[SoC Design]

SW Design for FFT (Lab1)

Chester Sungchung Park (박성정)

SoC Design Lab, Konkuk University

Webpage: http://soclab.konkuk.ac.kr

Teaching Assistants

- Youngho Seo (younghoseo@konkuk.ac.kr), M.S. candidate
- Sanghun Lee (sanghunlee@konkuk.ac.kr), M.S. candidate

Outline

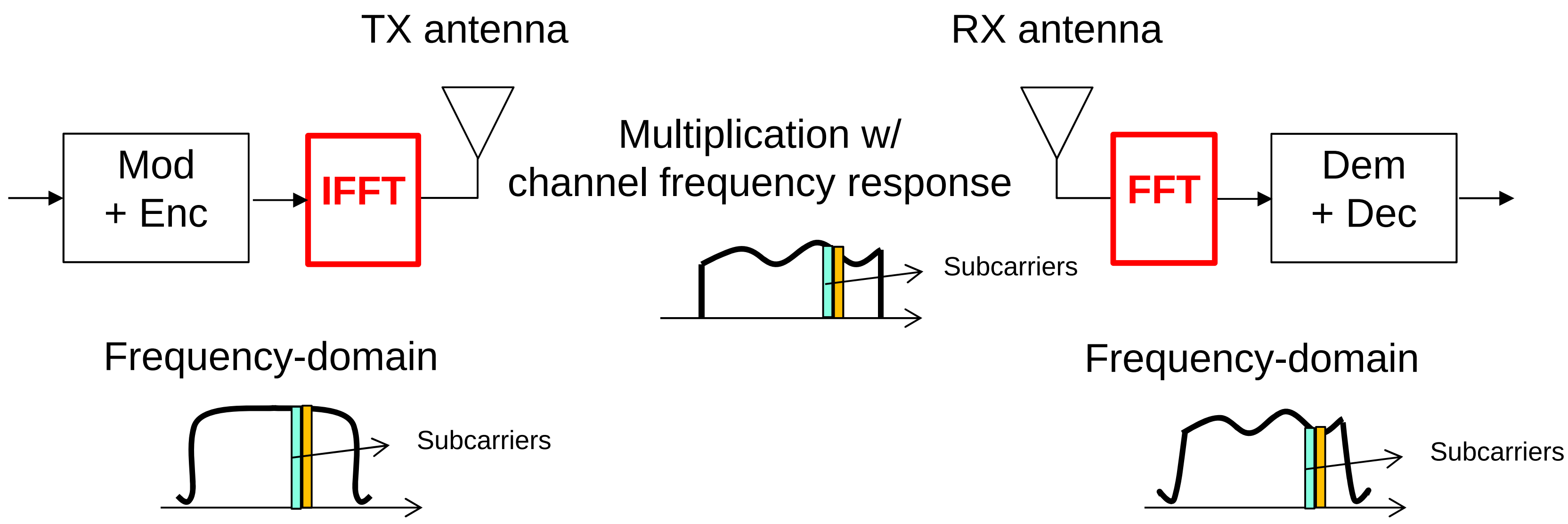
- Objectives
- Fast Fourier transform (FFT)
- Lab1: SW design

Objectives

- After completing this lab, you will be able to :
  - Program an application in either C or assembly
  - Run an application
  - Debug an application
  - Measure the execution time of an application
  - Optimize the performance of an application in either C or assembly

Introduction

- Orthogonal Frequency Division Multiplexing (OFDM)
  - FFT processor as major enabler



* The basic principle was first conceived: M. L. Doeiz, E. T. Heald and D. L. Martin, "Binary data transmission techniques for linear systems", Proc. IRE, vol. 45, pp. 656-661, May 1957.

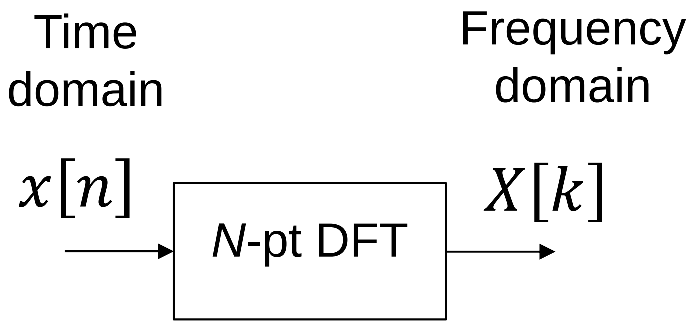
Introduction

- Orthogonal Frequency Division Multiplexing (OFDM)
  - WiFi (802.11a)
    - ✓ FFT/IFFT size: 64 samples
    - ✓ Sampling rate: 20 Msamples/sec
    - ✓ Processing time: 3.2 usec = 64 samples / 20 Msamples/sec
  - LTE (Rel-8)
    - ✓ FFT/IFFT size: 128/256/512/1024/2048 samples*
    - ✓ Sampling rate: 1.92/3.84/7.68/15.36/30.72 Msamples/sec
    - ✓ Processing time: 66.6 usec = 128 samples / 1.92 Msamples/sec

* In the uplink, the FFT/IFFT of various sizes (multiplies of 2, 3 and 5) should be implement to support SC-FDMA (DFTS-OFDM) in addition to the FFT/IFFT that support OFDM.

Discrete Fourier Transform (DFT)

❑ Mathematical expression



$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, k = 0, \cdots, N-1$$

$$W_N^{nk} := \exp\left(-j\frac{2\pi}{N}nk\right) \text{ (Twiddle factor)}$$

Converting a discrete-time signal ($x[n]$)
from time domain to frequency domain

Discrete Fourier Transform (DFT)

- Example: 8-pt DFT ($N = 8$)
  - Mathematical expression

$$X[k] = \sum_{n=0}^{7} x[n] W_8^{nk}, k = 0, \cdots, 7$$

$$W_8^{nk} := \exp(-j \frac{2\pi}{8} nk)$$

$$\Updownarrow$$

$$X[0] = x[0]W_8^0 + x[1]W_8^0 + x[2]W_8^0 + x[3]W_8^0 + x[4]W_8^0 + x[5]W_8^0 + x[6]W_8^0 + x[7]W_8^0$$

$$X[1] = x[0]W_8^0 + x[1]W_8^1 + x[2]W_8^2 + x[3]W_8^3 + x[4]W_8^4 + x[5]W_8^5 + x[6]W_8^6 + x[7]W_8^7$$

$$X[2] = x[0]W_8^0 + x[1]W_8^2 + x[2]W_8^4 + x[3]W_8^6 + x[4]W_8^8 + x[5]W_8^{10} + x[6]W_8^{12} + x[7]W_8^{14}$$

$$X[3] = x[0]W_8^0 + x[1]W_8^3 + x[2]W_8^6 + x[3]W_8^9 + x[4]W_8^{12} + x[5]W_8^{15} + x[6]W_8^{18} + x[7]W_8^{21}$$

$$X[4] = x[0]W_8^0 + x[1]W_8^4 + x[2]W_8^8 + x[3]W_8^{12} + x[4]W_8^{16} + x[5]W_8^{20} + x[6]W_8^{24} + x[7]W_8^{28}$$

$$X[5] = x[0]W_8^0 + x[1]W_8^5 + x[2]W_8^{10} + x[3]W_8^{15} + x[4]W_8^{20} + x[5]W_8^{25} + x[6]W_8^{30} + x[7]W_8^{35}$$

$$X[6] = x[0]W_8^0 + x[1]W_8^6 + x[2]W_8^{12} + x[3]W_8^{18} + x[4]W_8^{24} + x[5]W_8^{30} + x[6]W_8^{36} + x[7]W_8^{42}$$

$$X[7] = x[0]W_8^0 + x[1]W_8^7 + x[2]W_8^{14} + x[3]W_8^{21} + x[4]W_8^{28} + x[5]W_8^{35} + x[6]W_8^{42} + x[7]W_8^{49}$$

Discrete Fourier Transform (DFT)

- Computational complexity
  - Totally N^2 multiplications plus $N(N-1)$ additions
    - ✓ No reuse of intermediate calculations assumed
    - ✓ No use of twiddle factor properties assumed
    - ✓ Multiplication with unity included

	Multiplications	Additions
4	16	12
8	64	56
16	256	240
64	1024	992
N	N^2	$N(N-1)$

Discrete Fourier Transform (DFT)

❑ Computational complexity (cont'd)

- E.g., 8-pt DFT ($N = 8$): 64 multiplications plus 56 additions

$$X[0] = x[0]W_8^0 + x[1]W_8^0 + x[2]W_8^0 + x[3]W_8^0 + x[4]W_8^0 + x[5]W_8^0 + x[6]W_8^0 + x[7]W_8^0$$
$$X[1] = x[0]W_8^0 + x[1]W_8^1 + x[2]W_8^2 + x[3]W_8^3 + x[4]W_8^4 + x[5]W_8^5 + x[6]W_8^6 + x[7]W_8^7$$
$$X[2] = x[0]W_8^0 + x[1]W_8^2 + x[2]W_8^4 + x[3]W_8^6 + x[4]W_8^8 + x[5]W_8^{10} + x[6]W_8^{12} + x[7]W_8^{14}$$
$$X[3] = x[0]W_8^0 + x[1]W_8^3 + x[2]W_8^6 + x[3]W_8^9 + x[4]W_8^{12} + x[5]W_8^{15} + x[6]W_8^{18} + x[7]W_8^{21}$$
$$X[4] = x[0]W_8^0 + x[1]W_8^4 + x[2]W_8^8 + x[3]W_8^{12} + x[4]W_8^{16} + x[5]W_8^{20} + x[6]W_8^{24} + x[7]W_8^{28}$$
$$X[5] = x[0]W_8^0 + x[1]W_8^5 + x[2]W_8^{10} + x[3]W_8^{15} + x[4]W_8^{20} + x[5]W_8^{25} + x[6]W_8^{30} + x[7]W_8^{35}$$
$$X[6] = x[0]W_8^0 + x[1]W_8^6 + x[2]W_8^{12} + x[3]W_8^{18} + x[4]W_8^{24} + x[5]W_8^{30} + x[6]W_8^{36} + x[7]W_8^{42}$$
$$X[7] = x[0]W_8^0 + x[1]W_8^7 + x[2]W_8^{14} + x[3]W_8^{21} + x[4]W_8^{28} + x[5]W_8^{35} + x[6]W_8^{42} + x[7]W_8^{49}$$

Fast Fourier Transform (FFT)

- FFT is nothing but an efficient algorithm of calculating DFT.
  - DFT & FFT generate exactly the same results
- FFT reduces the computational complexity of DFT by using the properties of twiddle factor
  - Complex conjugate symmetry

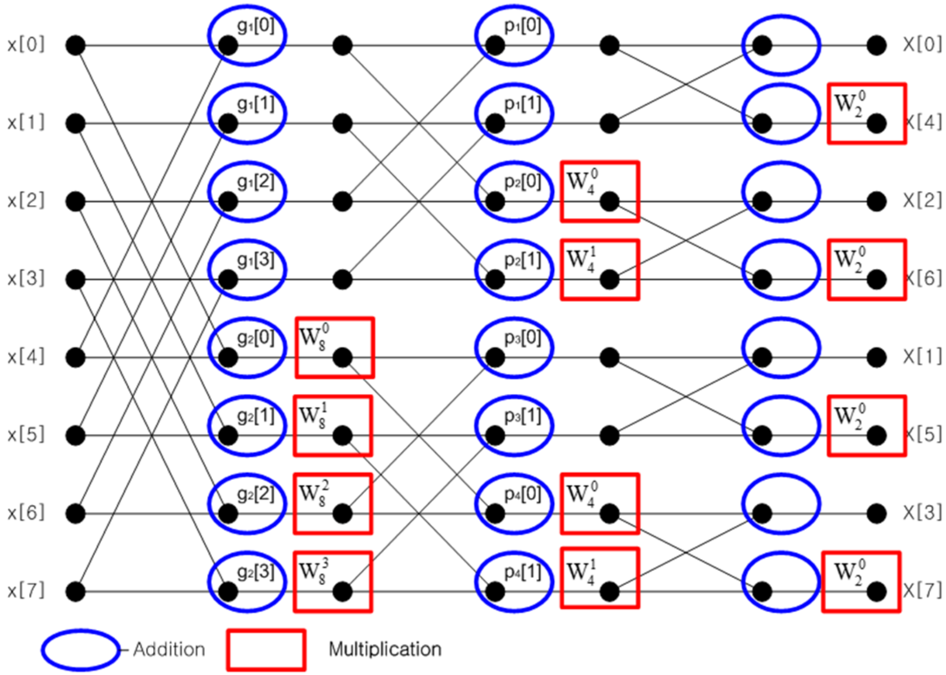
    $$W_N^{(N-n)k} = W_N^{-nk} = (W_N^{nk})^*$$

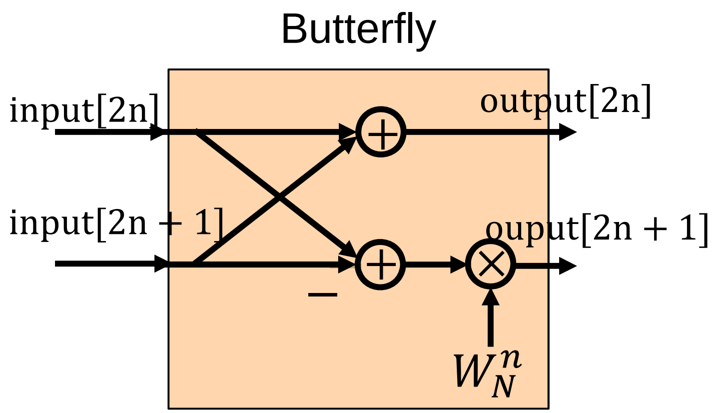
  - Periodicity

    $$W_N^{nk} = W_N^{(n+N)k} = W_N^{n(k+N)}$$

Fast Fourier Transform (FFT)

- Radix-2 decimation-in-frequency (DIF)
  - Example: 8-pt DFT ($N = 8$)





12 multiplications & 24 additions

Fast Fourier Transform (FFT)

- Computational complexity

	DFT		FFT	
	Multiplication	Addition	Multiplication	Addition
4	16	12	4	8
8	64	56	12	24
16	256	240	32	64
64	4096	4032	192	384
N	N^2	$N(N-1)$	$(N/2)\, log_2 N$	$N\, log_2 N$

Complexity reduction by $\sim \frac{log_2 N}{2N}$

Lab 1: SW Design

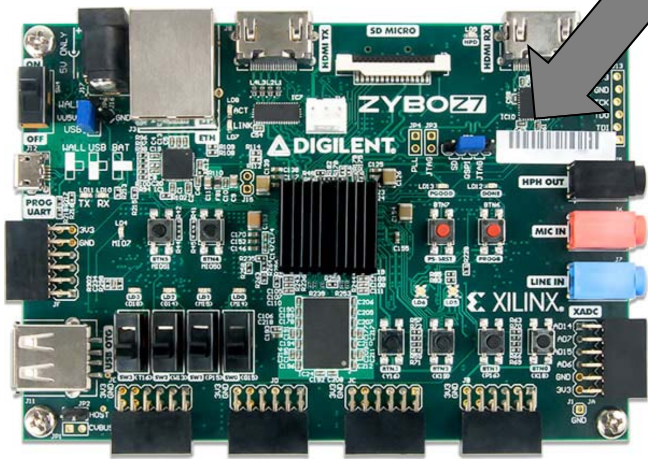
- Design flow

Vivado

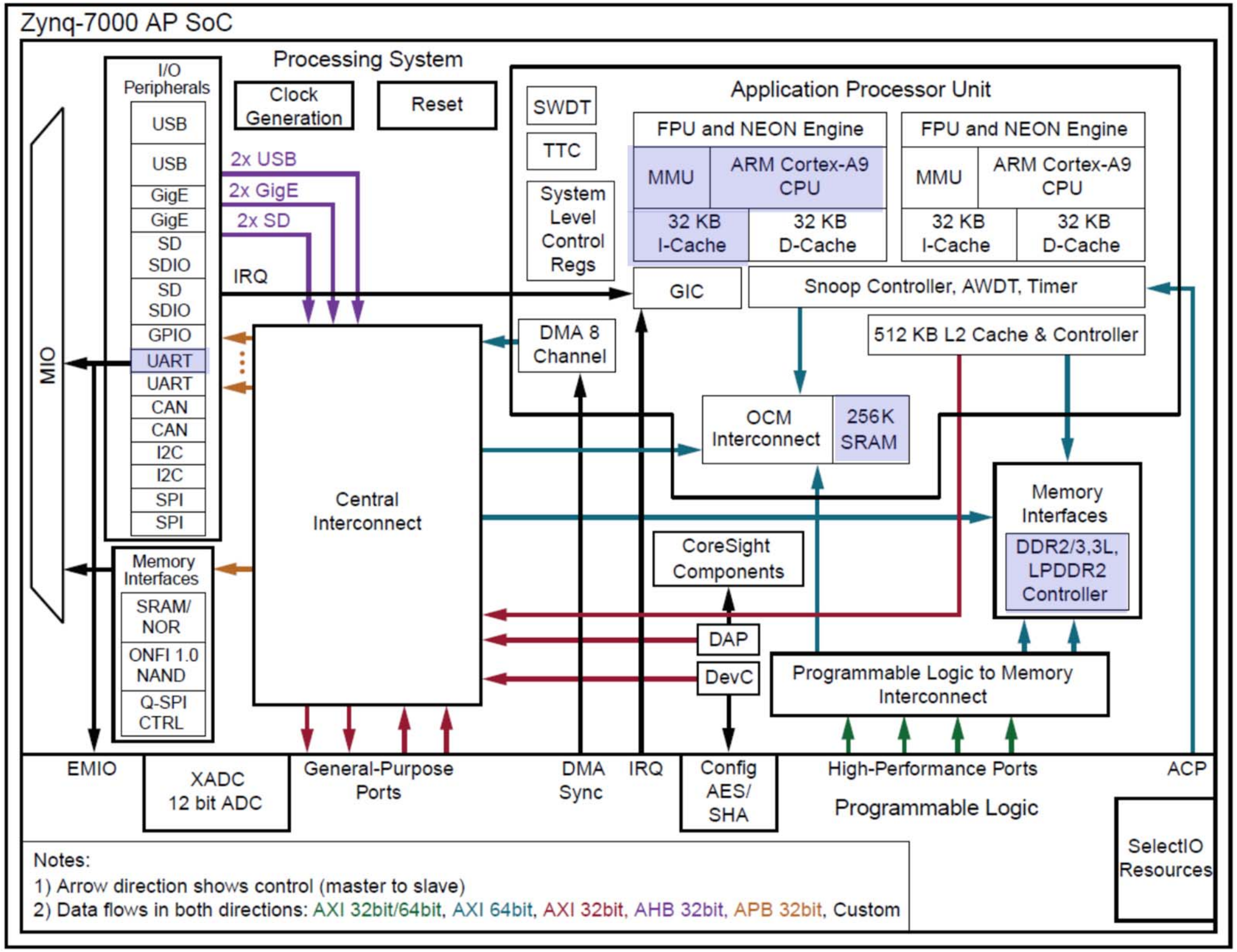
Used in this Lab

SDK

ZYNQ/Zybo



Block Diagram



Source Codes

❑ main

① Calls FFT_Assembly

② Compares the output of FFT with that of DFT

```c
int main() {
    XTime start,stop;
    int i = 0;

    float error_total, error_real, error_imag;
    float sig_total;
    float SNR;

    XTime_GetTime((XTime*)&start);
    DFT();
    XTime_GetTime((XTime*)&stop);
    printf("DFT          %8.3f us\n",((float)stop - (float)start)/COUNTS_PER_SECOND*1000000);

    XTime_GetTime((XTime*)&start);
    FFT();
    XTime_GetTime((XTime*)&stop);
    printf("FFT          %8.3f us\n",((float)stop - (float)start)/COUNTS_PER_SECOND*1000000);

    XTime_GetTime((XTime*)&start);
    FFT_Assembly();
    XTime_GetTime((XTime*)&stop);
    printf("FFT Assembly %8.3f us\n",((float)stop - (float)start)/COUNTS_PER_SECOND*1000000);


    error_total = 0;
    sig_total = 0;
    for(i = 0; i<N; i++){
        error_real = (X_DFT[i].re)-(X_FFT[i].re);
        error_imag = (X_DFT[i].im) -(X_FFT[i].im);

        error_total += error_real*error_real + error_imag*error_imag;

        sig_total += (X_DFT[i].re)*(X_DFT[i].re) + (X_DFT[i].im)*(X_DFT[i].im);
    }
    SNR = 10*log10(sig_total/error_total);
    xil_printf("FFT model SNR : %d dB\n",(int)SNR);


    error_total = 0;
    sig_total = 0;
    for(i = 0; i<N; i++){
        error_real = (X_DFT[i].re)-(X_FFT_Assembly[i].re);
        error_imag = (X_DFT[i].im) -(X_FFT_Assembly[i].im);

        error_total += error_real*error_real + error_imag*error_imag;

        sig_total += (X_DFT[i].re)*(X_DFT[i].re) + (X_DFT[i].im)*(X_DFT[i].im);
    }
    SNR = 10*log10(sig_total/error_total);
    xil_printf("FFT Assembly model SNR : %d dB\n",(int)SNR);

    return 0;
}
```

Source Codes

❑ DFT

① Takes the input (from header)
② Performs DFT
③ Generates the output

```
void DFT()
{
    int n = 0, i = 0 ,k = 0;

    complex input[N];
    complex temp_mult[N];

    int out_re[N] = {0,};
    int out_im[N] = {0,};

    for (n=0; n<N; n++)                                          ①
    {
        input[n].re=in_real[n];
        input[n].im=in_imag[n];
    }

    for (i=0; i<N; i++)                                          ②
    {
        X_DFT[i] = add_cal(init1_int,init2_int);
        for (k=0; k<N; k++)
        {
            temp_mult[k] = multiple(input[k],W[(k*i)%64]);
            X_DFT[i] = add_cal(temp_mult[k],X_DFT[i]);
        }
    }

    for (n=0; n<N; n++)                                          ③
    {
        out_re[n] = X_DFT[n].re;
        out_im[n] = X_DFT[n].im;
    }

}
```

Source Codes

❑ FFT

① Butterfly: Stage 1 ~ Stage 5
② Butterfly: Stage 6 (incomplete)
③ Reordering

```c
#ifndef STAGE6_H_
#define STAGE6_H_


void Stage6(complex *output, complex *input)
{
    int n;

    ////////////////////////////////////////////
    //
    //
    //      Fill Your Code Here.
    //
    //
    ////////////////////////////////////////////
}



#endif /* STAGE6_H_ */
```

```c
void FFT()
{
    complex input[N],temp[N];

    int out_re[N];
    int out_im[N];

    int data;
    int n,k;
    for (data=0;data<N;data++)
    {
        input[data].re=in_real[data];
        input[data].im=in_imag[data];
    }

    for (n=0;n<32;n++) //stage1
    {
        temp[n]=add_cal(input[n],input[n+32]);
        temp[n+32]=multiple(sub_cal(input[n],input[n+32]),W[n]);
    }

    for (n=0;n<16;n++) //stage2
    {
        for (k=0;k<2;k++)
        {
            input[n+(32*k)]=add_cal(temp[n+(32*k)],temp[n+((32*k)+16)]);
            input[n+((32*k)+16)]=multiple(sub_cal(temp[n+(32*k)],temp[n+((32*k)+16)]),W[2*n]);
        }
    }

    for(n=0;n<8;n++) //stage-3
    {
        for (k=0;k<4;k++)
        {
            temp[n+(16*k)] = add_cal(input[n+(16*k)],input[n+((16*k)+8)]);
            temp[n+((16*k)+8)] = multiple(sub_cal(input[n+(16*k)],input[n+((16*k)+8)]),W[4*n]);
        }
    }

    for (n=0;n<4;n++) //stage4
    {
        for (k=0;k<8;k++)
        {
            input[n+(8*k)] = add_cal(temp[n+(8*k)],temp[n+((8*k)+4)]);
            input[n+((8*k)+4)] =multiple(sub_cal(temp[n+(8*k)],temp[n+((8*k)+4)]),W[8*n]);
        }
    }

    for (n=0;n<2;n++) //stage5
    {
        for (k=0;k<16;k++)
        {
            temp[n+(4*k)]=add_cal(input[n+(4*k)],input[n+(4*k+2)]);
            temp[n+(4*k+2)]=multiple(sub_cal(input[n+(4*k)],input[n+(4*k+2)]),W[16*n]);
        }

    }
    // Stage 6
    Stage6(input, temp);

    for(n=0;n<N;n++)
    {
        X_FFT[n]=input[Re_ordering(n)];
    }

    for (n=0;n<N;n++)
    {
        out_im[n]=(X_FFT[n].im)>>10;
        out_re[n]=(X_FFT[n].re)>>10;
    }
}
```

①

②

③

Source Codes

❑ FFT_Assembly

① Butterfly: Stage 1 ~ Stage 5

② Butterfly: Stage 6 (incomplete)

③ Reordering

```
        .text
        .syntax   unified

        .align  4
        .global Stage6_Assembly
        .arm


Stage6_Assembly:
    ////////////////////////////////////////////
    //
    //
    //      Fill Your Code Here.
    //
    //
    ////////////////////////////////////////////
```

```c
void FFT_Assembly()
{
    complex input[N],temp[N];

    int out_re[N];
    int out_im[N];

    int data;
    int n,k;

    for (data=0;data<N;data++)
    {
        input[data].re=in_real[data];
        input[data].im=in_imag[data];
    }

    for (n=0;n<32;n++) //stage1
    {
        temp[n]=add_cal(input[n],input[n+32]);
        temp[n+32]=multiple(sub_cal(input[n],input[n+32]),W[n]);
    }

    for (n=0;n<16;n++) //stage2
    {
        for (k=0;k<2;k++)
        {
            input[n+(32*k)]=add_cal(temp[n+(32*k)],temp[n+((32*k)+16)]);
            input[n+((32*k)+16)]=multiple(sub_cal(temp[n+(32*k)],temp[n+((32*k)+16)]),W[2*n]);
        }
    }

    for(n=0;n<8;n++) //stage-3
    {
        for (k=0;k<4;k++)
        {
            temp[n+(16*k)] = add_cal(input[n+(16*k)],input[n+((16*k)+8)]);
            temp[n+((16*k)+8)] = multiple(sub_cal(input[n+(16*k)],input[n+((16*k)+8)]),W[4*n]);
        }
    }

    for (n=0;n<4;n++) //stage4
    {
        for (k=0;k<8;k++)
        {
            input[n+(8*k)] = add_cal(temp[n+(8*k)],temp[n+((8*k)+4)]);
            input[n+((8*k)+4)] =multiple(sub_cal(temp[n+(8*k)],temp[n+((8*k)+4)]),W[8*n]);
        }
    }

    for (n=0;n<2;n++) //stage5
    {
        for (k=0;k<16;k++)
        {
            temp[n+(4*k)]=add_cal(input[n+(4*k)],input[n+(4*k+2)]);
            temp[n+(4*k+2)]=multiple(sub_cal(input[n+(4*k)],input[n+(4*k+2)]),W[16*n]);
        }

    }
```

①

```c
    // Stage 6
    Stage6_Assembly(input, temp);
```

②

```c
    for(n=0;n<N;n++)
    {
        X_FFT_Assembly[n]=input[Re_ordering(n)];
    }

    for (n=0;n<N;n++)
    {
        out_im[n]=(X_FFT_Assembly[n].im)>>10;
        out_re[n]=(X_FFT_Assembly[n].re)>>10;
    }

}
```

③

Source Codes

❑ Stage6_Assembly.s

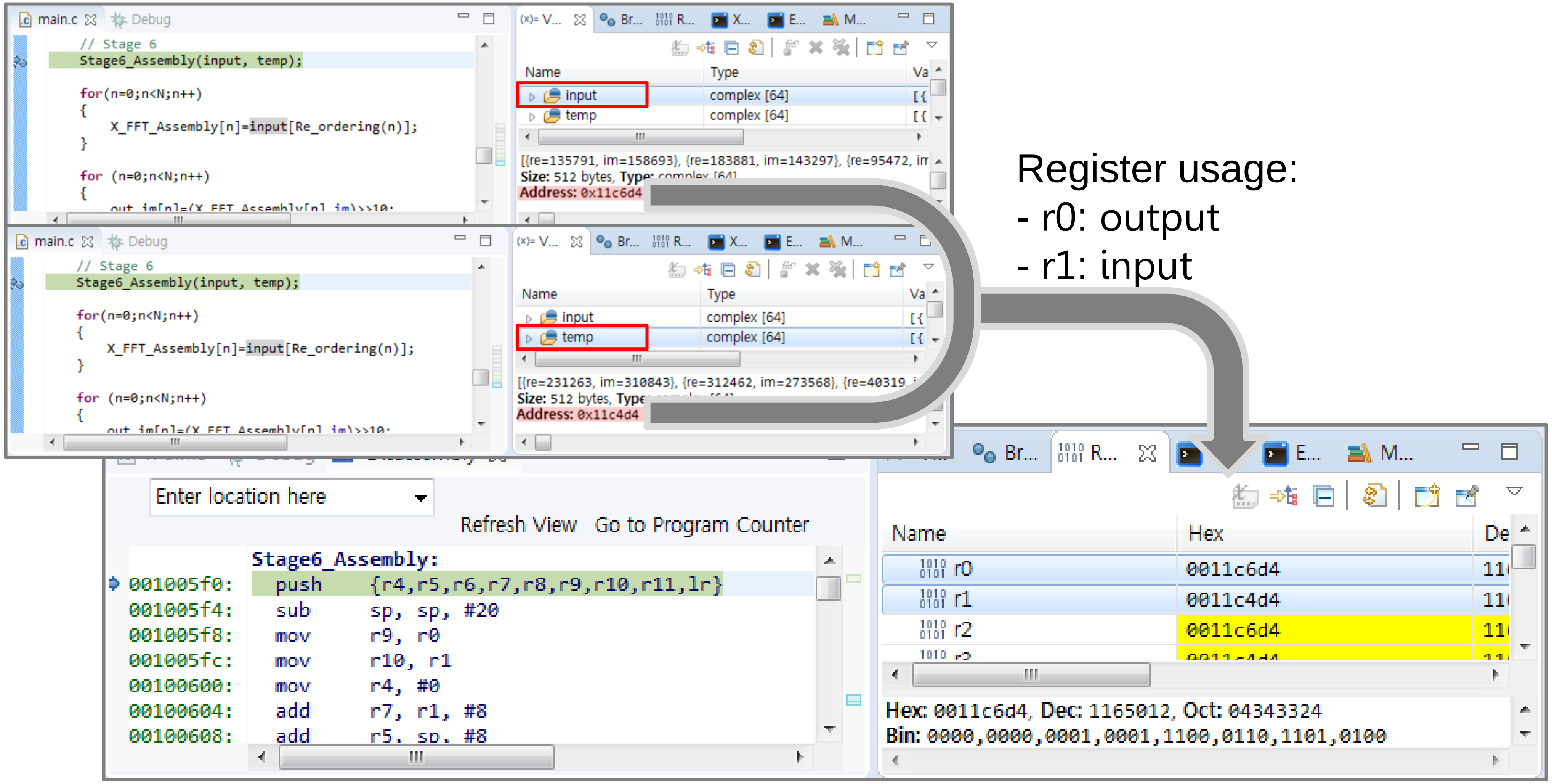
```c
#include <stdio.h>
#include <xtime_l.h>
#include <xil_cache.h>
#include <math.h>
#include "FFT_Header.h"
#include "Stage6.h"

#define N 64

complex X_DFT[N];
complex X_FFT[N];
complex X_FFT_Assembly[N];

int time;

//Function
extern void Stage6_Assembly(complex *output, complex *input);
extern void Stage6(complex *output, complex *input);

int Re_ordering(int x) {
    return 32 * (x % 2) + 16 * ((x % 4) / 2) + 8 * ((x % 8) / 4)
            + 4 * ((x % 16) / 8) + 2 * ((x % 32) / 16) + x / 32;
}
```

```asm
        .text
        .syntax   unified

        .align  4
        .global Stage6_Assembly
        .arm


Stage6_Assembly:
        push {r4, r5, r6, r7, r8, r9, r10, r11, lr}
        sub sp, sp, #20
        mov r9, r0
        mov r10, r1
        mov r4, #0
        add r7, r1, #8
        add r5, sp, #8
        add r6, r4, r10
        add r8, r7, r4
        ldr r3, [r8, #4]
        str r3, [sp]
        ldr r3, [r7, r4]
        mov r0, r5
        ldm r6, {r1, r2}
        bl add_cal_assembly
        add r3, r4, r9
        ldm r5, {r0, r1}
```

Register usage:
- r0: output
- r1: input

Source Codes

❑ Stage6_Assembly.s



Register usage:
- r0: output
- r1: input

Optimization Levels

- Compiler optimization levels (ARM gcc)
  - **-O0**: No optimization is performed.
  - -O1: Enables the most common forms of optimization that do not require decisions regarding size or speed.
  - -O2: Enables further optimizations, such as instruction scheduling.
  - **-O3**: Enables more aggressive optimizations, such as aggressive function inlining, and it typically increases speed at the expense of image size. Moreover, this option enables -ftree-vectorize, causing the compiler to attempt to automatically generate NEON code.
  - -Os: Selects optimizations that attempt to minimize the size of the image, even at the expense of speed.