
    signal->re[i]=re[i]+im[i+len4];
    signal->im[i]=im[i]-re[i+len4];
    signal->re[i+len4]=re[i]-im[i+len4];
    signal->im[i+len4]=im[i]+re[i+len4];

    signal->re[i]+=quantize(signal->im[i]*lc1[(i-start-len2)*stride]);
    signal->im[i]+=quantize(signal->re[i]*lc2[(i-start-len2)*stride]);
    signal->re[i]+=quantize(signal->im[i]*lc1[(i-start-len2)*stride]);

    signal->re[i+len4]+=quantize(signal->im[i+len4]*lc1[(i-start-len2)*3*stride]);
    signal->im[i+len4]+=quantize(signal->re[i+len4]*lc2[(i-start-len2)*3*stride]);
    signal->re[i+len4]+=quantize(signal->im[i+len4]*lc1[(i-start-len2)*3*stride]);

}
```

Rysunek 6: Czesc B_2 oraz mnozenie przez stałe

Ta część wykonuje natomiast operację B_2 oraz mnoży przez stałe odpowiednie stałe z tabeli *Rysunek 3* [4] kwantując przy tym wynik aby zachowac poprawność obliczeń. Jak widać pierwsza iteracja pętli jest pominięta i wyciągnięta przed pętle *for* gdyż uwzględnia ona przypadki z pominięciem mnożenia przez W_N^n które ma wartość 1. Zmienna *stride* zapewnia nam takie przesunięcie że obliczana aktualnie N punktowa struktura korzysta z właściwych stałych.

```
int quantize(int number){
    number>>=14;
    number++;
    number>>=1;
    return number;
}
```

Rysunek 7: Funkcja kwantująca

Ta funkca kwantuje wynik aby zlikwidowac wpływ mnożenia stałych przez 2^{15}. Dzielenie realizujemy poprzez przesunięcia bitowe w prawo. Zastosowaliśmy w niej binarne zaokrąglanie w górę. Wybraliśmy to rozwiązanie gdyż przy wielu próbach dawało to najbardziej satysfakcjonujące wyniki. Nie jesteśmy pewni czy jest to najlepsze rozwiązanie.

```
void fft(Signal* signal,int start,int length)

fft(signal,start,len2);
fft(signal,start+len2,len4);
fft(signal,start+len2+len4,len4);
```

Rysunek 8: Wejścia FFT oraz jej rekurencyjne zapętlenie

Teraz omówimy wejścia funkcji oraz jak wygląda jej zapętlanie. Funkcja przetwarza strukturę *Signal* która ma w sobie części rzeczywiste i urojone sygnału jak i jego całkowitą długość. Zmienna *start* mówi nam od którego punktu liczymy transformate, *length* jak długa jest ta część sygnału. Poprzednie wejścia są potrzebne aby poprawnie wykonać rekurencję. Zmienne *len2* , *len4* to odpowiednio *length/2* , *length/4*. Transformata jest przewidziana na maksymalną długość 32, lecz można bez problemu policzyć za jej pomocą 16 punktową podając za argumęt *length* wartość 16 a za *start* wartość 0. Oczywiście 8,4,2,1 punktowe też wchodzą w grę.

Należy jeszcze zwrócić uwagę na to że sygnał po przejściu przez transformatę ma pozamieniane wyjścia. Aby poprawnie odczytać wynik transformaty napisaliśmy funkcję która stosuje odwrócenie bitowe które było wcześniej omówione.

```
int bitSwap(int number,int binaryLength){
    int temp=0;
    for(int i=0;i<binaryLength/2;i++){
        temp|=(number&(1<<(binaryLength-i-1)))>>(binaryLength-1-2*i)|(number&(1<<(i)))<<(binaryLength-1-2*i);
    }
    if(binaryLength/2*2!=binaryLength)
    temp|=number&(1<<binaryLength/2);
    return temp;
}
```

Rysunek 9: Funkcja zamieniająca bity

0.3.2 Implementacja w systemie RNS

Implementacja transformaty w systemie RNS różni się od implementacji za pomocą liczb całkowitych jednynie sposobem reprezentacji liczb oraz wykonywania działań - reszta pozostaje bez zmian. Poniżej przedstawiamy funkcje implemętjące operacje arytmetyczne w RNS oraz te odpowiedzialne za zamiane liczby na i z systemu RNS:

```
int* INTToRNS(int INTnumber)
{

        int* RNSnumber = (int*) malloc((MODULO_AMOUNT + 1)*sizeof(int));
        for (int i = 0; i < MODULO_AMOUNT; i++) {
                RNSnumber[i] = INTnumber % modulo[i];
                if (RNSnumber[i] < 0)RNSnumber[i] += modulo[i];
                RNSnumber[MODULO_AMOUNT] = INTnumber;

        }
        return RNSnumber;
}
```

Rysunek 10: Funkcja zamiany liczby w systemie dziesiętnym na liczbe w systemie RNS

```
int RNSToINT(int* RNSnumber)
{
        int INTnumber = 0;
        int M = countM();
        if ((RNSnumber[0] + RNSnumber[2]) % 2)
        {//x1 + x3 nieparzyste

                INTnumber = (M / 2 + (modulo[1] * modulo[2] * RNSnumber[0]) / 2 - modulo[0] * modulo[2] * RNSnumber[1] + (modulo[0] * modulo[1] * RNSnumber[2]) / 2) % M;
                if (INTnumber < 0)INTnumber += M;
                if (RNSnumber[MODULO_AMOUNT] < 0 )INTnumber -= M;
        }
        else
        {//x1 + x3 parzyste

                INTnumber = ((modulo[1] * modulo[2] * RNSnumber[0]) / 2 - modulo[0] * modulo[2] * RNSnumber[1] + (modulo[0] * modulo[1] * RNSnumber[2]) / 2) % M;
                if (INTnumber < 0)INTnumber += M;
                if (RNSnumber[MODULO_AMOUNT] < 0)INTnumber -= M;
        }

        return INTnumber;
}
```

Rysunek 11: Funkcja zamiany liczby w systemie resztowym na liczbe w systemie dziesiętnym

```
int* add(int* rns1, int*rns2)
{

        int* sum = (int*) malloc((MODULO_AMOUNT + 1)*sizeof(int));
        sum[MODULO_AMOUNT] = rns1[MODULO_AMOUNT]+rns2[MODULO_AMOUNT];
        for(int i=0; i<MODULO_AMOUNT; i++)
        {
                int buf = (rns1[i] + rns2[i]) % modulo[i];
                if (buf < 0)buf += modulo[i];
                sum[i] = buf;
        }

        return sum;
}
```

Rysunek 12: Dodawanie w systemie RNS

```
int* sub(int* rns1, int*rns2)
{

        int* sub = (int*) malloc((MODULO_AMOUNT + 1)*sizeof(int));
        sub[MODULO_AMOUNT] = rns1[MODULO_AMOUNT]-rns2[MODULO_AMOUNT];
        for(int i=0; i<MODULO_AMOUNT; i++)
        {
                int buf = (rns1[i] - rns2[i]) % modulo[i];
                if (buf < 0)buf += modulo[i];
                sub[i] = buf;
        }

        return sub;
}
```

Rysunek 13: Odejmowanie w systemie RNS

```
int* mul(int* rns1, int*rns2)
{

        int* mul = (int*) malloc((MODULO_AMOUNT + 1)*sizeof(int));
        mul[MODULO_AMOUNT] = rns1[MODULO_AMOUNT]*rns2[MODULO_AMOUNT];
        for(int i=0; i<MODULO_AMOUNT; i++)
        {
                int buf = (rns1[i] * rns2[i]) % modulo[i];
                if (buf < 0)buf += modulo[i];
                mul[i] = buf;
        }

        return mul;
}
```

Rysunek 14: Mnożenie w systemie RNS

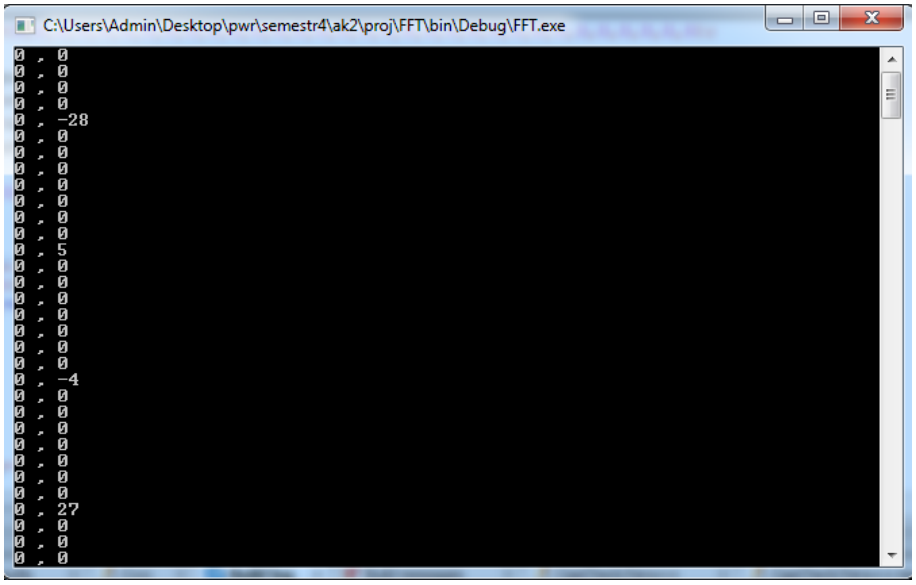
0.4 Wyniki i dyskusja

0.4.1 Prezentacja działania FFT

Tą część zaczniemy od zaprezentowania działania naszej transformaty na przykładzie sygnału przypominającego przebieg sinusoidalny.

```
int re[32]={0,1,2,1,0,-1,-2,-1,0,1,2,1,0,-1,-2,-1,0,1,2,1,0,-1,-2,-1,0,1,2,1,0,-1,-2,-1};
int im[32]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
```

Rysunek 15: Sygnał podany na wejście FFT



Rysunek 16: Wynik FFT, po lewej część rzeczywista a po prawej urojona

Jak widać charakterystyczne piki częstotliwości są na właściwych miejscach. Aby sprawdzić czy wynik jest poprawny policzyliśmy widmo tego samego sygnału w narzędziu *Matlab*. Jak się okazało wynik jest prawie identyczny tylko że *Matlab* wyliczył ją z użyciem liczb zmiennoprzecinkowych.

```
0.00000 +  0.00000i    0.00000 +  0.00000i    0.00000 +  0.00000i
0.00000 +  0.00000i    0.00000 - 27.31371i    0.00000 -  0.00000i
0.00000 -  0.00000i    0.00000 -  0.00000i    0.00000 +  0.00000i
0.00000 +  0.00000i    0.00000 +  0.00000i    0.00000 +  0.00000i
0.00000 +  4.68629i    0.00000 -  0.00000i    0.00000 -  0.00000i
0.00000 -  0.00000i    0.00000 +  0.00000i    0.00000 +  0.00000i
0.00000 +  0.00000i    0.00000 +  0.00000i    0.00000 -  4.68629i
0.00000 -  0.00000i    0.00000 -  0.00000i    0.00000 -  0.00000i
0.00000 -  0.00000i    0.00000 +  0.00000i    0.00000 +  0.00000i
0.00000 +  0.00000i    0.00000 + 27.31371i    0.00000 -  0.00000i
0.00000 -  0.00000i    0.00000 -  0.00000i
```

Rysunek 17: FFT policzone w Matlabie

0.4.2 Porównanie zwykłej FFT z wersją RNS

Najbradziej trywialne porównanie jaki przyszło nam do głowy to porównanie czasowe obu implementacji. Na podstawie pomiaru 1000 prób i uśrednienia ich otrzymaliśmy następujące wyniki:

```
Czas wykonanie FFTRNS to 0.000121 sekund Czas wykonania zwyklego FFT to 0.000057 sekund
Czas wykonanie FFTRNS to 0.000142 sekund Czas wykonania zwyklego FFT to 0.000055 sekund
Czas wykonanie FFTRNS to 0.000136 sekund Czas wykonania zwyklego FFT to 0.000046 sekund
```

Rysunek 18: Porównanie czasowe

Pomiar wykonywaliśmy funkcją napisaną w Assemblerze która zlicza ilość cykli procesora. Po podzieleniu liczby cykli przez taktowanie procesora otrzymywaliśmy czas wykonania. Od razu widać która wersja ma przewagę, same pomiary nie

uwzględniają czasu przejścia między systemem RNS a całkowitym więc tak naprawde zwykła FFT ma jeszcze większą przewage. Niestety tak nie powinno być gdyż wersja RNS bez uwzględnienia konwersji powinna być szybsza. Prawie na pewno to wina naszej implementacji.

0.5 Wnioski

Koniec końców udało nam się zrealizować celę które sobie postawiliśmy. Obie implementacje udało nam się zrealizować przy czym każda z nich generuje identyczne wyniki. Największym problemem przy tworzeniu projektu nie było same napisanie kodu lecz rozszyfrowanie działania algorytmów które mieliśmy zaimplementować. Jak zauważyliśmy powodem przewagi FFT nad DFT jest wykorzystanie tego że wiele tych samych operacji w DFT jest niepotrzebnie liczone po kilka razy. Szybka transformata Fouriera wykorzystuję ten fakt i dzięki rozbiciu DFT na mniejsze elemęty i użycie struktur motylka udaje się uzyskać złożoność niemal liniową. Do tego da się jeszcze zastosować sztuczki zmniejszające liczbe iloczynów i sum co pozwala zaoszczędzić jeszczę więcej czasu. Niestety zastosowanie wersji RNS zmniejszyło szybkość transformaty. Możliwe że dało sie to zrobić lepiej aby transformata używająca RNS też była skuteczna ale z powodu braku czasu nie przestudiowaliśmy tematu na tyle głęboko.

Bibliografia

[1] S. Oraintara, Y.-J. Chen, and T. Q. Nguyen. Integer fast fourier transform. IEEE Transactions on Signal Processing, 50(3):607–618, 2002.

[2] Assaid Othman Sharoun. RESIDUE NUMBER SYSTEM (RNS). Poznan University of Technology Academic Journals, 2013.

[3] Kazeem Alagbe Gbolagade. Effective Reverse Conversion in Residue Number System Processors.Department of Computer Science University of Ibadan Nigeria, 2010.