Lecture 3. FFT

Hyeong-Ju Kang

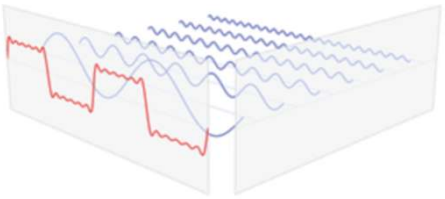
Outline

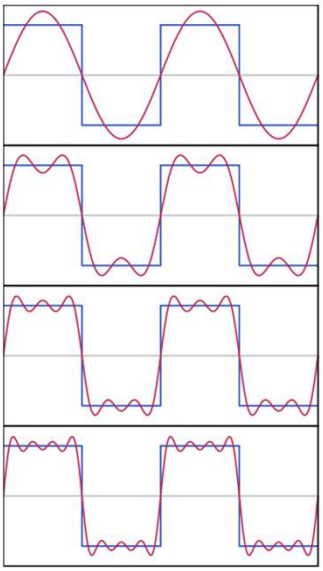
- Fast Fourier Transform (FFT)
- C Implementation
- Fixed Point Implementation
- Verilog Implementation – Memory-based
- Verilog Implementation – Pipelined

Hyeong-Ju Kang

From Fourier Series to FFT

- Fourier series
  - A periodic function can be composed of harmonically related sinusoids combined by a weighted summation.





Fourier Series

- Fourier series
  - A periodic function can be composed of harmonically related sinusoids combined by a weighted summation.

Trigonometric Form

$$f(t) = \frac{A_0}{2} + \sum_{n=1}^{\infty} (A_n \cos n\omega_0 t + B_n \sin n\omega_0 t)$$

$$A_0 = \frac{2}{T}\int_{t_0}^{t_0+T} f(t)\, dt$$

$$A_n = \frac{2}{T}\int_{t_0}^{t_0+T} f(t) \cos -n\omega_0 t \; dt$$

$$B_n = \frac{2}{T}\int_{t_0}^{t_0+T} f(t) \sin -n\omega_0 t \; dt$$

Complex Exponential Form

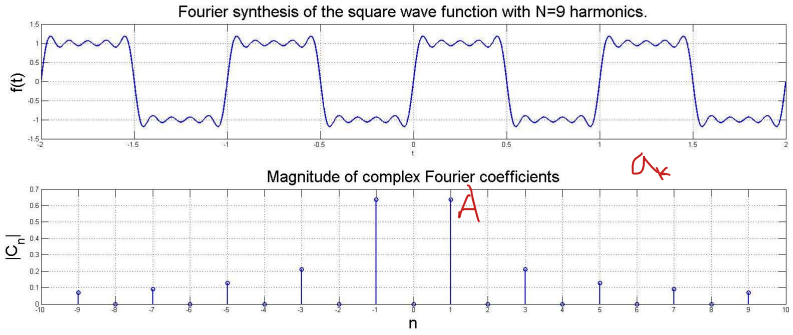
$$f(t) = \sum_{n=-\infty}^{\infty} \alpha_n e^{jn\omega_0 t}$$

$$\alpha_n = \frac{2}{T}\int_{t_0}^{t_0+T} f(t) e^{-jn\omega_0 t} dt$$

$$\omega_0 = \frac{2\pi}{T}$$

Fourier Series

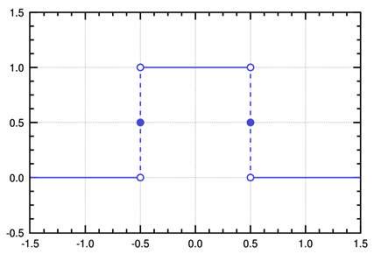
- Fourier series
  - A periodic function can be composed of harmonically related sinusoids combined by a weighted summation.

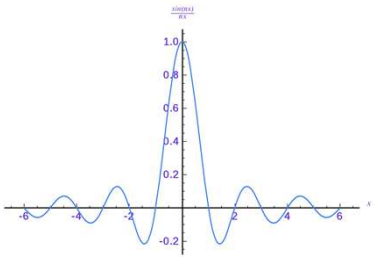


Fourier Transform

- Fourier transform
  - Non-periodic version of Fourier series
  - continuous time domain ←→ continuous frequency domain

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt$$

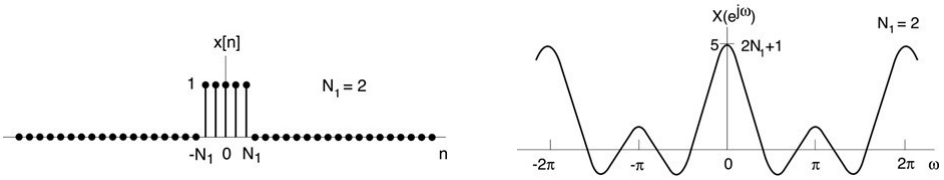




Discrete Time Fourier Transform

- Discrete time Fourier transform
  - for discrete time non-periodic functions
  - discrete time domain ←→ continuous frequency domain

$$X(\omega) = \sum_{k=0}^{\infty} x[k]e^{-j\omega k}$$



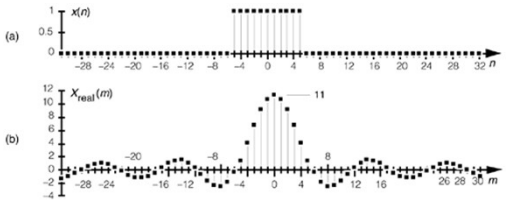
Discrete Fourier Transform

- Discrete Fourier transform
  - for discrete time periodic functions
  - discrete time domain ←→ discrete frequency domain

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

$$W_N = e^{-j2\pi/N}$$

  - N^2 complex multiplications/additions



Fast Fourier Transform

- Fast Fourier transform
  - to reduce the number of operations

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

$$= \sum_{n\ even}^{N/2-1} x[n]W_N^{nk} + \sum_{n\ odd}^{N/2-1} x[n]W_N^{nk}$$

$$= \sum_{r=0}^{N/2-1} x[2r]W_N^{2rk} + \sum_{r=0}^{N/2-1} x[2r+1]W_N^{(2r+1)k}$$

$$= \sum_{r=0}^{N/2-1} x[2r](W_N^2)^{rk} + W_N^k \sum_{r=0}^{N/2-1} x[2r+1](W_N^2)^{rk}$$

$$= \sum_{r=0}^{(\frac{N}{2})-1} x[2r]W_{N/2}^{rk} + W_N^k \sum_{r=0}^{(\frac{N}{2})-1} x[2r+1]W_{N/2}^{rk}$$

$$W_N = e^{-j2\pi/N}$$

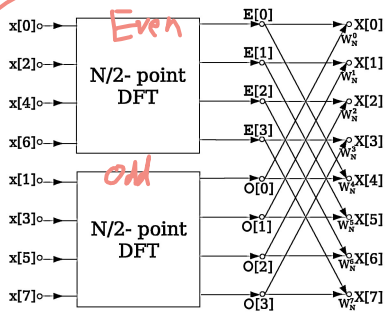
$$W_N^2 = e^{-2j2\pi/N} = e^{-j2\pi/(\frac{N}{2})} = W_{N/2}$$

Fast Fourier Transform

- Fast Fourier transform
  - to reduce the number of operations

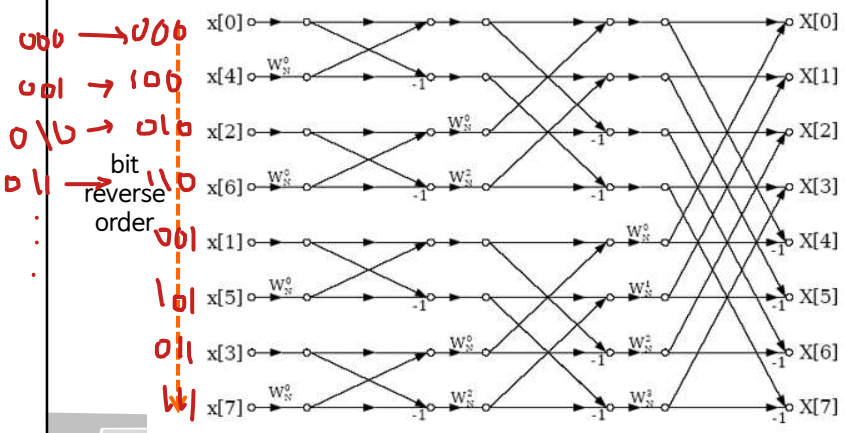
$$X[k] = \underbrace{\sum_{r=0}^{(\frac{N}{2})-1} x[2r]W_{N/2}^{rk}}_{\text{N/2 point DFT with even samples}} + W_N^k \underbrace{\sum_{r=0}^{(\frac{N}{2})-1} x[2r+1]W_{N/2}^{rk}}_{\text{N/2 point DFT with odd samples}}$$

$$= E[k] + W_N^k O[k]$$



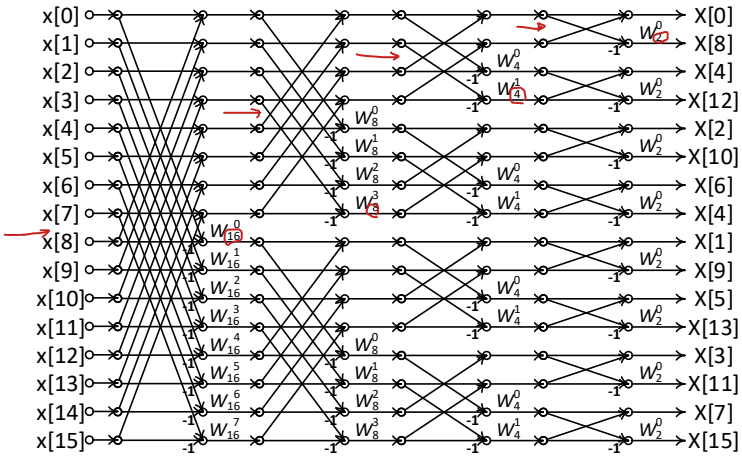
Fast Fourier Transform

- Fast Fourier transform – Decimation in Time
  - to recursively break down DFTs → N/2 $\log_2$ N complex multiplications

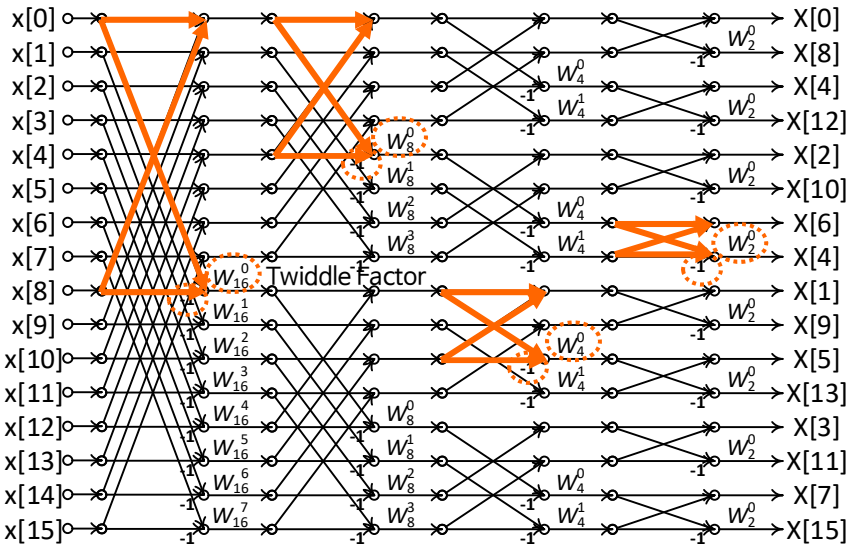


FFT – Decimation in Frequency

- Fast Fourier transform – Decimation in Frequency



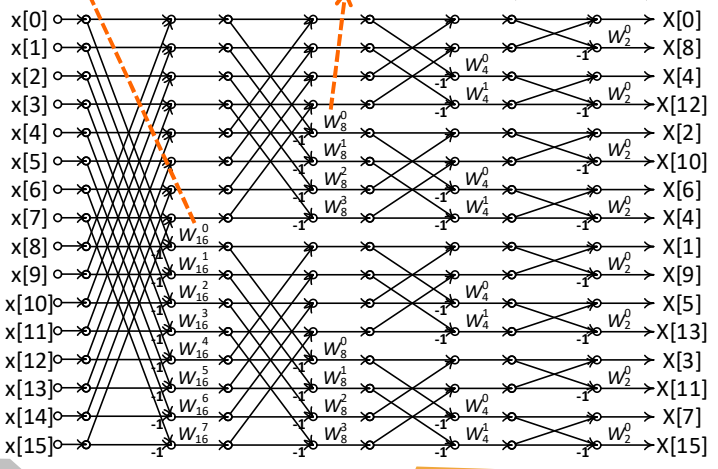
Butterfly Operation



Twiddle Factor

$$W_N^{0\sim N/2-1} = \cos\left(-2\pi\frac{0\sim N/2-1}{N}\right) + j\sin\left(-2\pi\frac{0\sim N/2-1}{N}\right)$$

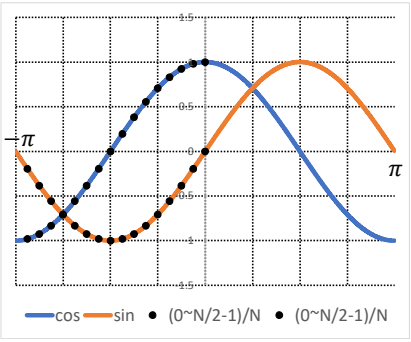
$$W_{N/2}^{0\sim N/4-1} = \cos\left(-2\pi\frac{0\sim N/4-1}{N/2}\right) + j\sin\sin\left(-2\pi\frac{0\sim N/4-1}{N/2}\right)$$

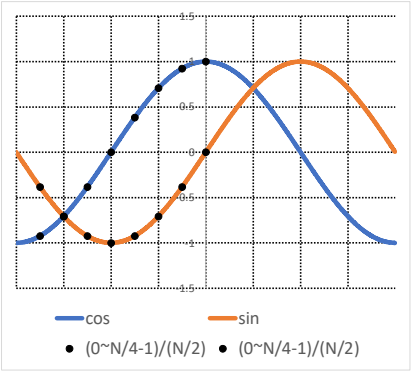


Twiddle Factor

$$W_N^{0\sim N/2-1} = \cos\left(-2\pi\frac{0\sim N/2-1}{N}\right) + j\sin\left(-2\pi\frac{0\sim N/2-1}{N}\right)$$

$$W_{N/2}^{0\sim N/4-1} = \cos\left(-2\pi\frac{0\sim N/4-1}{N/2}\right) + j\sin\sin\left(-2\pi\frac{0\sim N/4-1}{N/2}\right)$$

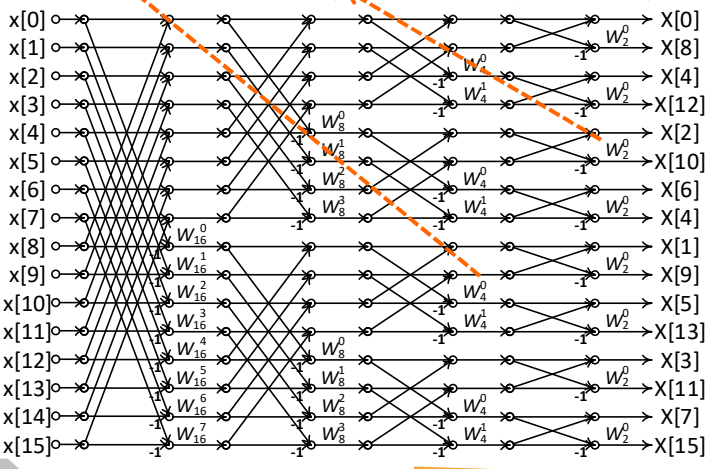




Twiddle Factor

$$W_{N/4}^{0\sim N/8-1} = \cos\left(-2\pi\frac{0\sim N/8-1}{N/4}\right) + j\sin\sin\left(-2\pi\frac{0\sim N/8-1}{N/4}\right)$$

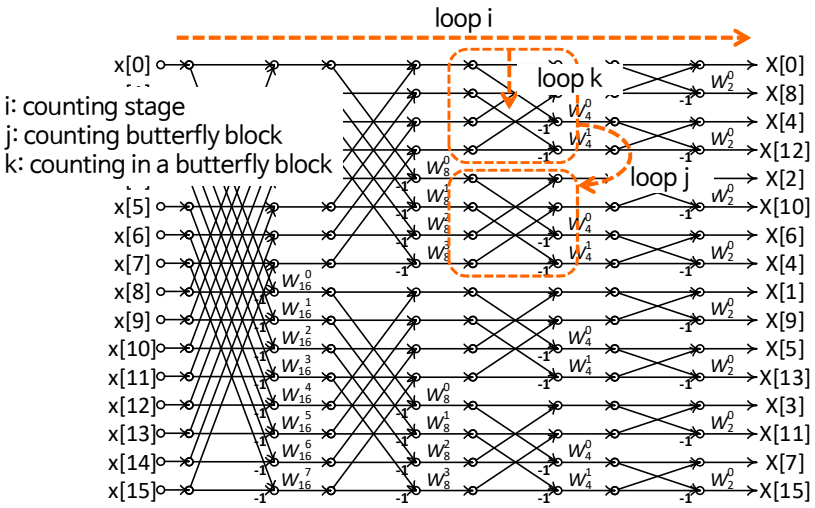
$$W_{N/8}^{0\sim N/16-1} = \cos\left(-2\pi\frac{0\sim N/16-1}{N/8}\right) + j\sin\sin\left(-2\pi\frac{0\sim N/8-1}{N/4}\right)$$



Outline

- Fast Fourier Transform (FFT)
- C Implementation
- Fixed Point Implementation
- Verilog Implementation – Memory-based
- Verilog Implementation – Pipelined

FFT – Decimation in Frequency

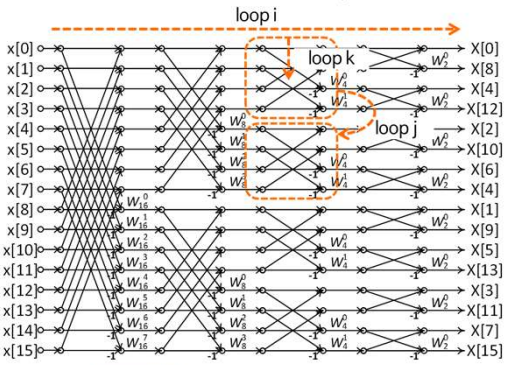


C Implementation – 1024 Point

```
for(i=0,p=1024;i<10;i++,p/=2) {
    for(j=0;j<1024/p;j++) {
        for(k=0;k<p/2;k++) {
            t_complex   bf0, bf1;
            complex_add(&bf0, out[j*p+k], out[j*p+k+p/2]);
            complex_sub(&bf1, out[j*p+k], out[j*p+k+p/2]);
            twiddle_mul(&bf1, k, p);
            out[j*p+k] = bf0;
            out[j*p+k+p/2] = bf1;
        }
    }
}
```

butterfly operation

i: counting stage
(p: butterfly block size)
j: counting butterfly block
k: counting in a butterfly block



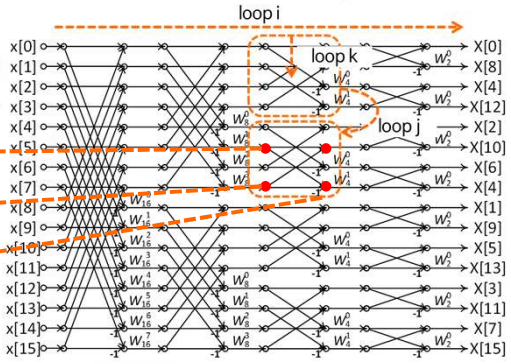
Indexing

```
for(i=0,p=1024;i<10;i++,p/=2) {
    for(j=0;j<1024/p;j++) {
        for(k=0;k<p/2;k++) {
            t_complex   bf0, bf1;
            complex_add(&bf0, out[j*p+k], out[j*p+k+p/2]);
            complex_sub(&bf1, out[j*p+k], out[j*p+k+p/2]);
            twiddle_mul(&bf1, k, p);
            out[j*p+k] = bf0;
            out[j*p+k+p/2] = bf1;
        }
    }
}
```

Element k in butterfly block j
(block size is p)

Half block size apart

Store at the read points



Complex Operation

```
typedef struct {
    float   r, i;
} t_complex;

void complex_add(t_complex *dst, t_complex src0, t_complex src1) {
    dst->r = src0.r + src1.r;
    dst->i = src0.i + src1.i;
}

void complex_sub(t_complex *dst, t_complex src0, t_complex src1) {
    dst->r = src0.r - src1.r;
    dst->i = src0.i - src1.i;
}

void complex_mul(t_complex *dst, t_complex src0, t_complex src1) {
    dst->r = src0.r * src1.r - src0.i * src1.i;
    dst->i = src0.r * src1.i + src0.i * src1.r;
}
```

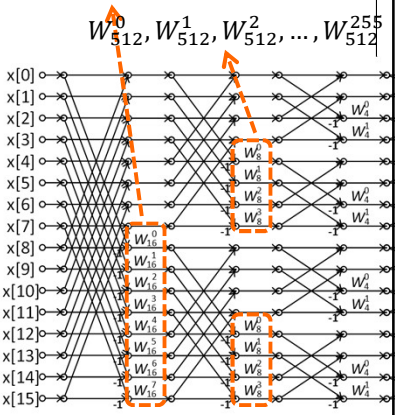

Twiddle Factor Generation/Multiplication

```
void twiddle_mul(t_complex *dst, int k, int N) {
    t_complex   twiddle_factor;
    const float pi = acos(-1.0);

    twiddle_factor.r = cos(-2*pi*k/N);
    twiddle_factor.i = sin(-2*pi*k/N);
    complex_mul(dst, *dst, twiddle_factor);
}
...
    for(i=0,p=1024;i<10;i++,p/=2) {
        for(j=0;j<1024/p;j++) {
            for(k=0;k<p/2;k++) {
                t_complex   bf0, bf1;
                complex_add(&bf0, out[j*p+k],
                complex_sub(&bf1, out[j*p+k],
                twiddle_mul(&bf1,  ,  );
                out[j*p+k] = bf0;
                out[j*p+k+p/2] = bf1;
            }
        }
    }
...
```

$$W_N^k = e^{-j2\pi k/N}$$

$$W_{1024}^0, W_{1024}^1, W_{1024}^2, \dots, W_{1024}^{511}$$

$$W_{512}^0, W_{512}^1, W_{512}^2, \dots, W_{512}^{255}$$



FFT Function

```
void fft(float in[1024], t_complex out[1024]) {
    int i, j, k, p;

    for(i=0;i<1024;i++) {
        out[i].r = in[i];
        out[i].i = 0;
    }
    for(i=0,p=1024;i<10;i++,p/=2) {
        for(j=0;j<1024/p;j++) {
            for(k=0;k<p/2;k++) {
                t_complex   bf0, bf1;
                complex_add(&bf0, out[j*p+k], out[j*p+k+p/2]);
                complex_sub(&bf1, out[j*p+k], out[j*p+k+p/2]);
                twiddle_mul(&bf1, k, p);
                out[j*p+k] = bf0;
                out[j*p+k+p/2] = bf1;
            }
        }
    }
}
```

store the input data in out array

get two data

store the results at the positions

Testbench – sin Input

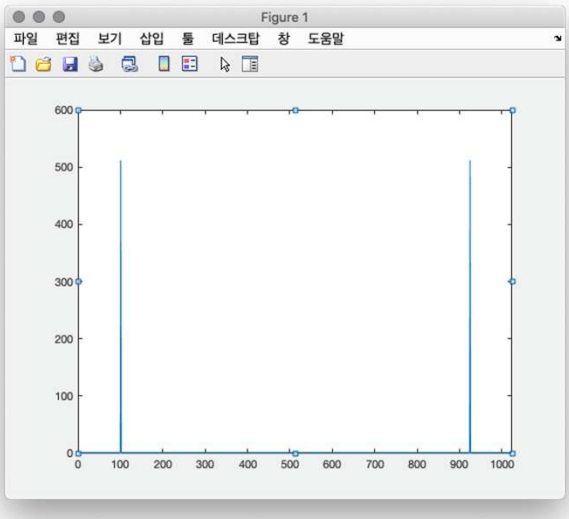
```
int bit_reverse(int in) {
    int i, out = 0;
    for(i=0;i<10;i++) {
        out <<= 1; out |= in & 0x01; in >>= 1;
    }
    return out;
}
int main(void) {
    float       fft_in[1024];
    t_complex   fft_out[1024];
    int     i;
    const float pi = acos(-1.0);
    for(i=0;i<1024;i++) {
        fft_in[i] = sin(2*pi*i*100/1024);
    }
    fft(fft_in, fft_out);
    for(i=0;i<1024;i++) {
       printf("%f %f\n",fft_out[bit_reverse(i)].r,fft_out[bit_reverse(i)].i);
    }
}
```

Frequency 100
(100 repetition in 1024 points)
→ corresponding to X[100]

Testbench – sin Input

```
...
float complex_mag(t_complex in) {
    return sqrt(in.r*in.r + in.i*in.i);
}
...
int main(void) {
    float       fft_in[1024];
    t_complex   fft_out[1024];
    int     i;
    const float pi = acos(-1.0);
    for(i=0;i<1024;i++) {
        fft_in[i] = sin(2*pi*i*100/1024);
    }
    fft(fft_in, fft_out);
    for(i=0;i<1024;i++) {
        printf("%f\n", complex_mag(fft_out[bit_reverse(i)]));
    }
}
```

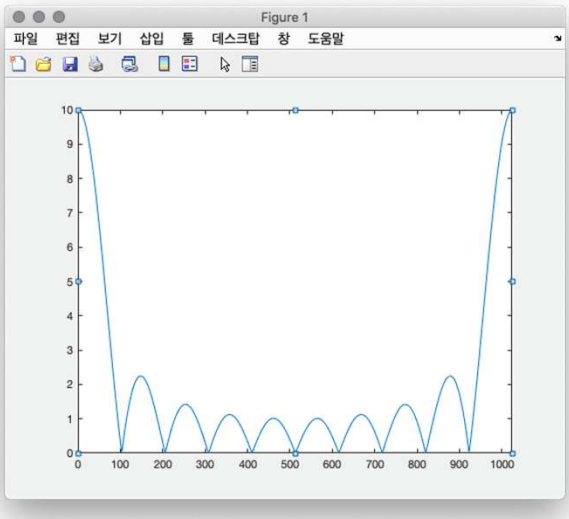

FFT 결과



Testbench – Rectangular

```
...
float complex_mag(t_complex in) {
    return sqrt(in.r*in.r + in.i*in.i);
}
...
int main(void) {
    float       fft_in[1024];
    t_complex   fft_out[1024];
    int     i;
    const float pi = acos(-1.0);
    for(i=0;i<1024;i++) {
        fft_in[i] = (i<5) ? 1 : (i>=1019) ? 1 : 0;
    }
    fft(fft_in, fft_out);
    for(i=0;i<1024;i++) {
        printf("%f\n", complex_mag(fft_out[bit_reverse(i)]));
    }
}
```


FFT 결과



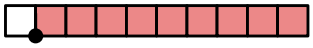
Outline

- Fast Fourier Transform (FFT)
- C Implementation
- Fixed Point Implementation
- Verilog Implementation – Memory-based
- Verilog Implementation – Pipelined

Bit Length

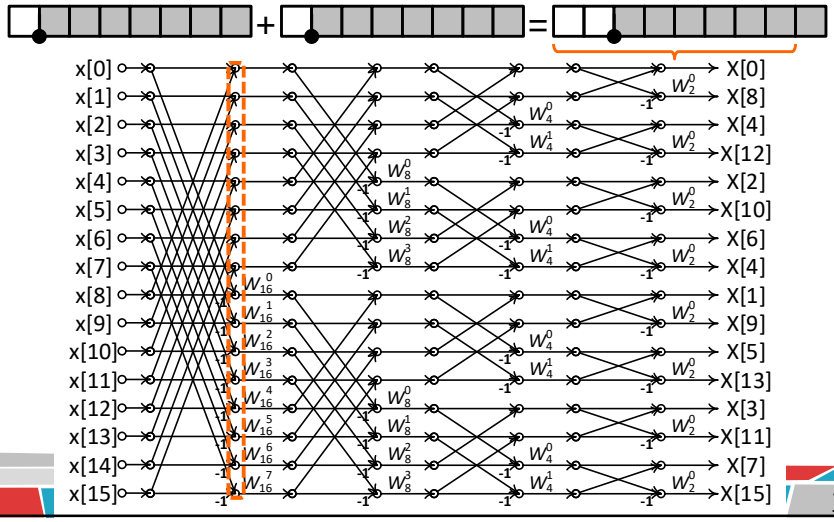
- Input data: 16 bit with 15 bit fraction
  - range: -1 ~ 1

- Twiddle factor: 10 bit with 9 bit fraction
  - range: -1 ~ 1



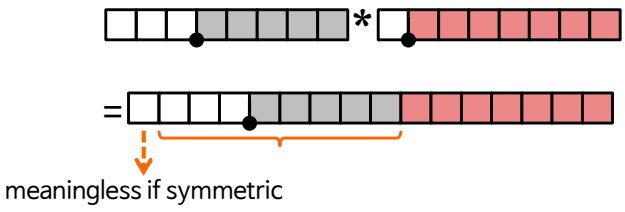
Bit Length

- After add or subtract, the range may be doubled.
  - To keep 16 bit length, reduce the fractional part.



Bit Length

- After multiplication, return to the original length
  - reduce the fractional part



Complex Operation

```
typedef struct {
    int     r, i;
} t_complex;

void complex_add(t_complex *dst, t_complex src0, t_complex src1) {
    dst->r = (src0.r + src1.r) >> 1;
    dst->i = (src0.i + src1.i) >> 1;
}

void complex_sub(t_complex *dst, t_complex src0, t_complex src1) {
    dst->r = (src0.r - src1.r) >> 1;
    dst->i = (src0.i - src1.i) >> 1;
}

void complex_mul(t_complex *dst, t_complex src0, t_complex src1) {
    dst->r = (src0.r * src1.r - src0.i * src1.i) >> 9;
    dst->i = (src0.r * src1.i + src0.i * src1.r) >> 9;
}
```

truncation for simplicity

Twiddle Factor Generation

```
void twiddle_mul(t_complex *dst, int k, int N) {
    t_complex   twiddle_factor;
    const float pi = acos(-1.0);

    twiddle_factor.r = floor(cos(-2*pi*k/N)*511 + 0.5);
    twiddle_factor.i = floor(sin(-2*pi*k/N)*511 + 0.5);
    complex_mul(dst, *dst, twiddle_factor);
}
```

- why *511
  - if *512, the resulting value will be in [-512, 512], which cannot be represented with 10 bits.

FFT Function

```
void fft(int in[1024], t_complex out[1024]) {
    int i, j, k, p;

    for(i=0;i<1024;i++) {
        out[i].r = in[i];
        out[i].i = 0;
    }
    for(i=0,p=1024;i<10;i++,p/=2) {
        for(j=0;j<1024/p;j++) {
            for(k=0;k<p/2;k++) {
                t_complex   bf0, bf1;
                complex_add(&bf0, out[j*p+k], out[j*p+k+p/2]);
                complex_sub(&bf1, out[j*p+k], out[j*p+k+p/2]);
                twiddle_mul(&bf1, k, p);
                out[j*p+k] = bf0;
                out[j*p+k+p/2] = bf1;
            }
        }
    }
}
```


Testbench – sin Input

```
...
float complex_mag(t_complex in) {
    return sqrt(in.r*in.r/32./32. + in.i*in.i/32./32.);
}
...
int main(void) {
    int     fft_in[1024];
    t_complex   fft_out[1024];
    int     i;
    const float pi = acos(-1.0);

    for(i=0;i<1024;i++) {
        fft_in[i] = floor(sin(2*pi*i*100/1024) * 32767 + 0.5);
    }
    fft(fft_in, fft_out);
    for(i=0;i<1024;i++) {
        printf("%f\n", complex_mag(fft_out[bit_reverse(i)]));
    }

}
```

why?

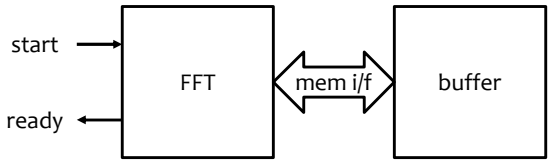
why not 32768?

Outline

- Fast Fourier Transform (FFT)
- C Implementation
- Fixed Point Implementation
- Verilog Implementation – Memory-based
  - I/F & Counting
  - Datapath
  - Twiddle Factor
  - Testbench & Further Works
- Verilog Implementation – Pipelined

Architecture & Interface

- 1024-point FFT
- start & ready
  - On idle (ready = 1), start can be asserted.
    - During the process, ready = 0.
  - If the FFT is finished, ready is asserted.
- To use an external buffer



  - data width: 32 bits (16-bit real and imaginary)

Module Input/Output

```
module fft (
              input     clk
            , input     n_reset
            , input     start
            , output    ready

            , output            cs
            , output            we
            , output    [9:0]   addr
            , output    [31:0]  w_data
            , input     [31:0]  r_data
);

reg             on_proc;
```

------> 1 during the process

for Loops → Counting

```
    for(i=0,p=1024;i<10;i++,p/=2) {
        for(j=0;j<1024/p;j++) {
            for(k=0;k<p/2;k++) {
                t_complex   bf0, bf1;
                complex_add(&bf0, out[j*p+k], out[j*p+k+p/2]);
                complex_sub(&bf1, out[j*p+k], out[j*p+k+p/2]);
                twiddle_mul(&bf1, k, p);
                out[j*p+k] = bf0;
                out[j*p+k+p/2] = bf1;
            }
        }
    }
```

cnt_p: 1024, 512, 256, ..., 2 → 512, 256, 128, ..., 1

cnt_pi: inverse of cnt_p → 1, 2, 4, ..., 512

cnt_j: 0, 1, 2, 3, ... ,1024/(cnt_p*2)−1 → 0, 1, 2, 3, ..., cnt_pi−1

cnt_k: 0, 1, 2, 3, ... , cnt_p−1

cnt_o: ? ? cycles for read, add/sub, mul, write

for Loops → Counting

```
Let's describe the following signals
    cnt_o, cnt_k, cnt_j, cnt_p
```

cnt_o: 0, 1, 2, 3, ..., 6
cnt_k: 0, 1, 2, 3, ... , cnt_p-1
cnt_j: 0, 1, 2, 3, ..., cnt_pi-1
cnt_p: 512, 256, 128, ..., 1

Counting Initialization & Ready

```
assign ready = ~on_proc;

always@(posedge clk or negedge n_reset) begin
    if(n_reset == 1'b0) begin
        on_proc <= 1'b0;
        cnt_p <= 10'h200;
        cnt_j <= 'b0;
        cnt_k <= 'b0;
        cnt_o <= 'b0;
    end else begin
        if((on_proc == 1'b0) && (start == 1'b1)) begin
            on_proc <= 1'b1;
            cnt_p <= 10'h200;
            cnt_j <= 'b0;
            cnt_k <= 'b0;
            cnt_o <= 'b0;
        end
        ...
    end
end
```

cnt_o: 0, 1, 2, 3, ..., ??
cnt_k: 0, 1, 2, 3, ... , cnt_p-1
cnt_j: 0, 1, 2, 3, ..., cnt_pi-1
cnt_p: 512, 256, 128, ..., 1

cnt_pi

```
wire        last_o = (cnt_o == 6);
wire        last_k = (cnt_k == cnt_p-1);
wire        last_j = (cnt_j == cnt_pi-1);
wire        last_p = (cnt_p == 10'h001);
always@(posedge clk or negedge n_reset) begin
        ...
                    if(last_j == 1'b1) begin
                        cnt_p <= cnt_p >> 1;
                        if(last_p == 1'b1) begin
                            on_proc <= 1'b0;
                        end
                    end
        ...
end
genvar i;
for(i=0;i<10;i++) begin
    assign cnt_pi[i] = cnt_p[9-i];
end
```

cnt_o: 0, 1, 2, 3, ..., ??

cnt_k: 0, 1, 2, 3, ... , cnt_p−1

cnt_j: 0, 1, 2, 3, ..., cnt_pi−1

cnt_p: 512, 256, 128, ..., 1

cnt_pi: inverse of cnt_p → 1, 2, 4, ..., 512

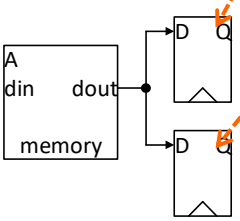
Outline

- Fast Fourier Transform (FFT)
- C Implementation
- Fixed Point Implementation
- Verilog Implementation – Memory-based
  - I/F & Counting
  - Datapath
  - Twiddle Factor
  - Testbench & Further Works
- Verilog Implementation – Pipelined

Operation

```
t_complex   bf0, bf1;
complex_add(&bf0, out[j*p+k], out[j*p+k+p/2]);
complex_sub(&bf1, out[j*p+k], out[j*p+k+p/2]);
twiddle_mul(&bf1, k, p);
out[j*p+k] = bf0;
out[j*p+k+p/2] = bf1;
```

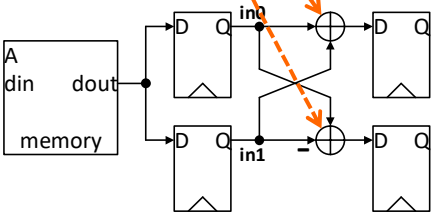
Read two points from memory one by one (two sets of real and imaginary)



Operation

```
t_complex   bf0, bf1;
complex_add(&bf0, out[j*p+k], out[j*p+k+p/2]);
complex_sub(&bf1, out[j*p+k], out[j*p+k+p/2]);
twiddle_mul(&bf1, k, p);
out[j*p+k] = bf0;
out[j*p+k+p/2] = bf1;
```

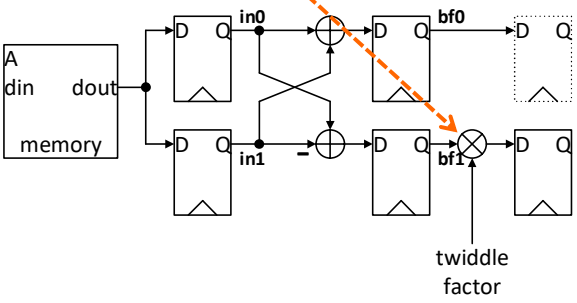
add and sub



Operation

```
t_complex   bf0, bf1;
complex_add(&bf0, out[j*p+k], out[j*p+k+p/2]);
complex_sub(&bf1, out[j*p+k], out[j*p+k+p/2]);
twiddle_mul(&bf1, k, p);
out[j*p+k] = bf0;
out[j*p+k+p/2] = bf1;
```

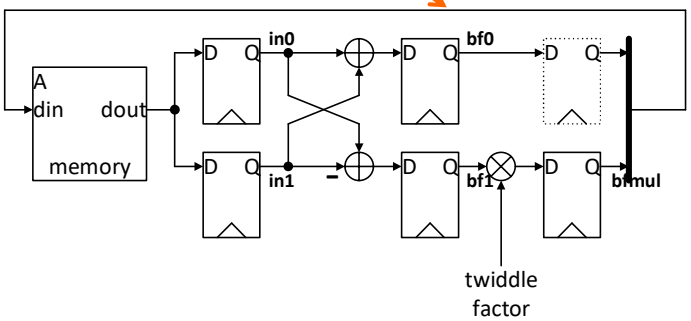
twiddle factor multiplication
(How to generate cos and sin?)



Operation

```
t_complex   bf0, bf1;
complex_add(&bf0, out[j*p+k], out[j*p+k+p/2]);
complex_sub(&bf1, out[j*p+k], out[j*p+k+p/2]);
twiddle_mul(&bf1, k, p);
out[j*p+k] = bf0;
out[j*p+k+p/2] = bf1;
```

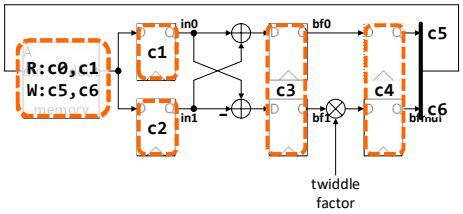
Write two points to memory
one by one



Operation Cycles

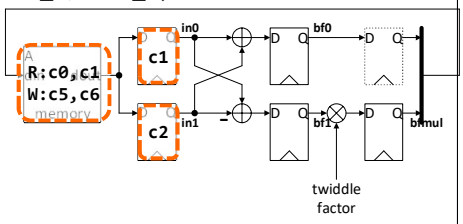
```
t_complex   bf0, bf1;
complex_add(&bf0, out[j*p+k], out[j*p+k+p/2]);
complex_sub(&bf1, out[j*p+k], out[j*p+k+p/2]);
twiddle_mul(&bf1, k, p);
out[j*p+k] = bf0;
out[j*p+k+p/2] = bf1;
```

```
Cycle 0: Read assert for Mem[j*p+k]
Cycle 1: Mem data → in0, Read assert for Mem[j*p+k+p/2]
Cycle 2: Mem data → in1
Cycle 3: in0 + in1 → bf0, in0 – in1 → bf1
Cycle 4: bf1 * twid → bfmul
Cycle 5: Write assert for Mem[j*p+k] with bf0
Cycle 6: Write assert for Mem[j*p+k+p/2] with bfmul
```



Datapath

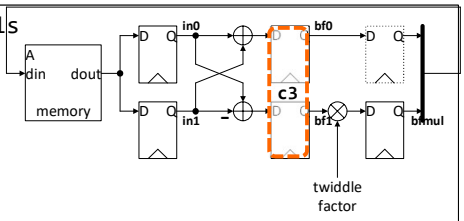
Let's describe the following signals
cs, we, in0(in0_r, in0_i), in1(in1_r, in1_i)




```
Cycle 0: Read assert for Mem[j*p+k]
Cycle 1: Mem data → in0, Read assert for Mem[j*p+k+p/2]
Cycle 2: Mem data → in1
Cycle 3: in0 + in1 → bf0, in0 – in1 → bf1
Cycle 4: bf1 * twid → bfmul
Cycle 5: Write assert for Mem[j*p+k] with bf0
Cycle 6: Write assert for Mem[j*p+k+p/2] with bfmul
```

Datapath

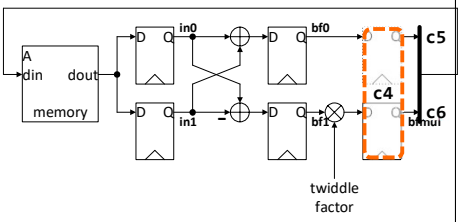
Let's describe the following signals
bf0(bf0_r, bf0_i)
bf1(bf1_r, bf1_i)



Cycle 0: Read assert for Mem[j*p+k]
Cycle 1: Mem data → in0, Read assert for Mem[j*p+k+p/2]
Cycle 2: Mem data → in1
Cycle 3: **in0 + in1 → bf0, in0 – in1 → bf1**
Cycle 4: bf1 * twid → bfmul
Cycle 5: Write assert for Mem[j*p+k] with bf0
Cycle 6: Write assert for Mem[j*p+k+p/2] with bfmul

Datapath

Let's describe the following signals
bfmul(bfmul_r, bfmul_i), w_data



Cycle 3: **in0 + in1 → bf0, in0 – in1 → bf1**
Cycle 4: **bf1 * twid → bfmul**
Cycle 5: Write assert for Mem[j*p+k] with **bf0**
Cycle 6: Write assert for Mem[j*p+k+p/2] with **bfmul**

Address

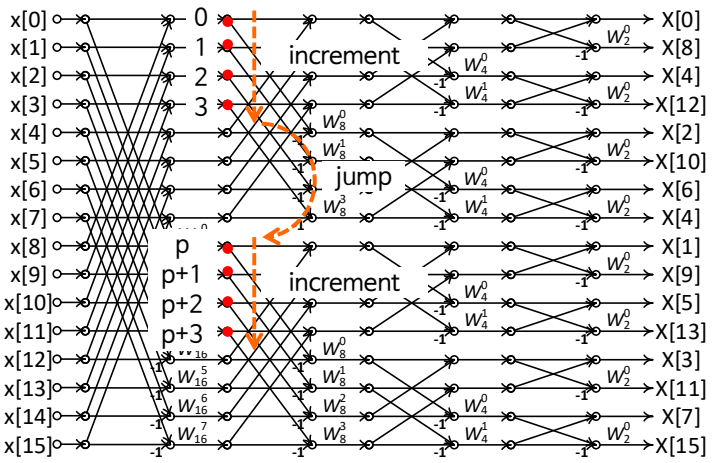
```
Cycle 0: Read assert for Mem[j*p+k]
Cycle 1: Mem data → in0, Read assert for Mem[j*p+k+p/2]
Cycle 2: Mem data → in1
Cycle 3: in0 + in1 → bf0, in0 – in1 → bf1
Cycle 4: bf1 * twid → bfmul
Cycle 5: Write assert for Mem[j*p+k] with bf0
Cycle 6: Write assert for Mem[j*p+k+p/2] with bfmul
```

```
wire    [9:0]   addr0 = cnt_j*cnt_p*2 + cnt_k;
wire    [9:0]   addr1 = cnt_j*cnt_p*2 + cnt_k + cnt_p;

assign addr = (cnt_o == 0) || (cnt_o == 5) ? addr0 : addr1;
```

- Multiplication does not seem so good.
  - We can use a shifter because cnt_p is one hot.
- In this lecture, let's use the incrementing property.

Address

```
Cycle 0: Read assert for Mem[j*p+k]
```



Address

```
Cycle 0: Read assert for Mem[j*p+k]
Cycle 1: Mem data → in0, Read assert for Mem[j*p+k+p/2]
Cycle 2: Mem data → in1
Cycle 3: in0 + in1 → bf0, in0 - in1 → bf1
Cycle 4: bf1 * twid → bfmul
Cycle 5: Write assert for Mem[j*p+k] with bf0
Cycle 6: Write assert for Mem[j*p+k+p/2] with bfmul
```

j=0 j=1 j=2

Mem[j*p+k]: 0, 1, 2, ..., p/2−1, p, p+1, p+2, ..., p+p/2−1, 2p, 2p+1, 2p+2, ...

Mem[j*p+k+p/2]: +p/2

addr0: 0, 1, 2, ..., cnt_p−1, cnt_p*2, +1, +2, ..., +cnt_p−1, cnt_p*4

+1 +1 +1 cnt_p+1

when? at the end of the butterfly block = last of cnt_k

addr1: +cnt_p

Address

```
Let's describe the following signals
    addr → addr0, addr1
```

addr0: 0, 1, 2, ..., cnt_p−1, cnt_p*2, +1, +2, ..., +cnt_p−1, cnt_p*4

+1 +1 +1 cnt_p+1

addr1: +cnt_p

Outline

- Fast Fourier Transform (FFT)
- C Implementation
- Fixed Point Implementation
- Verilog Implementation – Memory-based
  - I/F & Counting
  - Datapath
  - Twiddle Factor
  - Testbench & Further Works
- Verilog Implementation – Pipelined

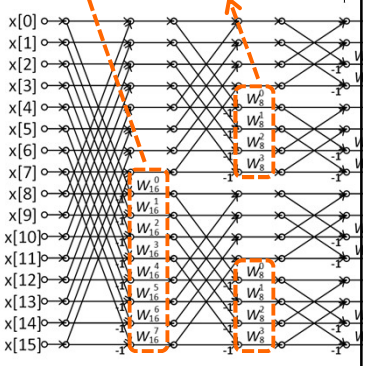
Twiddle Factor Generation

```
void twiddle_mul(t_complex *dst, int k, int N) {
    t_complex   twiddle_factor;
    const float pi = acos(-1.0);

    twiddle_factor.r = cos(-2*pi*k/N);
    twiddle_factor.i = sin(-2*pi*k/N);
    complex_mul(dst, *dst, twiddle_factor);
}
```

$$W_N^k = e^{-j2\pi k/N}$$

$$W_{1024}^0, W_{1024}^1, W_{1024}^2, \dots, W_{1024}^{511}$$

$$W_{512}^0, W_{512}^1, W_{512}^2, \dots, W_{512}^{255}$$



- cos and sin are required.

$$W_N^k = e^{-j2\pi k/N}$$
$$= \cos -2\pi k/N + j \sin -2\pi k/N$$

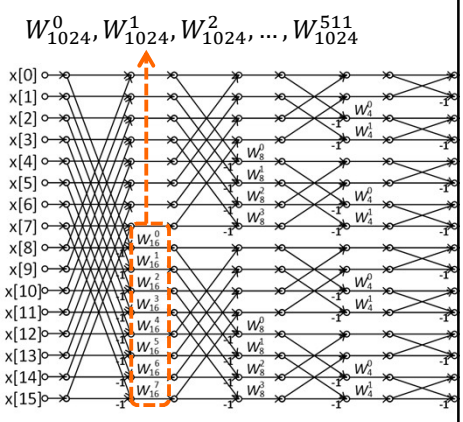
Non-linear Function

- CORDIC (**CO**ordinate **R**otation **DI**gital **C**omputer)
  - a simple and efficient algorithm to calculate trigonometric functions, hyperbolic functions, square roots, multiplications, divisions, and exponentials and logarithms
- Look-Up Table
  - How to make table?

cos, sin Look-Up Table

- Stage 0 $\cos -2\pi k/N + j \sin -2\pi k/N$
  - 0/1024, 1/1024, 2/2024, …, 511/1024

k	values
0	$\cos 2\pi(-0/1024), \sin 2\pi(-0/1024)$
1	$\cos 2\pi(-1/1024), \sin 2\pi(-1/1024)$
2	$\cos 2\pi(-2/1024), \sin 2\pi(-2/1024)$
3	$\cos 2\pi(-3/1024), \sin 2\pi(-3/1024)$
4	$\cos 2\pi(-4/1024), \sin 2\pi(-4/1024)$
5	$\cos 2\pi(-5/1024), \sin 2\pi(-5/1024)$
6	$\cos 2\pi(-6/1024), \sin 2\pi(-6/1024)$
7	$\cos 2\pi(-7/1024), \sin 2\pi(-7/1024)$
…	
511	$\cos 2\pi(-511/1024), \sin 2\pi(-511/1024)$



Look-Up Table Description

- Look-up table implementation
  - ROM
  - logic

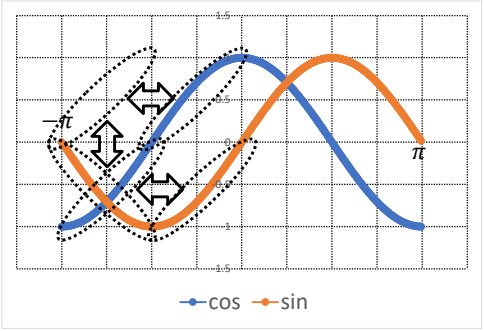
```
always@(*) begin
    case(cnt_k)
        0: begin cos = 511; sin = -0; end
        1: begin cos = 511; sin = -3; end
        2: begin cos = 511; sin = -6; end
        ...
        511: begin cos = -511; sin = -3; end
    endcase
end
```

  - Twiddle factor: 10 bit with 9 bit fraction

```
    twiddle_factor.r = floor(cos(-2*pi*k/N)*511 + 0.5);
    twiddle_factor.i = floor(sin(-2*pi*k/N)*511 + 0.5);
```


Look-Up Table Reduction

- cos and sin are symmetric functions.
  - k: 0~511 is 0~−π.



$$if\ \theta \le -\frac{\pi}{2},\begin{cases}\cos\theta = \sin(\theta+\pi/2)\\ \sin\theta = -\cos(\theta+\pi/2)\end{cases}$$

$$-\pi < \theta \le -\frac{\pi}{2} \rightarrow -\frac{\pi}{2} < \theta + \frac{\pi}{2} \le 0$$

Look-Up Table Reduction

- LUT indexing
  - k: 0~511 is 0~–π.

$$if\ \theta \le -\frac{\pi}{2},\begin{cases}\cos\theta = \sin(\theta+\pi/2)\\ \sin\theta = -\cos(\theta+\pi/2)\end{cases}$$

$$-\pi < \theta \le -\frac{\pi}{2} \rightarrow -\frac{\pi}{2} < \theta + \frac{\pi}{2} \le 0$$

$$\theta = -\frac{2\pi k}{1024}$$

$$\theta + \frac{\pi}{2} = -\frac{2\pi k}{1024} + \frac{\pi}{2} = -\frac{2\pi(k-256)}{1024}$$

rad	0	$-\frac{\pi}{2}$	$-\pi$
k	0, 1, 2, 3, …, 255,	256, 257, 258, 259,…,	511
LUT index	0, 1, 2, 3, …, 255,	0, 1, 2, 3,…,	255

Look-Up Table Description

$$if\ \theta \le -\frac{\pi}{2},\begin{cases}\cos\theta = \sin(\theta+\pi/2)\\ \sin\theta = -\cos(\theta+\pi/2)\end{cases}$$

k	0, 1, 2, 3, …, 255,	256, 257, 258, …,	511
LUT index	0, 1, 2, 3, …, 255,	0, 1, 2, …,	255

```
assign cos =
assign sin =

always@(*) begin
    case(          )
        0: begin lut_cos = 511; lut_sin = -0; end
        1: begin lut_cos = 511; lut_sin = -3; end
        2: begin lut_cos = 511; lut_sin = -6; end
        ...
        255: begin lut_cos = 3; lut_sin = -511; end
    endcase
end
```

Let's complete the description.

Twiddle Factor Generation

```
reg         [19:0]  twid_lut;
wire signed [9:0]   cos =
wire signed [9:0]   sin =
reg  signed [9:0]   twid_r, twid_i;

always@(posedge clk or negedge n_reset) begin
    if(n_reset == 1'b0) begin
        twid_r <= 'b0; twid_i <= 'b0;
    end else begin
        if(          ) begin
            twid_r <= cos; twid_i <= sin;
        end
    end
end
always@(*) begin
    case(          )
        0: twid_lut = {-10'd0,10'd511};
        1: twid_lut = {-10'd3,10'd511};
        2: twid_lut = {-10'd6,10'd511};
            ...
        254: twid_lut = {-10'd511,10'd6};
        255: twid_lut = {-10'd511,10'd3};
    endcase
end
```

When?

How to write this code?

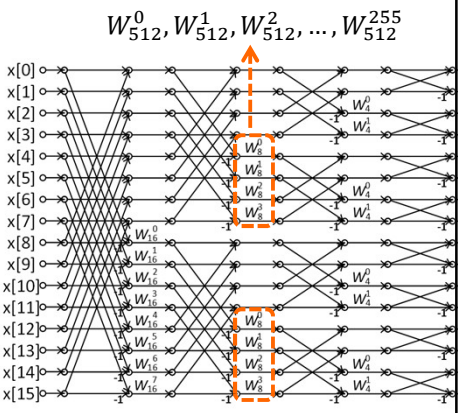
LUT Contents Output

```
#include <stdio.h>
#include <math.h>

int main(void) {
    const float pi = acos(-1.0);
    int k, c, s;
    for(k=0;k<256;k++) {
        c = floor(cos(-2*pi*k/1024)*511 + 0.5);
        s = floor(sin(-2*pi*k/1024)*511 + 0.5);
        printf("\t\t%d: twid_lut = {-10'd%d,10'd%d};\n", k, -s, c);
    }
}
```

cos, sin Look-Up Table

- Stage 1 $\cos -2\pi k/N + j \sin -2\pi k/N$
  - 0/512, 1/512, 2/512, …, 255/512

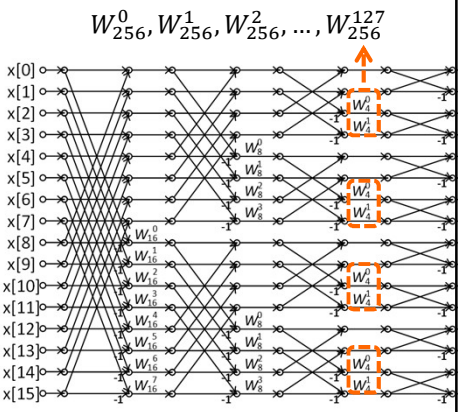
k	values
0	$\cos 2\pi(-0/512), \sin 2\pi(-0/512)$
1	$\cos 2\pi(-1/512), \sin 2\pi(-1/512)$
2	$\cos 2\pi(-2/512), \sin 2\pi(-2/512)$
3	$\cos 2\pi(-3/512), \sin 2\pi(-3/512)$
4	$\cos 2\pi(-4/512), \sin 2\pi(-4/512)$
5	$\cos 2\pi(-5/512), \sin 2\pi(-5/512)$
6	$\cos 2\pi(-6/512), \sin 2\pi(-6/512)$
7	$\cos 2\pi(-7/512), \sin 2\pi(-7/512)$
…	
255	$\cos 2\pi(-255/512), \sin 2\pi(-255/512)$



cos, sin Look-Up Table

- Stage 2 $\cos -2\pi k/N + j \sin -2\pi k/N$
  - 0/256, 1/256, 2/256, …, 127/256

k	values
0	$\cos 2\pi(-0/256), \sin 2\pi(-0/256)$
1	$\cos 2\pi(-1/256), \sin 2\pi(-1/256)$
2	$\cos 2\pi(-2/256), \sin 2\pi(-2/256)$
3	$\cos 2\pi(-3/256), \sin 2\pi(-3/256)$
4	$\cos 2\pi(-4/256), \sin 2\pi(-4/256)$
5	$\cos 2\pi(-5/256), \sin 2\pi(-5/256)$
6	$\cos 2\pi(-6/256), \sin 2\pi(-6/256)$
7	$\cos 2\pi(-7/256), \sin 2\pi(-7/256)$
…	
127	$\cos 2\pi(-127/256), \sin 2\pi(-127/256)$



Twiddle Factor Generation 2

- A new count cnt_t increases as cnt_k, but
  - stage 0: 0, 1, 2, 3, …
  - stage 1: 0, 2, 4, 6, …
  - stage 2: 0, 4, 8, 12, …

```
Let's describe cnt_t.
```


Outline

- Fast Fourier Transform (FFT)
- C Implementation
- Fixed Point Implementation
- Verilog Implementation – Memory-based
  - I/F & Counting
  - Datapath
  - Twiddle Factor
  - Testbench & Further Works
- Verilog Implementation – Pipelined

Testbench Idea

- To use DPI
  - Input stimulus is generated in C and transferred to Verilog through DPI.
  - The output of Verilog is compared with that of C.

DPI Functions

```
int         fft_in[1024];
t_complex   fft_out[1024];
void init_fft(void) {
    int     i;
    for(i=0;i<1024;i++) {
        fft_in[i] = (i<5) ? 32767 : (i>=1019) ? 32767 : 0;
    }
    fft(fft_in, fft_out);
}

unsigned get_input(void) {
    static int  rp = 0;
    return fft_in[rp++] & 0x0ffff;
}

unsigned get_output(void) {
    static int  rp = 0;
    unsigned out = ((fft_out[rp].i & 0x0ffff) << 16)
                 | (fft_out[rp].r & 0x0ffff);
    rp++;
    return out;
}
```

at initialization

input stimulus

pre-calculation of FFT

return input stimulus one by one

to pack 16-bit real and imaginary into a 32-bit vector

return output result one by one

Testbench – Initialization

```
module top_fft;

reg     clk, n_reset;
reg     start;
wire    ready;

initial begin
    $vcdplusfile("top_fft.vpd");
    $vcdpluson("top_fft", 0);
end
initial clk = 1'b0;
always #5 clk = ~clk;

import "DPI" function void init_fft();
import "DPI" function int unsigned get_input();
import "DPI" function int unsigned get_output();
```

import DPI functions

DUT Inst. & Mem. Modeling

```
wire            cs, we;
wire    [9:0]   addr;
wire    [31:0]  w_data;
reg     [31:0]  r_data;

fft i_fft (
          .clk(clk)
        , .n_reset(n_reset)
        , .start(start)
        , .ready(ready)

        , .cs(cs)
        , .we(we)
        , .addr(addr)
        , .w_data(w_data)
        , .r_data(r_data)
);
```

```
reg [31:0]  mem_data[0:1023];

always@(posedge clk) begin
    if(cs == 1'b1) begin
        if(we == 1'b1) mem_data[addr] <= w_data;
        else r_data <= mem_data[addr];
    end
end
```

Testbench – Input Stimulus

```
int     i;
reg [31:0]  c_data;
initial begin
    n_reset = 1'b1;
    start = 1'b0;
    init_fft();
    for(i=0;i<1024;i++) begin
        mem_data[i] = get_input();
    end
    #3;
    n_reset = 1'b0;
    #20;
    n_reset = 1'b1;
    @(posedge clk);
    @(posedge clk);
    start = 1'b1;
    @(posedge clk);
    start = 1'b0;
```

upload the input stimulus to memory

assert start

Testbench – Output Check

```
    @(posedge ready);
    @(posedge clk);
    for(i=0;i<1024;i++) begin
        c_data = get_output();
        if(mem_data[i] !== c_data) begin
            $display("Error: mem_data[%d] = %8X, c_data = %8X"
                    , i, mem_data[i], c_data);
        end
    end
    @(posedge clk);
    @(posedge clk);
    $finish;
end
```

when done

check output

Further Works – Cycle Reduction

```
Cycle 0: Read assert for Mem[j*p+k]
Cycle 1: Mem data → in0, Read assert for Mem[j*p+k+p/2]
Cycle 2: Mem data → in1
Cycle 3: in0 + in1 → bf0, in0 – in1 → bf1
Cycle 4: bf1 * twid → bfmul
Cycle 5: Write assert for Mem[j*p+k] with bf0
Cycle 6: Write assert for Mem[j*p+k+p/2] with bfmul
```

⇩

Let's reduce one cycle.

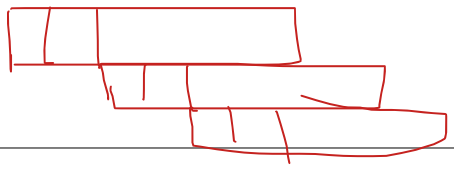
Further Works – Pipeline

- Non-pipelined

R D	D	ADD SUB	W MUL	W	R	R D	D	ADD SUB	W MUL	W	R	R D	D	ADD SUB	W MUL

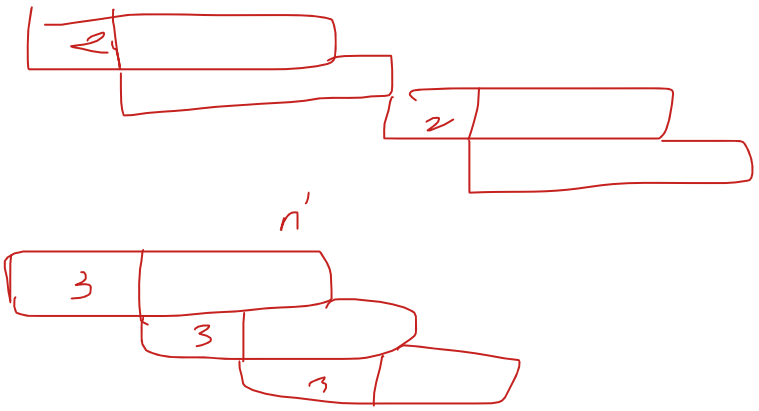
- Pipelined

Let's a draw pipelined schedule.
(A double port memory may be required.)



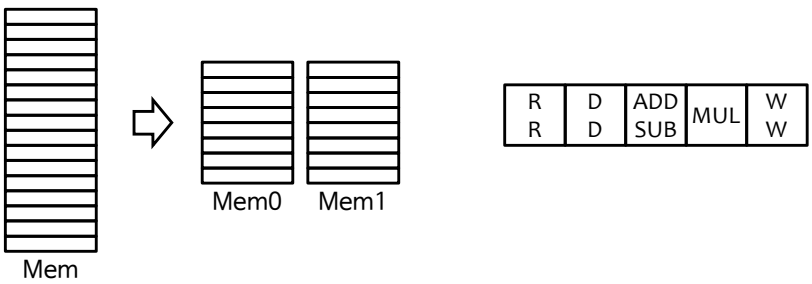
Further Works – Pipeline

Let's a draw pipelined schedule without a double port memory.



Further Works – Bank

- 4 memory accesses are required.
  - 2 reads and 2 writes
- If the memory is divided to two banks, two memory access can be processed at the same time.



Further Works – Bank

- Pipelined – 2 cycles/butterfly

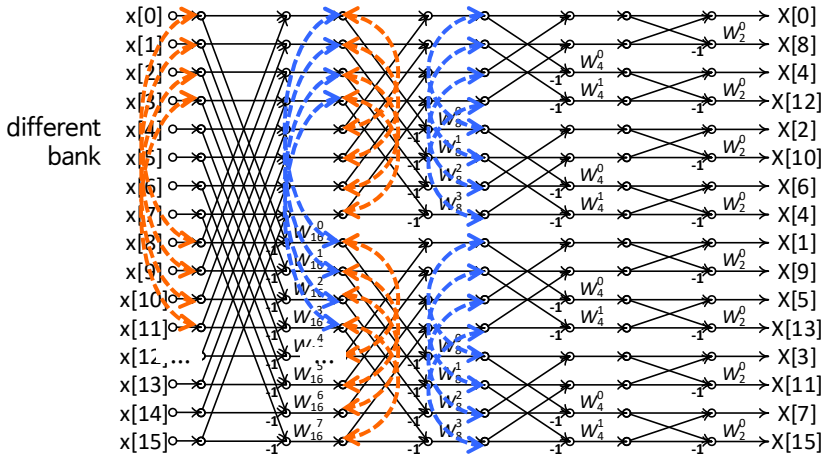
Let's a draw pipelined schedule.

Further Works – Bank

- Can 2 reads and 2 writes always be processed at the same time?
  - 2 reads at different banks
  - 2 writes at different banks

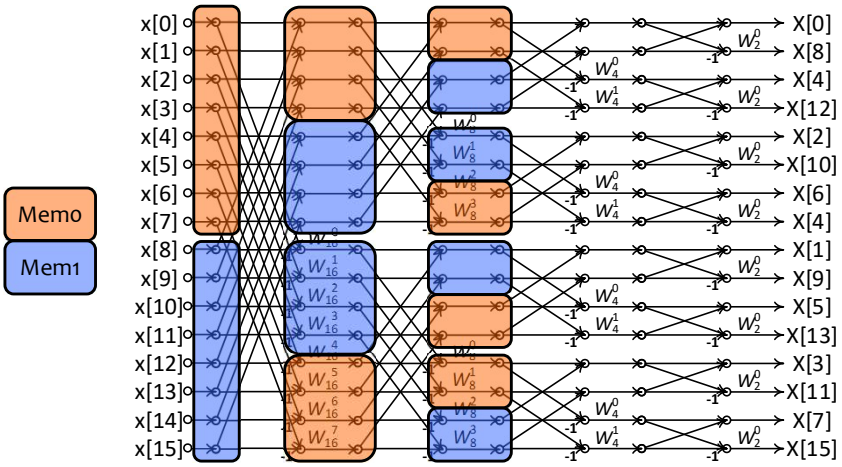
Further Works – Bank

- Bank selection



Further Works – Bank

- Bank selection



Outline

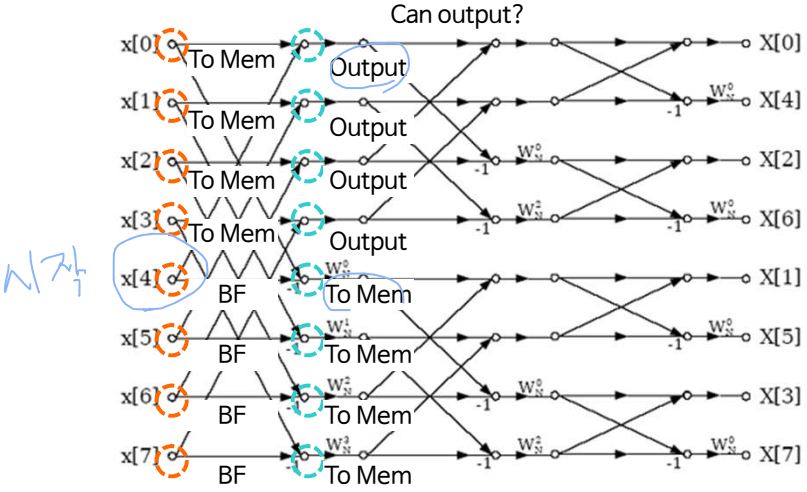
- Fast Fourier Transform (FFT)
- C Implementation
- Fixed Point Implementation
- Verilog Implementation – Memory-based
- Verilog Implementation – Pipelined
  - Idea
  - Butterfly Stage
  - Testbench & Configurable Module
  - Further Works

Architecture Comparison

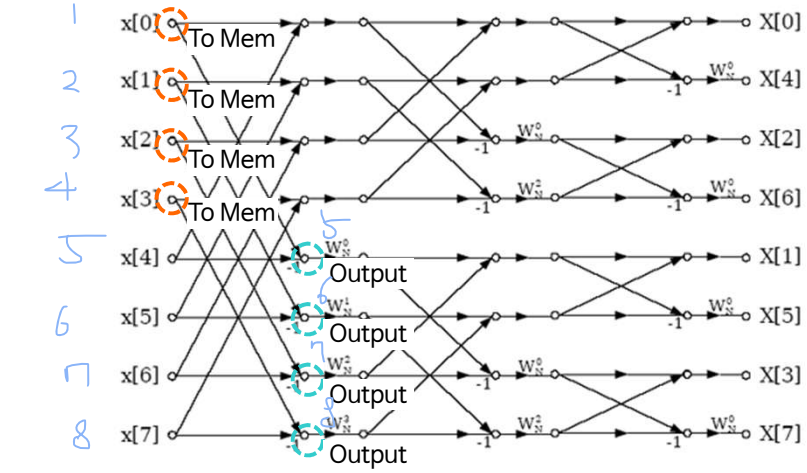
- Memory-based
  - A whole input data set is stored in a memory.
  - The FFT module processes data stage by stage, accessing the memory.
  - Double buffering may be required.
- Pipelined
  - Stream-input, stream-output
  - Pipeline between stages
    - Each stage is processed by a sub-module.

Idea

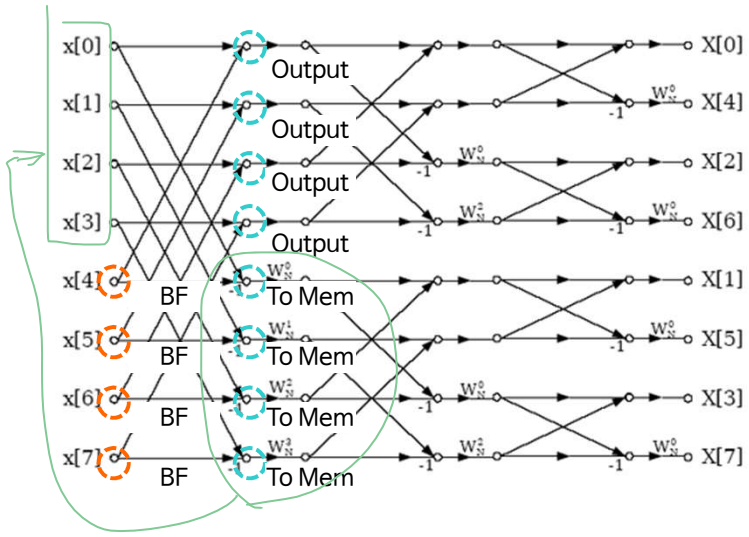
Can process butterfly?
Can output?



Idea

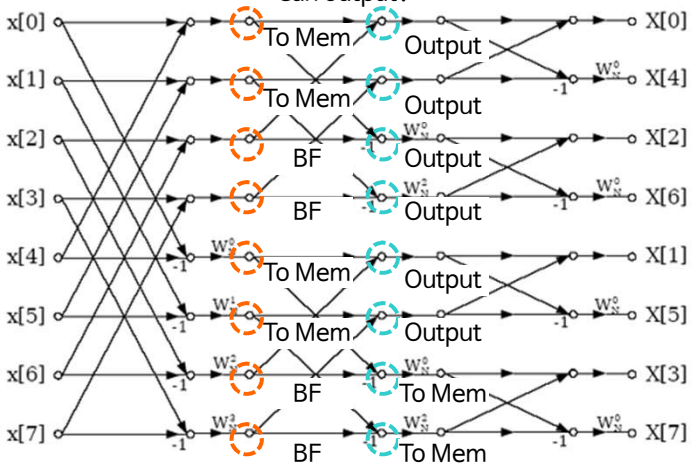


Idea

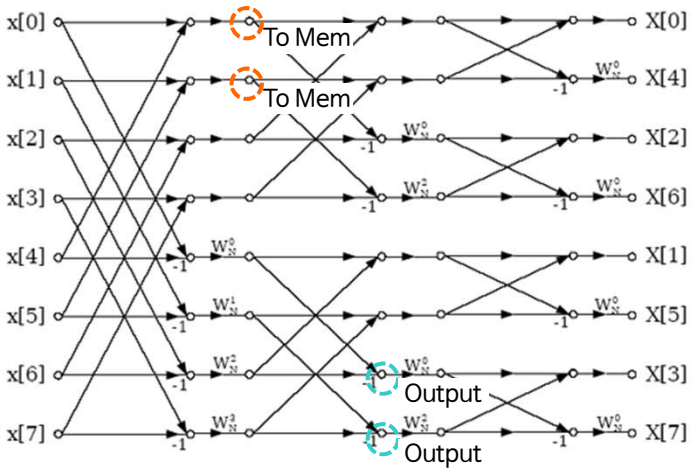


Idea

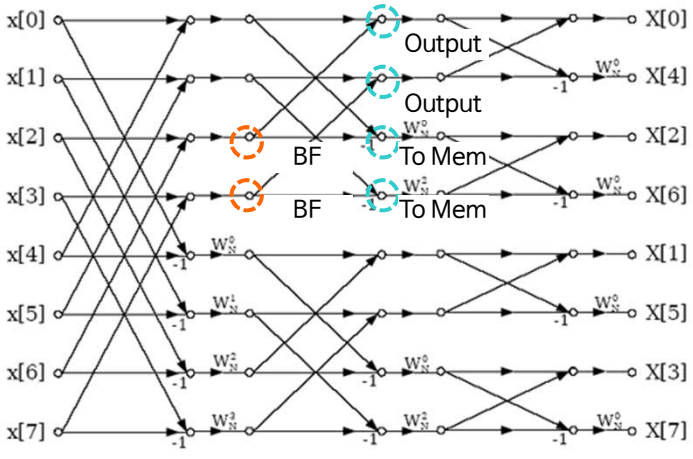
Can process butterfly?
Can output?

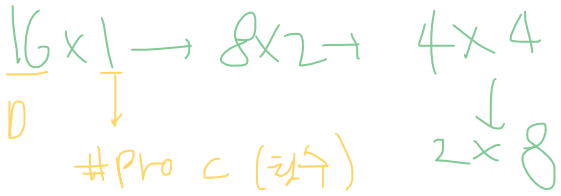


Idea


x[0]
x[1]
x[2]
x[3]
x[4]
x[5]
x[6]
x[7]
To Mem
To Mem
Output
Output
X[0]
X[4]
X[2]
X[6]
X[1]
X[5]
X[3]
X[7]

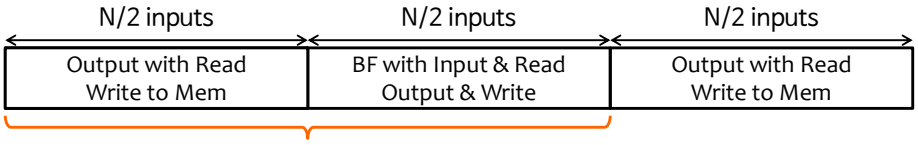
Idea


x[0]
x[1]
x[2]
x[3]
x[4]
x[5]
x[6]
x[7]
Output
Output
BF
BF
To Mem
To Mem
X[0]
X[4]
X[2]
X[6]
X[1]
X[5]
X[3]
X[7]

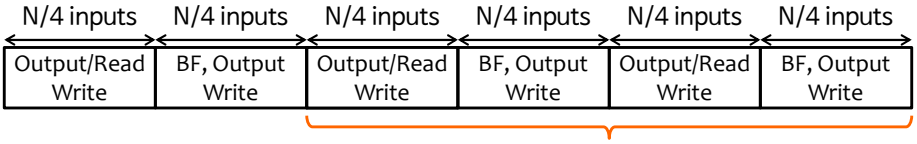


Stage Module Operation

- N-point butterfly stage



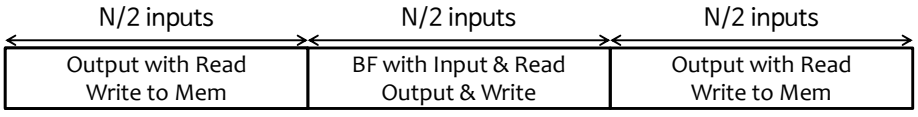
- Next (N/2-point butterfly) stage



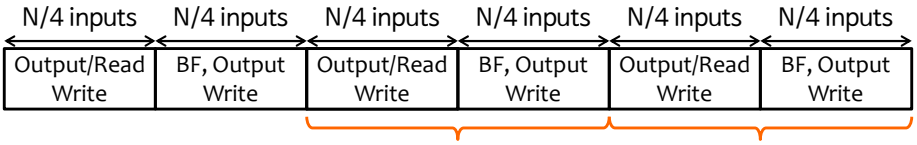
Stage-Level Pipeline

Stage Module Operation

- N-point butterfly stage



- Next (N/2-point butterfly) stage



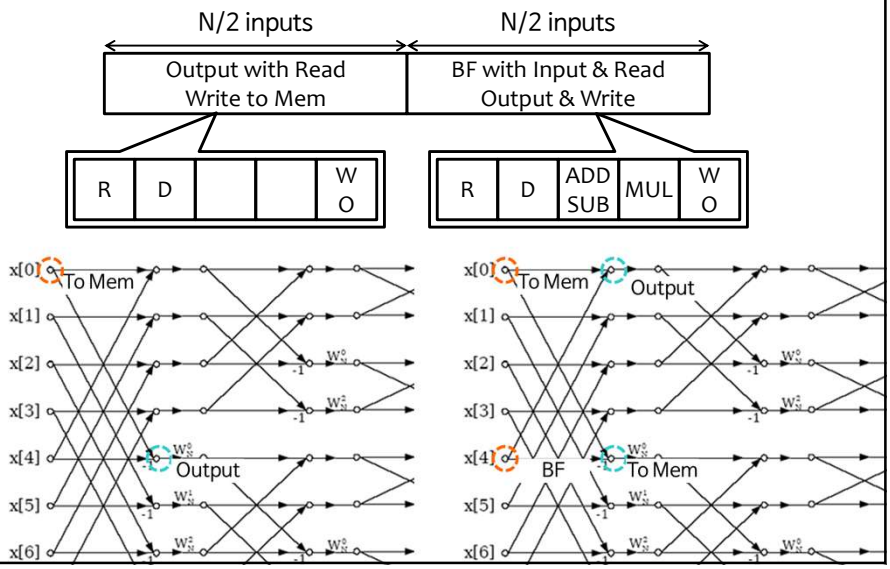
  - No difference between butterfly blocks
    - No need to count butterfly blocks

Outline

- Fast Fourier Transform (FFT)
- C Implementation
- Fixed Point Implementation
- Verilog Implementation – Memory-based
- Verilog Implementation – Pipelined
  - Idea
  - Butterfly Stage
  - Integration & Testbench
  - Configurable Module & Further Works

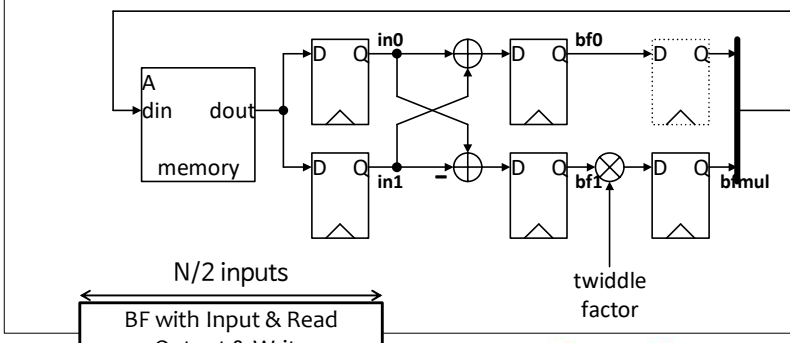
Stage Module Operation

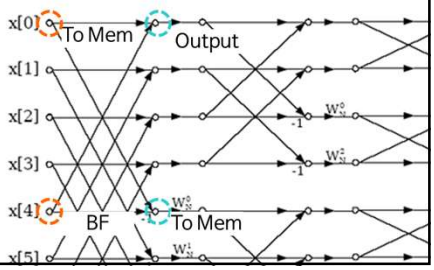
- Operation cycle



Stage Module Datapath

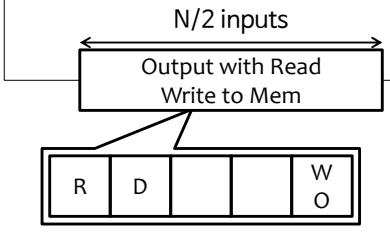
Let's modify the previous datapath.

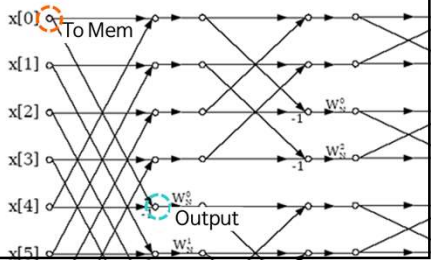




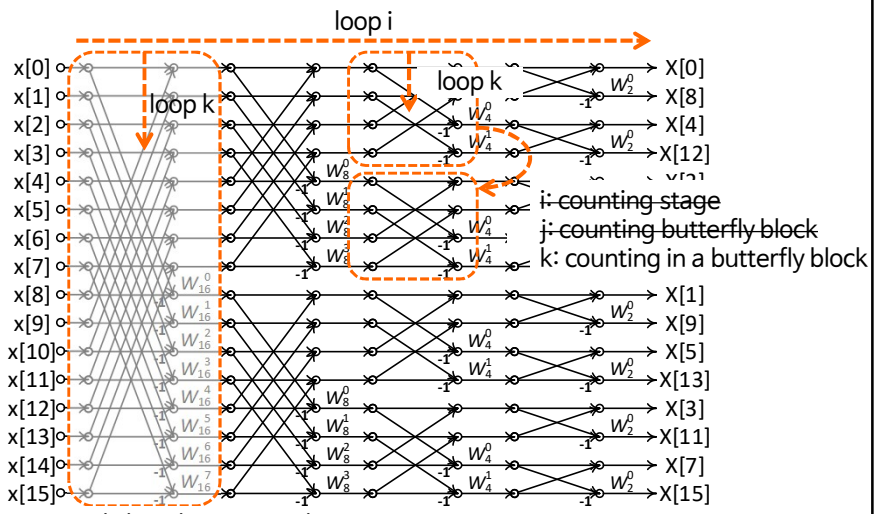
Stage Module Datapath

Let's modify the previous datapath.





Stage o Module



Only loop k is required to count 1024 inputs.

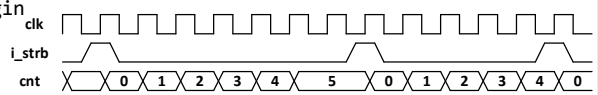
Module Input/Output

```
module fft_pipe_stg0 (
               input      clk
             , input      n_reset

             , input                 i_strb
             , input      [31:0]     i_data    } strobed input

             , output                o_strb
             , output     [31:0]     o_data    } strobed output
);


reg      [31:0]  i_data_d;
reg      [9:0]   cnt_k;  // counting input data
reg      [2:0]   cnt_o;  // counting butterfly operation
reg              first_garbage; ------> to be explained later
```

Counting

```
wire        last_o = (cnt_o == 4);
wire        last_k = (cnt_k == 1023);

always@(posedge clk or negedge n_reset) begin
    if(n_reset == 1'b0) begin
        cnt_k <= 'b0;
        cnt_o <= 7;
        i_data_d <= 'b0;
        first_garbage <= 1'b1;
    end else begin




        if(i_strb == 1'b1) i_data_d <= i_data;
    end
end
```

R	D	ADD SUB	MUL	W O



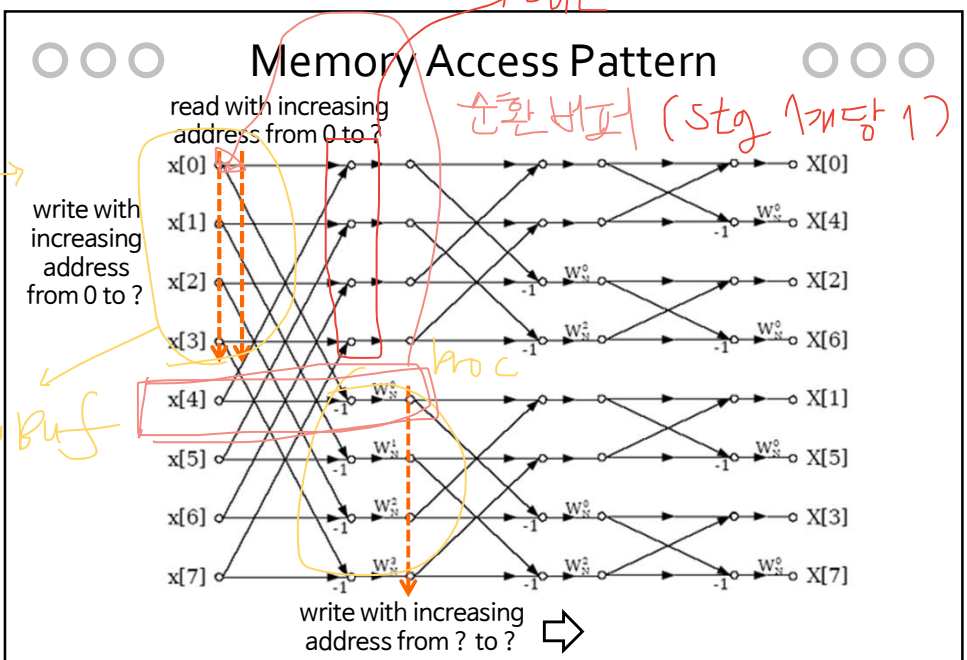
Let's describe cnt_o.

Memory Access

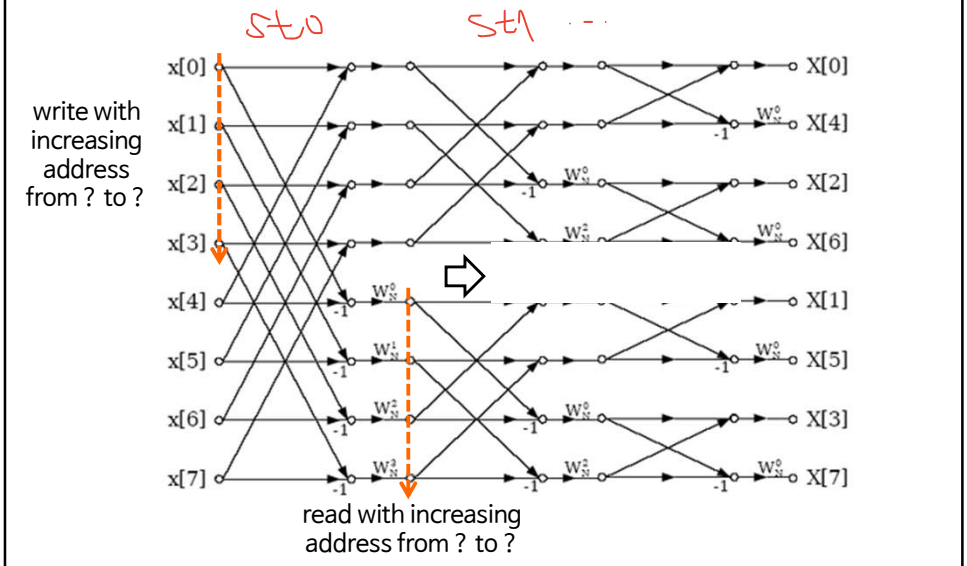
```
wire            cs;
wire            we;
reg     [8:0]   addr;
wire    [31:0]  r_data;
wire    [31:0]  w_data;

assign  cs =
assign  we =
always@(posedge clk or negedge n_reset) begin
    if(n_reset == 1'b0) begin
        addr <= 'b0;
    end else begin

    end
end
```

R	D	ADD SUB	MUL	W O

Memory Access Pattern



Memory Access Pattern



Address Calculation

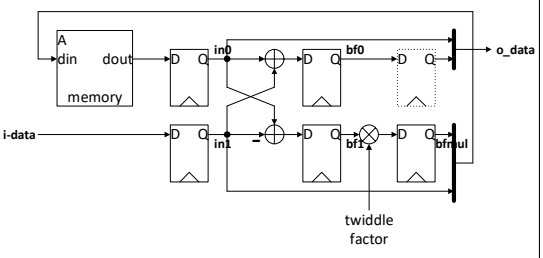
```
wire            cs;
wire            we;
reg     [8:0]   addr;
wire    [31:0]  r_data;
wire    [31:0]  w_data;

assign  cs = (cnt_o == 0) || (cnt_o == 4);
assign  we = (cnt_o == 4);
always@(posedge clk or negedge n_reset) begin
    if(n_reset == 1'b0) begin
        addr <= 'b0;
    end else begin

    end
end
```

Let's describe addr.

Datapath

Let's describe the following signals.
- in0(in0_r, in0_i), in1(in1_r, in1_i)

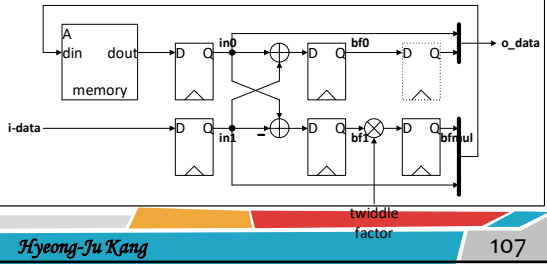
R	D	ADD SUB	MUL	W O



Datapath

Let's describe the following signals.
- bf0(bf0_r, bf0_i), bf1(bf1_r, bf1_i)

R	D	ADD SUB	MUL	W O



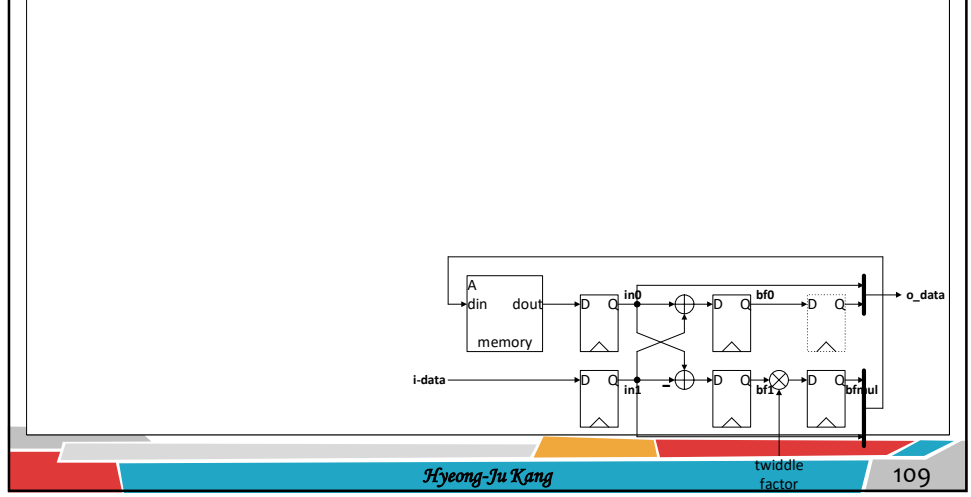
Twiddle Factor Generation

Let's describe the following signals.
- twid_lut
- twid_r, twid_i

R	D	ADD SUB	MUL	W O

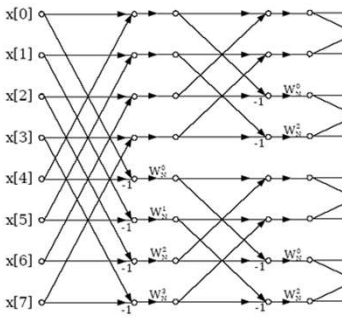
Datapath

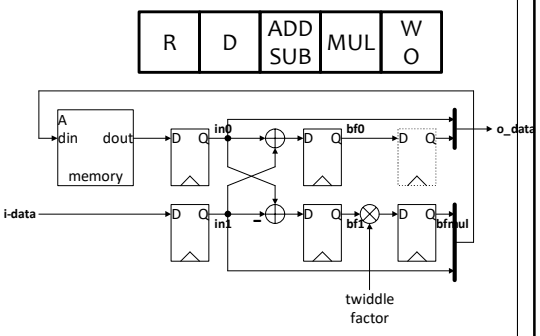
Let's describe the following signals.
- bfmul(bfmul_r, bfmul_i)



Output

Let's describe the following signals and instantiate a memory.
- w_data
- o_strb, o_data



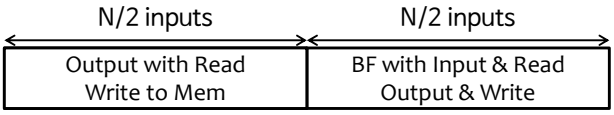


Memory Instantiation

```
mem_single #(
               .WD(32)
             , .DEPTH(512)
) i_mem (
               .clk(clk)
             , .cs(cs)
             , .we(we)
             , .addr(addr)
             , .din(w_data)
             , .dout(r_data)
);

endmodule
```


Garbage Output

- N-point butterfly stage

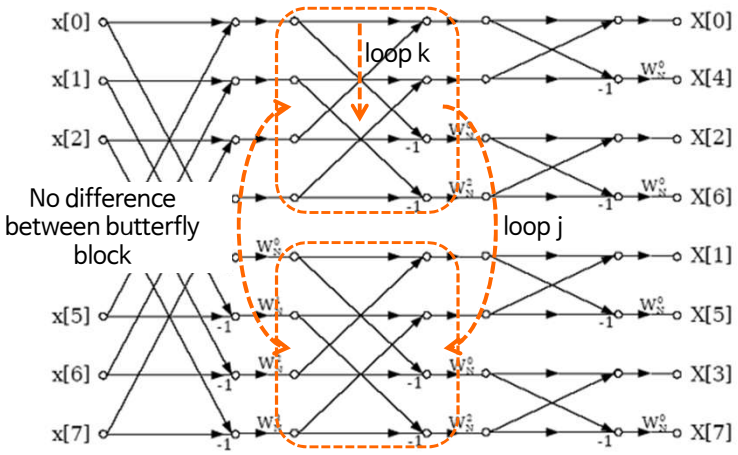


  - At the first half of inputs, the results of the previous inputs are going out from memory.
  - At the very first, garbage data are going out.
    - Those should be blocked.

Garbage Block

```
reg             first_garbage;
...
always@(posedge clk or negedge n_reset) begin
    if(n_reset == 1'b0) begin
        ...
        first_garbage <= 1'b1;
    end else begin
        ...
        if(last_o == 1'b1) begin
            ...
            if(cnt_k == 511) first_garbage <= 1'b0;
        end
        ...
    end
end
...
assign o_strb = (cnt_o == 4) && (first_garbage == 1'b0);
...
```


Stage 1 Module



Only loop k is required to count 512 inputs.

Stage 1 – Modification

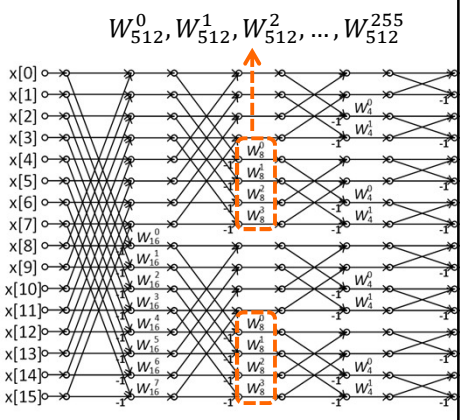
1024 points
→ 512 points

```
module fft_pipe_stg0 (

reg     [9:0]   cnt_k;
wire            last_k = (cnt_k == 1023);

if(cnt_k == 511) first_garbage <= 1'b0;

reg     [8:0]   addr;

wire signed [9:0]   cos = (cnt_k[8:0]<256) ? twid_lut[9:0]: twid_lut[19:10];
wire signed [9:0]   sin = (cnt_k[8:0]<256) ? twid_lut[19:10]: -twid_lut[9:0];

case(cnt_k[7:0])

assign w_data = (cnt_k < 512) ? i_data_d : {bfmul_i, bfmul_r};
assign o_data = (cnt_k < 512) ? {in0_i[15:0], in0_r[15:0]} : {bf0_i, bf0_r};

mem_single #(
              .WD(32)
            , .DEPTH(512)
) i_mem (
```


Stage 1 – Look-Up Table

- Stage 1 $\cos -2\pi k/N + j \sin -2\pi k/N$
  - 0/512, 1/512, 2/512, …, 255/512

k	values
0	$\cos 2\pi(-0/512), \sin 2\pi(-0/512)$
1	$\cos 2\pi(-1/512), \sin 2\pi(-1/512)$
2	$\cos 2\pi(-2/512), \sin 2\pi(-2/512)$
3	$\cos 2\pi(-3/512), \sin 2\pi(-3/512)$
4	$\cos 2\pi(-4/512), \sin 2\pi(-4/512)$
5	$\cos 2\pi(-5/512), \sin 2\pi(-5/512)$
6	$\cos 2\pi(-6/512), \sin 2\pi(-6/512)$
7	$\cos 2\pi(-7/512), \sin 2\pi(-7/512)$
…	
255	$\cos 2\pi(-255/512), \sin 2\pi(-255/512)$



Stage 1 – LUT Generation

1024 points
→ 512 points

```
#include <stdio.h>
#include <math.h>

int main(void) {
    const float pi = acos(-1.0);
    int k, c, s;
    for(k=0;k<256;k++) {
        c = floor(cos(-2*pi*k/1024)*511 + 0.5);
        s = floor(sin(-2*pi*k/1024)*511 + 0.5);
        printf("\t\t%d: twid_lut = {-10'd%d,10'd%d};\n", k, -s, c);
    }
}
```


Stage 2 – Modification

1024 points
→ 256 points

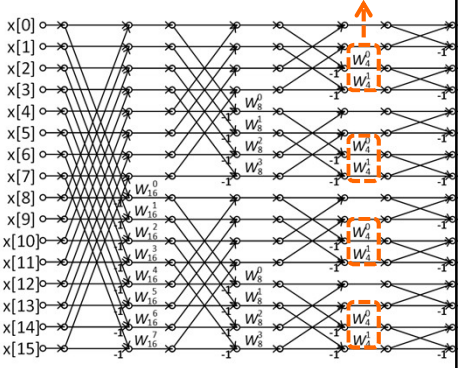
```
module fft_pipe_stg0 (

reg     [9:0]   cnt_k;
wire            last_k = (cnt_k == 1023);

if(cnt_k == 511) first_garbage <= 1'b0;

reg     [8:0]   addr;

wire signed [9:0]   cos = (cnt_k[8:0]<256) ? twid_lut[9:0]: twid_lut[19:10];
wire signed [9:0]   sin = (cnt_k[8:0]<256) ? twid_lut[19:10]: -twid_lut[9:0];

case(cnt_k[7:0])

assign w_data = (cnt_k < 512) ? i_data_d : {bfmul_i, bfmul_r};
assign o_data = (cnt_k < 512) ? {in0_i[15:0], in0_r[15:0]} : {bf0_i, bf0_r};

mem_single #(
              .WD(32)
            , .DEPTH(512)
) i_mem (
```

Stage 2 – Look-Up Table

- Stage 2 $\cos -2\pi k/N + j \sin -2\pi k/N$
  - 0/256, 1/256, 2/256, …, 127/256

k	values
0	$\cos 2\pi(-0/256), \sin 2\pi(-0/256)$
1	$\cos 2\pi(-1/256), \sin 2\pi(-1/256)$
2	$\cos 2\pi(-2/256), \sin 2\pi(-2/256)$
3	$\cos 2\pi(-3/256), \sin 2\pi(-3/256)$
4	$\cos 2\pi(-4/256), \sin 2\pi(-4/256)$
5	$\cos 2\pi(-5/256), \sin 2\pi(-5/256)$
6	$\cos 2\pi(-6/256), \sin 2\pi(-6/256)$
7	$\cos 2\pi(-7/256), \sin 2\pi(-7/256)$
…	
127	$\cos 2\pi(-127/256), \sin 2\pi(-127/256)$

$W_{256}^{0}, W_{256}^{1}, W_{256}^{2}, \ldots, W_{256}^{127}$



Stage 2 – LUT Generation

```
#include <stdio.h>
#include <math.h>

int main(void) {
    const float pi = acos(-1.0);
    int k, c, s;
    for(k=0;k<256;k++) {
        c = floor(cos(-2*pi*k/1024)*511 + 0.5);
        s = floor(sin(-2*pi*k/1024)*511 + 0.5);
        printf("\t\t%d: twid_lut = {-10'd%d,10'd%d};\n", k, -s, c);
    }
}
```

1024 points
→ 256 points

Other Stages

- Let's make modules for stage 3, 4, ..., 7.
  - 128, 64, ..., 8 points

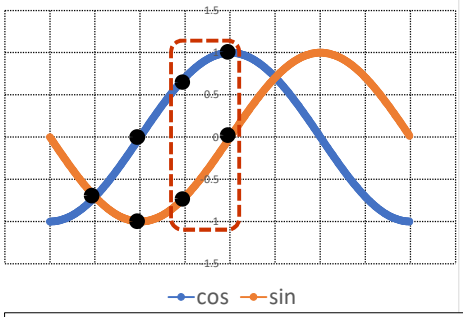
Stage 8 – Modification

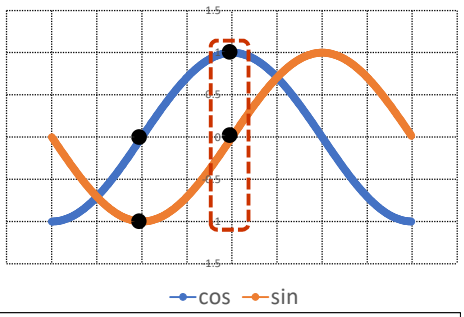
1024 points
→ 4 points

```
module fft_pipe_stg0 (

reg     [9:0]   cnt_k;
wire            last_k = (cnt_k == 1023);

if(cnt_k == 511) first_garbage <= 1'b0;

reg     [8:0]   addr;

wire signed [9:0]   cos = (cnt_k[8:0]<256) ? twid_lut[9:0]: twid_lut[19:10];
wire signed [9:0]   sin = (cnt_k[8:0]<256) ? twid_lut[19:10]: -twid_lut[9:0];

case(cnt_k[7:0])

assign w_data = (cnt_k < 512) ? i_data_d : {bfmul_i, bfmul_r};
assign o_data = (cnt_k < 512) ? {in0_i[15:0], in0_r[15:0]} : {bf0_i, bf0_r};

mem_single #(
               .WD(32)
             , .DEPTH(512)
) i_mem (
```

Look-Up Table – Stage 7 & 8

- Stage 7 – 8 points
  - 0/8, 1/8, 2/8, 3/8
- Stage 8 – 4 points
  - 0/4 ,1/4






```
always@(*) begin
  case(cnt_k[0])
    0: twid_lut = {-10'd0,10'd511};
    1: twid_lut = {-10'd361,10'd361};
  endcase
end
```

```
always@(*) begin
  case(cnt_k[???])
    ?: twid_lut = {-10'd0,10'd511};
  endcase
end
```

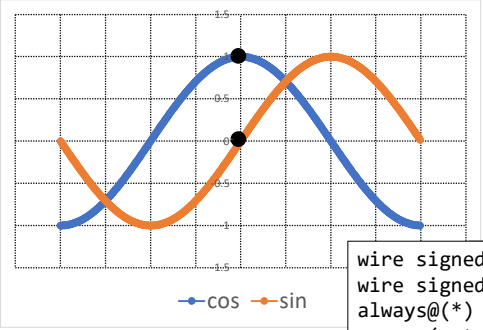

Stage 9 – Modification

1024 points
→ 2 points

```
module fft_pipe_stg0 (

reg     [9:0]   cnt_k;
wire            last_k = (cnt_k == 1023);

if(cnt_k == 511) first_garbage <= 1'b0;

reg     [8:0]   addr;

wire signed [9:0]   cos = (cnt_k[8:0]<256) ? twid_lut[9:0]: twid_lut[19:10];
wire signed [9:0]   sin = (cnt_k[8:0]<256) ? twid_lut[19:10]: -twid_lut[9:0];

case(cnt_k[7:0])

assign w_data = (cnt_k < 512) ? i_data_d : {bfmul_i, bfmul_r};
assign o_data = (cnt_k < 512) ? {in0_i[15:0], in0_r[15:0]} : {bf0_i, bf0_r};

mem_single #(
              .WD(32)
            , .DEPTH(512)
) i_mem (
```

Stage 9 – Look-Up Table

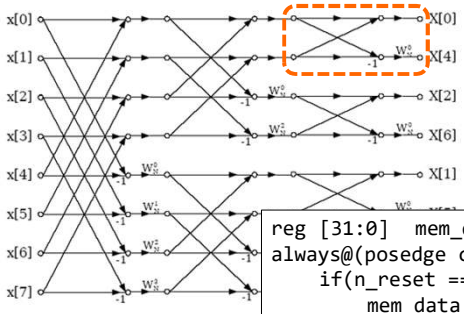
- Stage 9 – 2 points
  - 0/2




```
wire signed [9:0]    cos = (cnt_k[-1:0]<0) ? ...
wire signed [9:0]    sin = (cnt_k[-1:0]<0) ? ...
always@(*) begin
  case(cnt_k[???])
    ?: twid_lut = {-10'd0,10'd511};
  endcase
end
```


Stage 9 – Memory Instantiation

- The memory has only one entry.




```
reg [31:0]  mem_data;
always@(posedge clk or negedge n_reset) begin
    if(n_reset == 1'b0) begin
        mem_data <= 'b0;
    end else begin
        if((cs ==  1'b1) && (we == 1'b1)) begin
            mem_data <= w_data;
        end
    end
end
assign r_data = mem_data;
```

Outline

- Fast Fourier Transform (FFT)
- C Implementation
- Fixed Point Implementation
- Verilog Implementation – Memory-based
- Verilog Implementation – Pipelined
  - Idea
  - Butterfly Stage
  - Integration & Testbench
  - Configurable Module & Further Works

FFT Module

```
module fft (
          input     clk
        , input     n_reset

        , input             i_strb
        , input     [31:0]  i_data
        , output            o_strb
        , output    [31:0]  o_data
);

wire        o_strb0;
wire[31:0]  o_data0;
wire        o_strb1;
wire[31:0]  o_data1;
wire        o_strb2;
...
wire        o_strb9;
wire[31:0]  o_data9;
```

```
fft_pipe_stg0 i_stg0 (
                .clk(clk)
              , .n_reset(n_reset)

              , .i_strb(i_strb)
              , .i_data(i_data)
              , .o_strb(o_strb0)
              , .o_data(o_data0)
);

fft_pipe_stg1 i_stg1 (
                .clk(clk)
              , .n_reset(n_reset)

              , .i_strb(o_strb0)
              , .i_data(o_data0)
              , .o_strb(o_strb1)
              , .o_data(o_data1)
);
```

FFT Module

```
fft_pipe_stg3 i_stg3 (
               .clk(clk)
             , .n_reset(n_reset)

             , .i_strb(o_strb2)
             , .i_data(o_data2)
             , .o_strb(o_strb3)
             , .o_data(o_data3)
);
fft_pipe_stg4 i_stg4 (
               .clk(clk)
             , .n_reset(n_reset)

             , .i_strb(o_strb3)
             , .i_data(o_data3)
             , .o_strb(o_strb4)
             , .o_data(o_data4)
);

...
```

```
fft_pipe_stg8 i_stg8 (
               .clk(clk)
             , .n_reset(n_reset)

             , .i_strb(o_strb7)
             , .i_data(o_data7)
             , .o_strb(o_strb8)
             , .o_data(o_data8)
);
fft_pipe_stg9 i_stg9 (
               .clk(clk)
             , .n_reset(n_reset)

             , .i_strb(o_strb8)
             , .i_data(o_data8)
             , .o_strb(o_strb9)
             , .o_data(o_data9)
);

assign o_strb = o_strb9;
assign o_data = o_data9;

endmodule
```


Testbench – Initialization

```
module top_fft;

reg             clk, n_reset;
reg             i_strb;
reg     [31:0]  i_data;
wire            o_strb;
wire    [31:0]  o_data;

initial begin
    $vcdplusfile("top_fft.vpd");
    $vcdpluson(0, top_fft);
end
initial clk = 1'b0;
always #5 clk = ~clk;

import "DPI" function void init_fft();
import "DPI" function int unsigned get_input();
import "DPI" function int unsigned get_output();
```

Testbench – Input Stimulus

```
int     i;
initial begin
    n_reset = 1'b1;
    i_strb = 1'b0;
    i_data = 'bx;
    init_fft();
    #3;
    n_reset = 1'b0;
    #20;
    n_reset = 1'b1;
    @(posedge clk);
    @(posedge clk);
    repeat(2) begin
        for(i=0;i<1024;i++) begin
            #1;
            i_strb = 1'b1;
            i_data = get_input();
            @(posedge clk);
            #1;
            i_strb = 1'b0;
            i_data = 'bx;
            repeat(4) @(posedge clk);
        end
    end
    @(posedge clk);
    @(posedge clk);
    @(posedge clk);
    $finish;
end
```

4 cycle interval

after one input

Testbench – Output Check

```
fft i_fft (
                .clk(clk)
              , .n_reset(n_reset)
              , .i_strb(i_strb)
              , .i_data(i_data)
              , .o_strb(o_strb)
              , .o_data(o_data)
);

int j;
reg [31:0]  c_data;
initial begin
    for(j=0;j<1024;j++) begin
        @(posedge o_strb);
        @(posedge clk);
        c_data = get_output();
        if(o_data !== c_data) begin
            $display("Error: o_data[%d] = %8X, c_data = %8X"
                    , j, o_data, c_data);
        end
    end
end
endmodule
```

check output at o_strb

Outline

- Fast Fourier Transform (FFT)
- C Implementation
- Fixed Point Implementation
- Verilog Implementation – Memory-based
- Verilog Implementation – Pipelined
  - Idea
  - Butterfly Stage
  - Integration & Testbench
  - Configurable Module & Further Works

Paramerization

- Stage 0 ~ Stage 9 modules
  - 1024, 512, 256, ..., 2 points
  - Almost the same code
  - Different in a few numbers
  - → Let's parameterize.

Parameterization

1024 points
→ N points

```
module fft_pipe_stg_param #(
              N = 1024
) (
              input     clk
            , input     n_reset

            , input             i_strb
            , input     [31:0]  i_data
            , output            o_strb
            , output    [31:0]  o_data
);

localparam  W_K = $clog2(N);
reg     [31:0]      i_data_d;
reg     [W_K-1:0]   cnt_k;  // counting input data
reg     [2:0]       cnt_o;  // counting butterfly operation
reg                 first_garbage;

wire        last_o = (cnt_o == 4);
wire        last_k = (cnt_k == N-1);
```


Parameterization

1024 points
→ N points

```
if(cnt_k == 511) first_garbage <= 1'b0;

reg     [8:0]   addr;

wire signed [9:0]   cos = (cnt_k[8:0]<256) ? twid_lut[9:0]: twid_lut[19:10];
wire signed [9:0]   sin = (cnt_k[8:0]<256) ? twid_lut[19:10]: -twid_lut[9:0];

case(cnt_k[7:0])

assign w_data = (cnt_k < 512) ? i_data_d : {bfmul_i, bfmul_r};
assign o_data = (cnt_k < 512) ? {in0_i[15:0], in0_r[15:0]} : {bf0_i, bf0_r};

mem_single #(
              .WD(32)
            , .DEPTH(512)
) i_mem (
```

Twiddle Factor LUT

- Different LUTs should be described for each N.

```
if(N==1024) begin
    always@(*) begin
        case(cnt_k[7:0])
            0: twid_lut = {-10'd0,10'd511};
            1: twid_lut = {-10'd3,10'd511};
            ...
        endcase
    end
end else if(N==512) begin
    always@(*) begin
        case(cnt_k[6:0])
            0: twid_lut = {-10'd0,10'd511};
            1: twid_lut = {-10'd6,10'd511};
            ...
        endcase
    end
end else if(N==256) begin
    ...
end
```


Twiddle Factor LUT

- One LUT, but different indexing.

```
always@(*) begin
    case(cnt_k[W_K-3:0] << (10-W_K))
        0: twid_lut = {-10'd0,10'd511};
        1: twid_lut = {-10'd3,10'd511};
        2: twid_lut = {-10'd6,10'd511};
        ...
        253: twid_lut = {-10'd511,10'd9};
        254: twid_lut = {-10'd511,10'd6};
        255: twid_lut = {-10'd511,10'd3};
    endcase
end
```

For N=1024 and W_K=10,
cnt_k[7:0] ⟨⟨ 0

For N=512 and W_K=9,
cnt_k[6:0] ⟨⟨ 1

For N=256 and W_K=8,
cnt_k[5:0] ⟨⟨ 2

1024

- But, big LUTs for all modules?

Edge Cases

- If N = 4 or N = 2, isn't there a problem?

```
Consider the followings.
N = 4 → W_K = 2
N = 2 → W_K = 1
```

1024 points
→ N points

Twiddle Factor Generation

```
wire signed [9:0]   cos;
wire signed [9:0]   sin;
wire        [7:0]   lut_idx;
if(N>4) begin


end else if(N==4) begin


end else if(N==2) begin


end
```

normal cases

edge cases
→cases discussed in the previous section (stage 8 and stage 9)

Memory Instantiation

```
reg     [W_K-1:0]    addr;    // W_K-2 is right.
if(N>2) begin
```

normal cases

```
end else begin
```

edge cases
→cases discussed in the previous section (stage 9)

```
end
```


Naïve Integration

```
module fft (
          input     clk
        , input     n_reset

        , input             i_strb
        , input     [31:0]  i_data
        , output            o_strb
        , output    [31:0]  o_data
);

wire        o_strb0;
wire[31:0]  o_data0;
wire        o_strb1;
wire[31:0]  o_data1;
wire        o_strb2;
...
wire        o_strb9;
wire[31:0]  o_data9;
```

```
fft_pipe_stg_param #(
                .N(1024)
) i_stg0 (
                .clk(clk)
              , .n_reset(n_reset)

              , .i_strb(i_strb)
              , .i_data(i_data)
              , .o_strb(o_strb0)
              , .o_data(o_data0)
);

fft_pipe_stg_param #(
                .N(512)
) i_stg1 (
                .clk(clk)
              , .n_reset(n_reset)

              , .i_strb(o_strb0)
              , .i_data(o_data0)
              , .o_strb(o_strb1)
              , .o_data(o_data1)
);
```

Naïve Integration

```
fft_pipe_stg_param #(
                .N(256)
) i_stg3 (
                .clk(clk)
              , .n_reset(n_reset)

              , .i_strb(o_strb2)
              , .i_data(o_data2)
              , .o_strb(o_strb3)
              , .o_data(o_data3)
);
...
fft_pipe_stg _param #(
                .N(4)
) i_stg8 (
                .clk(clk)
              , .n_reset(n_reset)

              , .i_strb(o_strb7)
              , .i_data(o_data7)
              , .o_strb(o_strb8)
              , .o_data(o_data8)
);
```

```
fft_pipe_stg _param #(
                .N(2)
) i_stg9 (
                .clk(clk)
              , .n_reset(n_reset)

              , .i_strb(o_strb8)
              , .i_data(o_data8)
              , .o_strb(o_strb9)
              , .o_data(o_data9)
);

assign o_strb = o_strb9;
assign o_data = o_data9;

endmodule
```

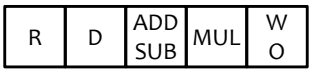
Integration With Generate

```
module fft (
        input           clk
      , input           n_reset
      , input              i_strb
      , input     [31:0]   i_data
      , output             o_strb
      , output    [31:0]   o_data

);
wire  [9:0]       i_strbs;
wire  [9:0][31:0] i_datas;
wire  [9:0]       o_strbs;
wire  [9:0][31:0] o_datas;

genvar   i;
for(i=0;i<10;i++) begin
   if(i==0) begin
      assign i_strbs[i] = i_strb;
      assign i_datas[i] = i_data;
   end else begin
      assign i_strbs[i] = o_strbs[i-1];
      assign i_datas[i] = o_datas[i-1];
   end
```

```
   fft_pipe_stg_param #(
           .N(1024/(2**i))
   ) i_stg (
           .clk(clk)
         , .n_reset(n_reset)

         , .i_strb(i_strbs[i])
         , .i_data(i_datas[i])
         , .o_strb(o_strbs[i])
         , .o_data(o_datas[i])
   );
end

assign o_strb = o_strbs[9];
assign o_data = o_datas[9];

endmodule
```

Further Works

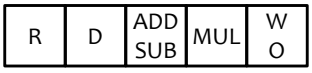
- Operation level pipeline



Let's draw operation level pipeline.

Further Works

- Operation level pipeline



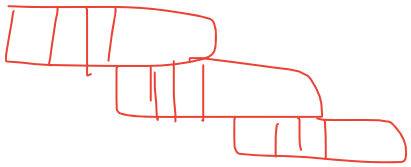
Let's draw operation level pipeline.

Further Works

- Operation level pipeline

R	D	ADD SUB	MUL	W O

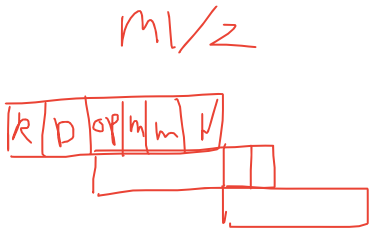
Let's draw operation level pipeline.



Further Works

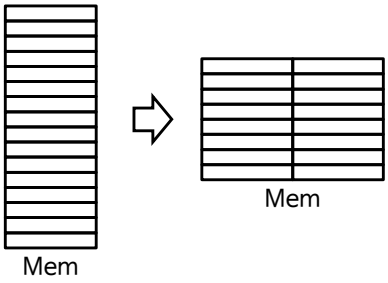
- How to reduce cycles?

Let's reduce the average cycles/butterfly to 2.



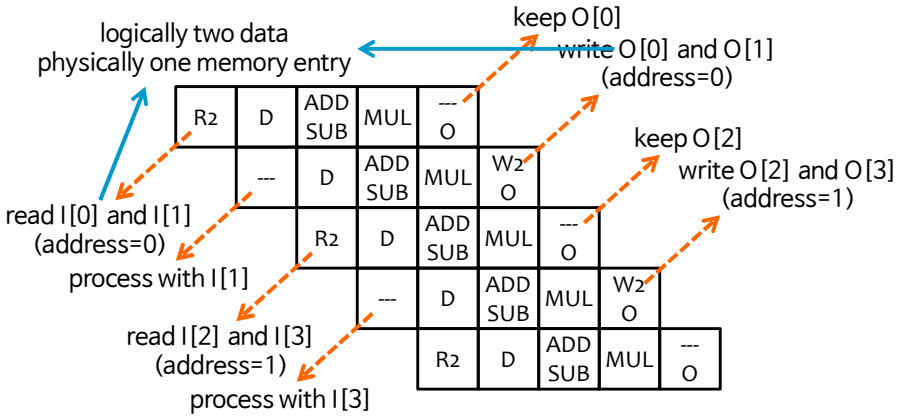
Further Works – Pairing

- 2 memory accesses are required.
  - 1 read and 1 write
- Pairing
  - Two data on consecutive places are paired into one entry.



Further Works – Pairing

- Pairing
  - Two data on consecutive places are paired into one entry.



Further Works – Pairing

- Pairing
  - Two data on consecutive places are read and write at the same time.
  - Pairing is possible because of consecutive accesses with incrementing addresses.

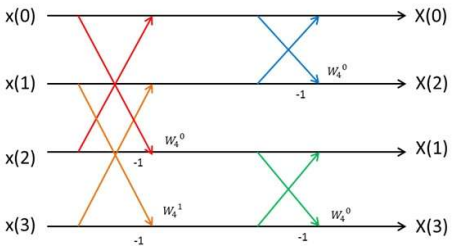
Further Works – FFT Specific

	Memory-based	Pipelined
Memory Size	N	N/2 + N/4 + … + 1 = N
Cycles/Frame	log N * N/2 * C / BF	N * C
# of BF	-	log N

- Operator efficiency is higher in memory-based.
  - BF operators in the pipelined structure do nothing in the half of the time.
  - Can we place many BFs in the memory-based structure?
- Equal memory requirement?
  - The data width should be increased by 1-bit at each stage.
  - At the final stage, the data have additional log N bits.
  - In the memory-based structure, the memory should be able to accommodate the longest bit width.

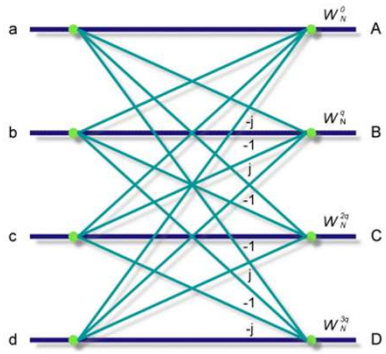
Further Works – FFT Specific

- Radix-4 FFT



Two stages of
Radix-2 Butterfly

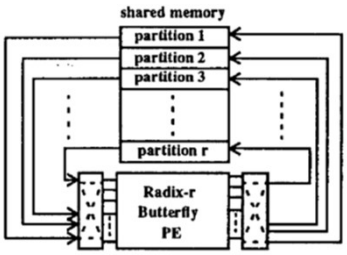
4 complex multipliers
for 4 radix-2 BF

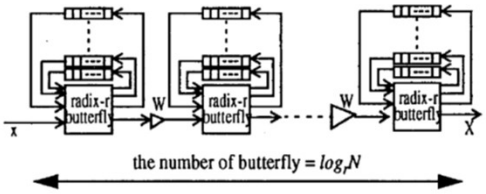


Radix-4 Butterfly

3 complex multipliers
for 1 radix-4 BF

Radix-r Structures





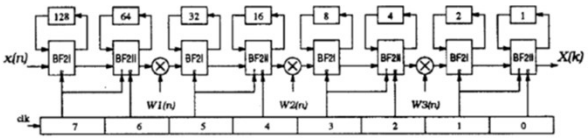


Figure 4: R2^{2}SDF pipeline FFT architecture for $N = 256$