INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE INGENIERÍA EN ELECTRÓNICA
ARQUITECTURA DE COMPUTADORAS I

Proyecto 1. Transformada rápida de Fourier con Yocto Project

Osvaldo Jesús Alfaro González
Alejandro Flores Herrera
Allan Gutiérrez Quesada
Eduardo Salazar Villalobos

Profesor: Johan Carvajal Godínez

II Semestre 2019

Tabla de contenidos

Descripción del proyecto

Este proyecto consiste en implementar una FFT en un sistema embebido. Se puede seccionar en dos partes, la elaboración del algoritmo para realizar la transformada discreta de Fourier, en lenguaje ensamblador NASM para una arquitectura x86 de Intel; y el acondicionamiento de la herramienta Yocto para poder recibir esa programación, crear una imagen compatible con el simulador de embebido a utilizar y poder entregar el resultado correspondiente.

En la sección de ensamblador, se procede a utilizar los registros comunes del procesador, junto con los de punto flotante para realizar las operaciones. El cálculo de los coeficientes se da internamente y el usuario solamente ingresa previamente el valor que va a tener el vector de entrada, y la cantidad de puntos. Se intentó trabajar en su mayoría con registros de 32 bits, pero en la sección de impresión de valores, se incluyen operaciones con el stack, lo cual involucra registros de 64 bits, entonces la compilación final se tiene que dar en un entorno de 64 bits.

Para el entorno de Yocto, se utiliza la versión poky-warrior. Luego de la instalación, se procede a incluir NASM para poder compilar el archivo NASM_FFT.asm. Además, se debe decidir el simulador del sistema embebido para utilizar como “Target”. En versiones de Linux, es usual que el predeterminado sea qemux86, pero para el caso trabajado no es posible utilizar este debido a que funciona correctamente para archivos NASM en arquitecturas de 32 bits. Entonces, se necesita entrar a los archivos de poky y cambiar a qemux86-64.

Parte 1. Implementación de algoritmo FFT en lenguaje ensamblador x86

Primeramente se buscó y se analizó la ecuación matemática que calcula la Transformada Rápida de Fourier, como se muestra a continuación:

$$X_k = \sum_{n=0}^{N-1} x_n \left(\cos\left(\frac{2\pi kn}{N} \right) - jsen\left(\frac{2\pi kn}{N} \right) \right) \qquad k = 0, 1, 2 \ldots \tag{1}$$

donde x_n representa el vector de entrada de la función discreta en el tiempo y N la cantidad de puntos que se quieren muestrear del espectro de la señal.

Para la implementación del algoritmo FFT, se segmentó el código en distintos macros con funcionalidades específicas, que facilitan la construcción, organización y entendimiento del código. A continuación, se muestra cada uno de los macros creados, junto con una breve explicación y diagramas de flujos explicativos en pseudocódigo para los macros que lo requieren.

1.1 Macros

exit

- Función: Realiza una llamada al sistema para salir del programa.
- Entradas: Ninguna.
- Salidas: Ninguna.
- Código:

```
%macro exit 0
        mov eax, SYS EXIT
        mov edi, 0
        syscall
%endmacro
```

multiply

- Función: Multiplica dos números punto flotante en precisión simple (SPFP).
- Entradas: Dos factores SPFP.
- Salidas: Producto SPFP en el registro xmm0.
- Código:

```
%macro multiply 2
    movss    xmm0, %1
    mulss    xmm0, %2
%endmacro
```

divide

- Función: Realiza una división de dos números SPFP.
- Entradas: Dividendo y divisor SPFP.
- Salidas: Cociente SPFP en el registro xmm0.
- Código:

```
%macro divide 2
    movss   xmm0, %1
    divss   xmm0, %2
 %endmacro
```

suma

- Función: Realiza una suma de dos números SPFP.
- Entradas: Sumandos SPFP.
- Salidas: Total de la suma SPFP en el registro xmm0.
- Código:

```
%macro suma 2
    movss   xmm0, %1
    addss   xmm0, %2
 %endmacro
```

resta

- Función: Realiza una resta de dos números SPFP.
- Entradas: Minuendo y sustraendo SPFP.
- Salidas: Diferencia SPFP en el registro xmm0.
- Código:

```
%macro resta 2
    movss   xmm0, %1
    subss   xmm0, %2
 %endmacro
```

printF

- Función: Imprimir un número punto flotante. Requiere uso de la librería *printf*.
- Entradas: Número SPFP, formato de impresión.
- Salidas: Impresión en terminal del número.
- Código:

```
%macro printF 2
    push    rbp
    mov     rbp, rsp
    movss   xmm0, %1
    movss   dword [rbp-4], xmm0
    cvtss2sd xmm0, dword [rbp-4]
    lea     edi, %2
    mov     eax, 1
```

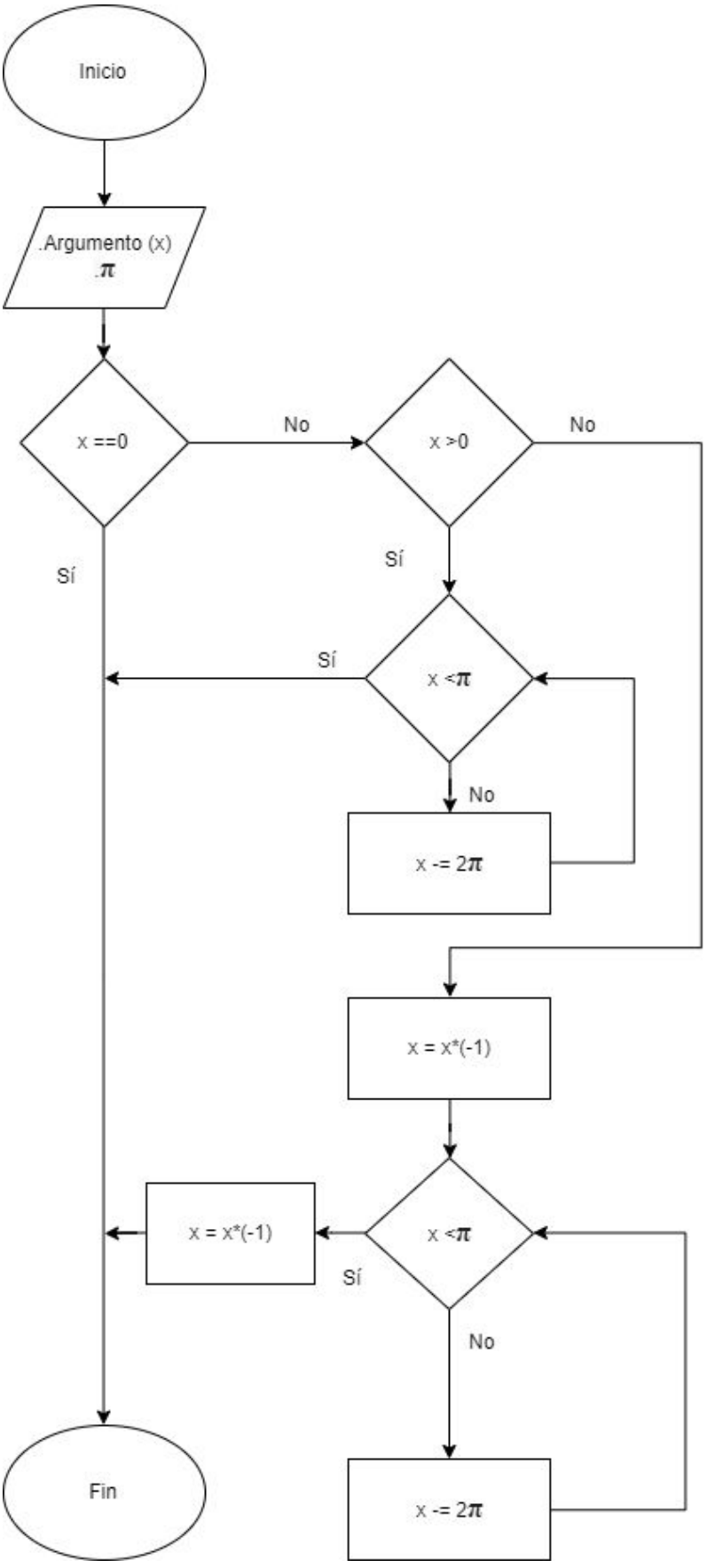
```
    call    printf
    mov     rsp, rbp
    pop     rbp
%endmacro
```

rango

- Función: Convierte un número flotante a su equivalente en el rango [-π,π]. Utiliza de memoria el valor de -1.
- Entradas: Número SPFP a convertir, valor de número π SPFP.
- Salidas: Número equivalente SPFP, en el registro xmm0.
- Código:

```
%macro rango 2
    movss xmm0, %1
    pxor xmm1, xmm1      ;xor para que xmm1 sea 0
    ucomiss xmm0, xmm1   ;comparacion con 0
    je %%fin             ;si es cero, salta al fin
    jbe %%negativo       ;si es menor, a negativo
    jmp %%positivo       ;si es mayor, a positivo
%%positivo:
    ucomiss xmm0, %2     ;compara con pi:
    jbe %%fin            ;si es menor, salta a  fin
    resta xmm0, %2       ;se le resta 2*pi
    resta xmm0, %2
    jmp %%positivo
%%negativo:
    ;se le cambia el signo para que quede positivo
    mulss xmm0, [negativo]
    ;se hace lo mismo que cuando es positivo:
%%negativo2:
    ucomiss xmm0, %2     ;compara con pi:
    jbe %%cambioSigno    ;si es menor, salta a cambio de signo
    resta xmm0, %2       ;se le resta 2*pi
    resta xmm0, %2
    jmp %%negativo2
%%cambioSigno: ;cambia el signo del resultado, porque -sin(x)=sin(-x)
    mulss xmm0, [negativo] ;se multiplica por -1
%%fin:
%endmacro
```

- Diagrama de flujo:


Inicio
.Argumento (x)
.π
x ==0
No
x >0
No
Sí
Sí
Sí
x <π
No
x -= 2π
x = x*(-1)
x = x*(-1)
x <π
Sí
No
Fin
x -= 2π

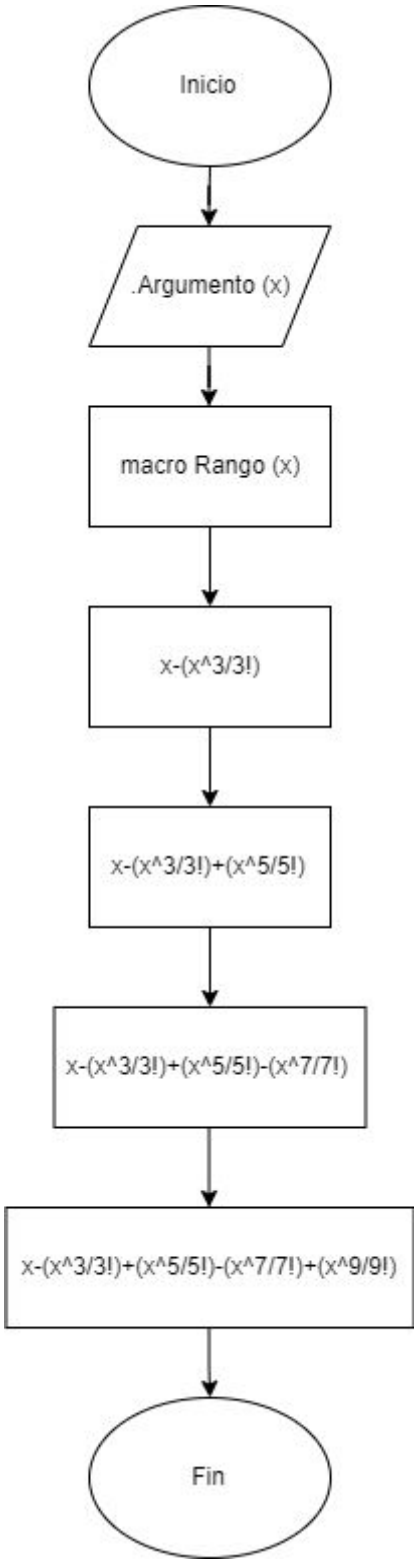
sen

- Función: Aproxima el seno de un número SPFP en radianes, por medio de un polinomio de Taylor de orden 9. Utiliza desde memoria el valor de π/2 y factores previamente calculados del polinomio de Taylor. Además se utiliza el macro *rango.*
- Entradas: Argumento SPFP.
- Salidas: Seno del argumento, almacenado en el registro xmm0.
- Código:

```
%macro sen 1
    rango %1, [pi] ;se pone el numero en el rango de -pi a pi
    movss %1, xmm0


    ;la suma se va almacenando en xmm1:
    ;primer término de la serie:    x
    movss xmm1, %1


    ;segundo término de la serie:   -x^3/3!
    movss xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, [factor3]
    multiply xmm0, [negativo] ;se multiplica por -1
    suma xmm0,xmm1      ;se suma con el termino anterior, resultado en xmm0
    movss xmm1, xmm0    ;se almacena el resultado en xmm1


    ;tercer término de la serie:    x^5/5!
    movss xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, [factor5]
    suma xmm0,xmm1  ;se suma con los términos anteriores, resultado en xmm0
    movss xmm1, xmm0   ;se almacena el resultado en xmm1


    ;cuarto término de la serie:    -x^7/7!
    movss xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, [factor7]
```

```
    multiply xmm0, [negativo] ;se multiplica por -1
    suma xmm0,xmm1  ;se suma con los términos anteriores, resultado en xmm0
    movss xmm1, xmm0   ;se almacena el resultado en xmm1


    ;quinto termino de la serie:    x^9/9!
    movss xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, %1
    multiply xmm0, [factor9]
    suma xmm0,xmm1  ;se suma con los términos anteriores, resultado en xmm0
%endmacro
```

- Diagrama de flujo:

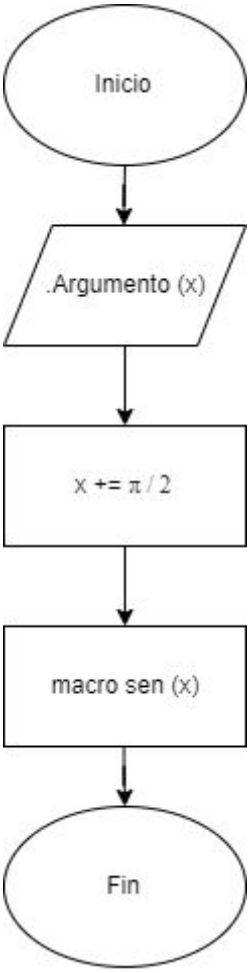


cos

- Función: Aproxima el coseno de un número SPFP en radianes a partir del macro *sen*, calculando el seno del argumento + π/2. Utiliza desde memoria el valor de π/2.
- Entradas: Argumento SPFP.
- Salidas: Coseno del argumento, almacenado en el registro xmm0.
- Código:

```
%macro cos 1
    movss xmm0, %1
     suma xmm0, [piMedios] ;se le suma pi medios para calcular coseno a partir
de seno
    movss %1, xmm0
    sen %1
%endmacro
```

- Diagrama de flujo:



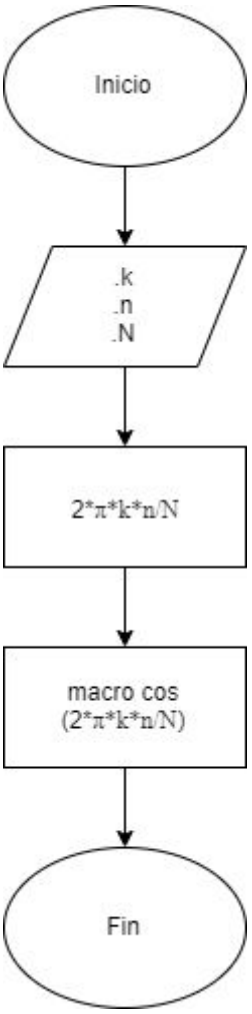
coefRe

- Función: Calcula la parte real del exponencial complejo de la FFT. Utiliza desde memoria el valor de π y el macro *cos*.
- Entradas: Valores de contadores para k, n y valor de N (número de puntos de la FFT).
- Salidas: Coeficiente real SPFP en el registro xmm0.
- Código:

```
%macro coefRe 3
    mov eax, 2
```

```
    cvtsi2ss xmm0, eax      ;se guarda un 2 en xmm0
    multiply xmm0, [pi]     ;se multiplica por pi

    mov eax, %1 ;k
    cvtsi2ss xmm1, eax
    multiply xmm0, xmm1     ; se multiplica por k

    mov eax, %2 ;n
    cvtsi2ss xmm1, eax
    multiply xmm0, xmm1     ; se multiplica por n

    mov eax, %3 ;N
    cvtsi2ss xmm1, eax
    divide xmm0, xmm1       ; se divide entre N

    movss xmm10, xmm0       ; es mejor no usar xmm0 para calcular el coseno,
    cos xmm10               ; porque xmm0 se modifica dentro del macro
%endmacro
```

- Diagrama de flujo:



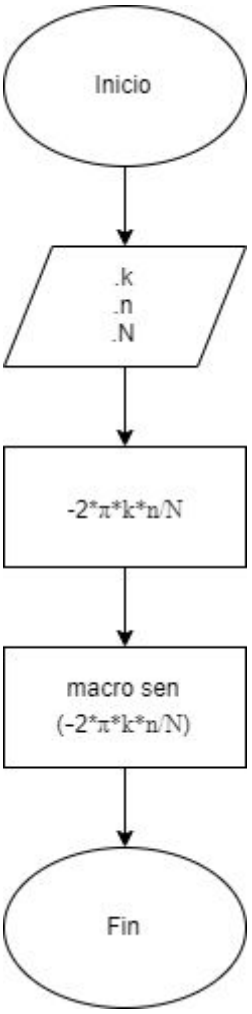
coefIm

- Función: Calcula la parte imaginaria del exponencial complejo de la FFT. Utiliza desde memoria el valor de π, -1 y el macro *sen*.
- Entradas: Valores de contadores para k, n y valor de N (número de puntos de la FFT).
- Salidas: Coeficiente imaginario SPFP en el registro xmm0.
- Código:

```
%macro coefIm 3
    mov eax, 2
    cvtsi2ss xmm0, eax      ;se guarda un 2 en xmm0
    multiply xmm0, [pi]     ;se multiplica por pi
    multiply xmm0, [negativo];se multiplica por -1


    mov eax, %1 ;k
    cvtsi2ss xmm1, eax
    multiply xmm0, xmm1     ; se multiplica por k
```

```
    mov eax, %2 ;n
    cvtsi2ss xmm1, eax
    multiply xmm0, xmm1      ; se multiplica por n

    mov eax, %3 ;N
    cvtsi2ss xmm1, eax
    divide xmm0, xmm1        ; se divide entre N

    movss xmm10, xmm0        ; es mejor no usar xmm0 para calcular el seno,
    sen xmm10                ; porque xmm0 se modifica dentro del macro
%endmacro
```

- Diagrama de flujo:



FFT

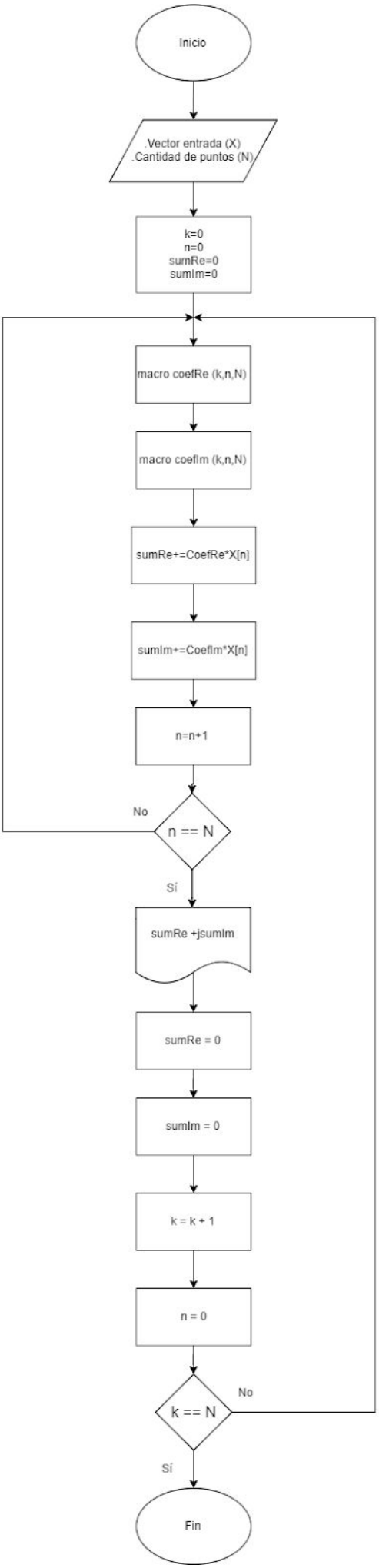
- Función: Calcula e imprime la transformada rápida de fourier de un vector de entrada de números SPFP, como parte real y parte imaginaria, por medio de la multiplicación del exponencial complejo del algoritmo por el vector de entrada. Utiliza los macros *coefRe, coefIm* y *printF*.

- Entradas: Dirección de memoria del vector de entrada previamente almacenado, con valores definidos como doubleword y número de puntos de la FFT.
- Salidas: Impresión en la terminal del resultado de la FFT como vector columna.
- Código:

```
%macro FFT 2
    mov ebx, %2 ;tamaño de las matrices, se mueve a rbx porque rbx no se ve afectado por llamadas

    mov r12d, 0 ;Contador para  k
    mov r13d, 0 ;Contador para  n
    mov r14d, 0 ;Contador para cantidad de X(k) calculados

    pxor xmm4, xmm4 ; se usa xmm4 para almacenar la suma real
    pxor xmm5, xmm5 ; se usa xmm5 para almacenar la suma imaginaria

%%ciclo1:    ;el desplazamiento en el vector de entrada es de 4, por usar doubleword

    ;multiplicacion parte real
    coefRe r12d, r13d, %2       ;se calcula cos(2*pi*k*n/N), se guarda en xmm0
    multiply xmm0,[%1+r13d*4]   ;se multiplica por el elemento del vector
                                ;entrada, resultado se guarda en xmm0
    suma xmm0, xmm4    ;suma el resultado de la multip. con lo que hay en xmm4
    movss xmm4, xmm0            ;guarda eso de nuevo en xmm4

    ;multiplicacion parte imaginaria
    coefIm r12d, r13d, %2      ;se calcula -sen(2*pi*k*n/N), se guarda en xmm0
    multiply xmm0,[%1+r13d*4]   ;resultado se guarda en xmm0
    suma xmm0, xmm5    ;suma el resultado de la multip. con lo que hay en xmm5
    movss xmm5, xmm0            ;guarda eso de nuevo en xmm5

    inc r13d    ;se aumenta n
    cmp r13d, ebx ;cuando rbx sea igual a r13 es porque termino una fila
    jne %%ciclo1

    printF xmm4, [formato1] ;se imprime parte real e imaginaria
    printF xmm5, [formato2]

    pxor xmm4, xmm4 ; se reinicia la suma
    pxor xmm5, xmm5 ; se reinicia la suma
    inc r12d    ;se aumenta k
    mov r13d, 0 ;se reinicia n
```

```
    cmp r12d, ebx ;se verifica si ya se calcularon todos los X(k)
    jne %%ciclo1
%endmacro
```

- Diagrama de flujo:


Inicio
.Vector entrada (X)
.Cantidad de puntos (N)
k=0
n=0
sumRe=0
sumIm=0
macro coefRe (k,n,N)
macro coefIm (k,n,N)
sumRe+=CoefRe*X[n]
sumIm+=CoefIm*X[n]
n=n+1
No
n == N
Sí
sumRe +jsumIm
sumRe = 0
sumIm = 0
k = k + 1
n = 0
k == N
No
Sí
Fin

1.2 Código adicional

Para el correcto funcionamiento de los macros descritos anteriormente, se requiere almacenar valores en memoria, así como hacer llamados a librerías externas. Todo el código adicional, así como el código principal que hace llamado al macro *FFT*, la documentación y comprobación de funcionamiento se encuentra en el repositorio GitHub, disponible en el siguiente enlace:

https://github.com/agutierrez1997/GitArqui/tree/ProyectoFFT

Parte 2. Generación de imagen de Linux

La segunda parte de este proyecto consiste en generar una imagen en Linux mediante el uso de la herramienta Yocto. Dicha imagen se crea a partir de un programa elaborado en ensamblador NASM, es decir, arquitectura x86, donde se utiliza el comando gcc para linkear el archivo de nasm y poder generar un archivo ejecutable, el cual se llama en el simulador QEMU que utiliza Yocto para verificar que la creación de la imagen se realizó de manera efectiva.

El proyecto Yocto brinda la posibilidad de emplear la herramienta “nasm” para compilar el programa elaborado en ensamblador, para ello se debe habilitar el uso de dicha herramienta dentro de los archivos de Yocto, específicamente en los “HOSTTOOLS” dentro del archivo llamado “bitbake.conf” en la carpeta de meta, la cual se encuentra en el poky-warrior.

Luego de haber habilitado nasm en Yocto, se procede a realizar un ejemplo sencillo para comprobar el funcionamiento de la creación de la imagen en Yocto del archivo en nasm, se genera una carpeta principal con el sufijo “meta-”, el cual se debe emplear por definición de Yocto, por lo que está tiene el nombre de “meta-pruebanasm”. Seguidamente, se crean dos carpetas, una llamada conf y otra llamada recipe-helloworld. En la carpeta de conf se crea un archivo llamado layer.conf, el cual va a llevar la siguiente estructura

```
# We have a conf and classes directory, add to BBPATH

BBPATH .= ":${LAYERDIR}"


# We have recipes-* directories, add to BBFILES

BBFILES += "${LAYERDIR}/recipes-*/*.bb \

                ${LAYERDIR}/recipes-*/*/*.bbappend"


BBFILE_COLLECTIONS += "pruebanasm"

BBFILE_PATTERN_pruebanasm = "^${LAYERDIR}/"

BBFILE_PRIORITY_pruebanasm = "5"

LAYERVERSION_pruebanasm = "4"

LAYERSERIES_COMPAT_pruebanasm = "warrior"
```

Una vez guardado el layer.conf, se ingresa a la carpeta de recipe-helloworldnasm, donde se crean una carpeta y un archivo .bb. La carpeta tiene por nombre helloworldnasm-1.0, en la cual se aloja el archivo de nasm, llamado hello.asm. Regresando a la carpeta de recipe-helloworldnasm y editando de la siguiente manera el archivo .bb, el

cual se llamara helloworldnasm.bb

```
# Recipe created by recipetool
# This is the basis of a recipe and may need further editing in order to be ful$
# (Feel free to remove these comments when editing.)

# Unable to find any files that looked like license statements. Check the accom$
# documentation and source headers and set LICENSE and LIC_FILES_CHKSUM accordi$
#
# NOTE: LICENSE is being set to "CLOSED" to allow you to at least start buildin$
# this is not accurate with respect to the licensing of the software being buil$
# will not be in most cases) you must specify the correct value before using th$
# recipe for anything other than initial testing/development!
LICENSE = "CLOSED"
LIC_FILES_CHKSUM = ""

# No information for SRC_URI yet (only an external source tree was specified)
SRC_URI = "file://hello.asm"

# NOTE: no Makefile found, unable to determine what needs to be done

S = "${WORKDIR}"

TARGET_CC_ARCH += "${LDFLAGS}"

do_configure () {
                # Specify any needed configure commands here
```

```
                :
}

do_compile () {
                nasm -f elf32 -o hello.o hello.asm
                ${CC} hello.o -o hello
}

do_install () {
                install -d ${D}${bindir}
                install -m 0755 hello ${D}${bindir}
}
```

Luego de haber creado el archivo .bb que compila e instala el archivo .asm en Yocto, se procede a colocar la ubicación del repositorio meta-pruebanasm en el archivo bblayers,conf del built en el poky warrior. Y por último, se agrega la creación de la imagen, con el nombre de la receta en, en la misma carpeta de built, pero en el archivo llamado local.conf.

Después de realizar lo anterior, se realiza el bitbake de la receta en el source, así como el bitbake del core-image-minimal, para así poder llamar al simulador qemu y a su vez, en este simulador, llamar al ejecutable “hello” que es el que tiene el mensaje de “Hello, World!”.

Para traslapar la solución al problema al simulador de 64 bits, primeramente se elige dicho simulador en el archivo de local.conf de la carpeta built. También se modifica el código de hello.asm para que sea de 64 bits y al archivo .bb dentro de la receta, en el compile, se la agrega el comando -no-pie. Y se realizan los mismos procedimientos del bitbake para la receta y el core-image-minimal.

Parte 3. Demostración de funcionamiento de la aplicación

Para comprobar el funcionamiento del programa implementado, se realizó un script en GNU Octave llamado “VectorEntrada.m” que genera un vector de entrada aleatorio del tamaño indicado y además, imprime este vector con la sintaxis adecuada para ser copiado previamente al código en lenguaje ensamblador para el cálculo de la FFT. Este script se muestra a continuación:

```
function[X]=VectorEntrada(N)
 X=[]; %Vector de entrada para la fft
 cant=1; %para elegir la cantidad que se desea por fila, solo para que se vea
mejor en el codigo
 fprintf("X:\n\t dd ");
 for n=0:N-1
    X(n+1,1)=(15*rand);
    fprintf("%f",X(n+1,1));
    if n~=(N-1)
      if (cant==9)
        cant=0;
        fprintf("\n\t dd ")
      else
        fprintf(", ")
      end
    else
      fprintf("\n")
    end
    cant=cant+1;
 end
 end
```

Con el script anterior, se calcularon tres vectores de entrada de 16, 256 y 1024 elementos para realizar tres pruebas de funcionamiento. El programa utilizado para el cálculo llamado “NASM_FFT.asm” se encuentra disponible en el repositorio GitHub. A continuación se muestran los resultados de cada una de las pruebas:

3.1 Primera prueba

A continuación se muestra el vector de entrada de 16 elementos utilizado para la primera prueba:

```
X =
        0.20248
        0.42656
        14.63102
        1.86529
```

0.86237
13.43935
13.08786
12.46647
4.69287
7.15233
7.55647
13.31624
1.57023
2.11285
9.81936
8.60641

Con este vector de entrada se obtuvo el siguiente resultado con el programa creado en lenguaje ensamblador:

1.1181e+02 +0.0000e+00j
-2.0312e+01 -5.4460e+00j
1.1711e+00 +1.0504e+01j
9.7996e+00 -5.4632e+00j
-3.7611e+01 +1.2913e+01j
-2.4144e+01 +1.0585e+01j
3.9650e+00 +9.0954e+00j
1.6710e+01 +7.8751e+00j
-6.9629e+00 +8.8651e-02j
1.6728e+01 -7.7974e+00j
3.9912e+00 -9.0836e+00j
-2.4166e+01 -1.0585e+01j
-3.7673e+01 -1.3280e+01j
9.8068e+00 +5.4632e+00j
9.6584e-01 -1.0530e+01j
-2.0304e+01 +5.3685e+00j

Por otro lado, comprobando con el comando fft() de GNU Octave, se tiene el siguiente resultado para el mismo vector de entrada:

111.80817 + 0.00000i
-20.29569 - 5.39415i
0.99058 + 10.52352i
9.81092 - 5.42157i
-37.76677 + 13.12333i
-24.17427 + 10.62144i
3.93491 + 9.08406i
16.69747 + 7.81742i
-6.96285 + 0.00000i
16.69747 - 7.81742i
3.93491 - 9.08406i
-24.17427 - 10.62144i
-37.76677 - 13.12333i
9.81092 + 5.42157i
0.99058 - 10.52352i

-20.29569 + 5.39415i

En la prueba anterior es posible observar el correcto funcionamiento del programa, con cierto margen de error debido a la aproximación de senos y cosenos con polinomio de Taylor que se realiza.

3.2 Segunda prueba

A continuación, se tiene el vector de entrada de 256 elementos utilizado para la segunda prueba. Para este caso, se muestran en este documento los primeros 16 elementos del vector, así como los primeros 16 elementos de los resultados. El vector de entrada y resultados completos están disponibles en el repositorio GitHub.

X =

11.915219
2.696423
6.223795
0.336657
4.378528
14.662405
12.037341
7.055432
3.407904
11.578168
8.331297
0.860811
6.331554
14.089702
9.722978
10.534180
...

Con el vector de entrada anterior, se obtuvo el siguiente resultado con el programa creado (se muestran los primeros 16 elementos):

1.9461e+03 +0.0000e+00j
6.1262e+01 +3.7552e+00j
2.9544e+01 +5.4351e+01j
-5.4673e+01 +8.7583e+01j
3.4036e+01 -5.6269e-03j
-1.3658e+01 -3.7513e+01j
-1.8197e+01 +2.7040e+01j
-2.6345e-01 -6.9220e+01j
3.0149e+01 +3.5711e+01j
-7.6425e+01 +1.5971e+01j
-3.5853e+01 -4.7026e+01j
8.9127e+01 +5.7974e+01j
-4.2764e+01 -4.9169e+01j

-9.0242e+01 -6.3108e+01j
-2.7970e+00 +5.1987e+01j
8.2627e+01 +5.1370e+01j
…

Comprobando el resultado con GNU Octave, se obtiene lo siguiente (primeros 16 elementos):

1.9461e+03 + 0.0000e+00i
6.1348e+01 + 3.8786e+00i
2.9382e+01 + 5.4468e+01i
-5.4937e+01 + 8.7603e+01i
3.3875e+01 + 1.4489e-01i
-1.3603e+01 - 3.7444e+01i
-1.8256e+01 + 2.7085e+01i
-3.0632e-01 - 6.9173e+01i
3.0316e+01 + 3.5540e+01i
-7.6666e+01 + 1.6069e+01i
-3.5795e+01 - 4.6910e+01i
8.9011e+01 + 5.7861e+01i
-4.2859e+01 - 4.9029e+01i
-9.0141e+01 - 6.3179e+01i
-2.6468e+00 + 5.2035e+01i
8.2387e+01 + 5.1297e+01i
…

3.3 Tercera prueba

A continuación, se tiene, como en el caso anterior, los primeros 16 elementos del vector de entrada de 1024 elementos utilizado para la tercera prueba:

X =

9.974645
9.341211
1.895943
14.121372
9.447419
14.789702
14.440906
12.061343
12.951904
10.968718
2.466296
4.515795
10.697916
8.250674
8.517256
11.078942

...

El resultado obtenido con el programa es el siguiente (primeros 16 elementos):

```
7.5725e+03    +0.0000e+00j
1.6087e+02    -6.5316e+01j
1.0155e+02    +1.2347e+02j
6.0377e+00    -1.6934e+02j
9.6634e+00    +3.6209e+01j
-4.1085e+01    +3.7824e+01j
-3.6408e+01    +2.1551e+01j
1.5616e+02    -7.8844e+01j
4.9933e+01    -6.5156e+01j
2.6880e+01    -9.3681e+01j
-5.5480e+00    -1.3788e+02j
9.3828e+01    -7.5849e+01j
5.7781e+01    +1.4293e+02j
-1.7990e+02    -4.5062e+01j
6.6230e+01    -2.0490e+01j
-7.4967e+01    -6.2659e+01j
…
```

A partir de GNU Octave, se obtiene para el mismo vector de entrada el siguiente resultado (primeros 16 elementos):

```
 7.5725e+03 + 0.0000e+00i
 1.6092e+02 - 6.5183e+01i
 1.0138e+02 + 1.2353e+02i
 6.0369e+00 - 1.6940e+02i
9.3972e+00 + 3.6292e+01i
-4.1172e+01 + 3.7686e+01i
-3.6132e+01 + 2.1821e+01i
 1.5596e+02 - 7.9199e+01i
 4.9823e+01 - 6.5168e+01i
2.6579e+01 - 9.3699e+01i
-5.2946e+00 - 1.3788e+02i
9.3971e+01 - 7.6039e+01i
5.7540e+01 + 1.4307e+02i
-1.7996e+02 - 4.5507e+01i
6.6438e+01 - 2.0323e+01i
-7.4853e+01 - 6.2479e+01i
…
```

Nuevamente, se observa el correcto funcionamiento del algoritmo implementado, con un margen de error debido a la aproximación de funciones seno y coseno.