同济大学计算机科学与技术系

信号处理导论　大作业

FFT 硬件加速的原理与实践

实验报告

学　　号：1951505

姓　　名：姜星宇

指导教师：张冬冬

日　　期：2022/6/20

目录

1 介绍

1.1 实验背景

傅里叶变换（Fourier transform，FT）是一种将关于空间或时间的函数向关于空间频率或瞬时频率的函数转换的一种数学变换[1]。1822 年，法国科学家傅里叶在研究热力学时，发现无论是连续还是非连续的函数，都可以被分解为一系列正弦的级数。经过一段时间的研究，傅里叶变换逐渐被数理科学等领域广泛应用。

起初，傅里叶变换被应用在简化一些例如微分方程等问题上，在频域上有些函数将变得更加容易分析。例如在傅里叶热传导定律结合高斯定理

$$\oiint_S \kappa \nabla T \cdot \mathrm{d}\boldsymbol{S} = \oiint_S \boldsymbol{p} \cdot \mathrm{d}\boldsymbol{S} = \iiint_V \frac{\partial e}{\partial t} \mathrm{d}V \Leftrightarrow \frac{\kappa}{\rho c_m} \nabla \cdot (\nabla e) = \frac{\partial e}{\partial t} \tag{1-1}$$

在单方向上可以简化为关于能量密度的偏微分方程

$$C \frac{\partial^2 e}{\partial x^2} = \frac{\partial e}{\partial t}. \tag{1-2}$$

在偏微分方程理论中，这个方程通常需要根据边界条件确定特解。对于上述二阶波动方程，需要两个边界条件共同确定，这将会变得比较复杂。如果采用傅里叶变换，可以将这种特定形式的方程根据边界条件的频率确定解的形式与参数ξ。具体的领域除了这里的较早的热力学，近代物理中例如量子力学等需要用到波动形式偏微分方程的领域同样具有广泛的应用。

同时，卷积运算在频域上等同于简单的乘法，因此为理论的分析上提供了一个强大的工具。因此在工业界，信号处理中，傅里叶变换具有不可替代的作用。例如信号处理中的自相关函数

$$R(\tau) = \lim_{T \to \infty} \frac{1}{2T} \int_{-T}^{T} f(t) f(t+\tau) \mathrm{d}t \tag{1-3}$$

在傅里叶变换后得到的功率谱密度函数，可以表示出不同频率对于功率的贡献程度，从而反应信号强弱分布的规律与性质。

1.2 傅里叶变换

傅里叶变换（FT）关于频率ξ的定义如下

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) \mathrm{e}^{-\mathrm{i}2\pi\xi x} \mathrm{d}x, \tag{1-4}$$

逆变换为

$$f(x)=\int_{-\infty}^{\infty}\hat{f}(\xi)\mathrm{e}^{\mathrm{i}2\pi\xi x}\mathrm{d}\xi\,. \tag{1-5}$$

离散时间傅里叶变换（discrete-time Fourier transform，DTFT）应用于离散的时间序列上，有如下形式

$$\hat{f}(\xi)=\sum_{x=-\infty}^{\infty}f[x]\mathrm{e}^{-\mathrm{i}2\pi\xi x}\Delta x\,, \tag{1-6}$$

当$\Delta x=1/T$时逆变换为

$$f[x]=T\int_{\frac{1}{T}}\hat{f}(\xi)\mathrm{e}^{\mathrm{i}2\pi\xi xT}\mathrm{d}\xi\,. \tag{1-7}$$

离散傅里叶变换（discrete Fourier transform，DFT）在离散时间傅里叶变换的基础上进行有限等长采样，即

$$\hat{f}[\xi]=\sum_{x=0}^{N-1}f[x]\mathrm{e}^{-\mathrm{i}2\pi\xi x}\frac{1}{N} \tag{1-8}$$

或者令$\xi=\Xi\,k/N$，$x=X\,n/N$，$\Xi X=N$，（采取这种假设求得的离散傅里叶变换频谱的分辨率$\frac{\Xi}{N}=1/\left(\frac{X}{N}N\right)=\frac{f_s}{N}$，$\Xi$为采样间隔的倒数即采样率$f_s$）

$$\hat{f}\left[\Xi\frac{k}{N}\right]=\sum_{x=0}^{N-1}f\left[X\frac{n}{N}\right]\mathrm{e}^{-\mathrm{i}2\pi\Xi\frac{k}{N}X\frac{n}{N}}\frac{1}{N}=\sum_{x=0}^{N-1}f\left[X\frac{n}{N}\right]\mathrm{e}^{-\mathrm{i}2\pi\frac{kn}{N}}\frac{1}{N}, \tag{1-9}$$

归一化后写为

$$X_k=\sum_{n=0}^{N-1}x_n\mathrm{e}^{-\frac{\mathrm{i}2\pi}{N}kn}\,, \tag{1-10}$$

逆变换则为

$$x_n=\frac{1}{N}\sum_{k=0}^{N-1}X_k\cdot\mathrm{e}^{\frac{\mathrm{i}2\pi}{N}kn}\,. \tag{1-11}$$

可以认为，从连续函数的傅里叶变换（或者离散时间傅里叶变换）中进行一段时间的采样，再离散化，可以得到离散傅里叶变换，即

$$\int_{-\infty}^{\infty}f(x)\mathrm{e}^{-\mathrm{i}\omega x}\mathrm{d}x\Rightarrow\int_{0}^{X}f(x)e^{-i\omega x}\mathrm{d}x\Rightarrow\sum_{n=0}^{N}f\left(X\frac{n}{N}\right)\mathrm{e}^{-\mathrm{i}\omega X\frac{n}{N}}\frac{X}{N} \tag{1-12}$$

归一化系数X/N并和频域、时域的单位刻度后所得结果。

经过离散傅里叶变换的离散化后，就可以使用计算机存储与计算。直接使

用定义公式计算的时间复杂度是$O(N^2)$，而在计算中，每一个指数因子是一样的，因此其中包含一些重复运算，而实际的复杂度可以通过快速傅里叶变换算法（fast Fourier transform，FFT）降至$O(N \log N)$[2]。最常用的 FFT 算法为 Cooley-Turkey FFT 算法，采用分治策略，将$1 \times N$的 DFT 转换为$N_1 \times N_2$计算。令$N_1 < N_2$，N_1则称为基数，通常采用基数为 2 的蝶形二分算法如下。

```
Function fft
input: x, N, s
output: X[0 .. N - 1]
if N = 1 then
    X[0] <- x[0]
else
    X[0 .. N / 2 - 1] <- fft(x, N / 2, 2 * s)
    X[N / 2 .. N - 1] <- fft(x + s, N / 2, 2 * s)
    for k = 0 to N / 2 - 1 do
        p <- X[k]
        q <- exp(-2πi * k / N) * X[k + N / 2]
        X[k] = p + q
        X[k + N / 2] = p - q
    end for
end if
```

图 1-1 Cooley-Turkey FFT 算法

上述 FFT 算法的每一步的运算通常可以使用下图描述。

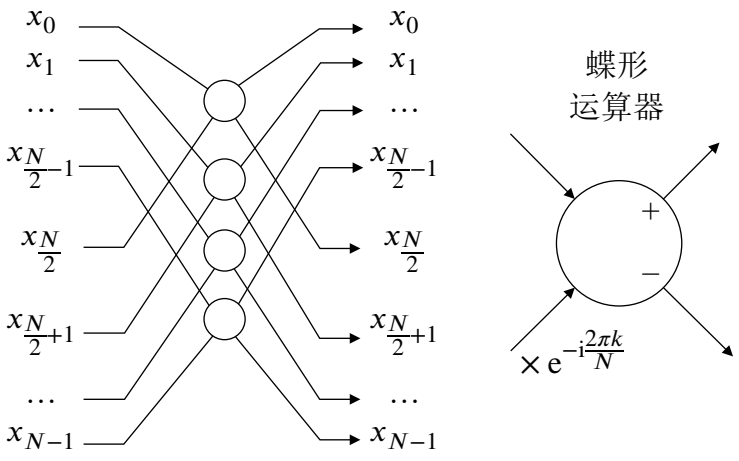

图 1-2 运算器单元

但是左侧输入的运算操作数在上一级来源于不同的下标，因此如果从$N = 1$开始运算，初始下标的计算将十分复杂，不能按照实际输入的顺序。这种情形如果使用从N到 1 的递归，一般就需要使用额外的$O(N \log N)$的空间存储后选取计算。因此有如下非递归算法解决这一问题，称为迭代 FFT。这个算法首先对

于下标进行调整，从而可以方便地通过循环找到每一次计算对应的一系列数。

```
Function iterative-fft
input: x[0, .. N]
output: X[0 .. N - 1]
for k = 0 to N - 1 do // bit reverse
    X[rev(k)] = x[k]
end for
for s = 1 to log(N) do
    m <- 2 ^ s
    wm <- exp(-2πi / m)
    for k = 0 to N - 1 by m do
        w <- 1
        for j = 0 to m / 2 - 1 do
            p <- X[k + j]
            q <- X[k + j + m / 2] * w
            X[k + j] <- u + t
            X[j + j + m / 2] <- u - t
            w <- w * wm
        end for
    end for
end for
return X
```

图 1-3 FFT 算法的非递归形式

1.3 FFT 硬件实现

根据上述常用 FFT 算法的描述，可以发现，运算重复使用蝶形运算器，使用通常的 CPU 运算需要对于每一个运算器进行一次三角函数运算、一次乘法运算，以及两次加减法运算，这一系列运算需要重复约$\frac{N}{2}\log_2 N$次。

如果使用 FPGA 等定制芯片，则可以采用组合逻辑将运算器描述出来，并采用多个运算器进行并行计算，可以最大化 FFT 的效率。运算器的硬件描述需要浮点数加减法以及乘法的支持，同时对于余弦函数的运算，由于角度为$1/N$的整数倍，因此可以使用查表得出结果。

具体而言，使用定制芯片计算，首先需要确定浮点数的表示，根据 IEEE-754 标准，64 位双精度浮点型如下图，具有 1 位符号位、11 位指数位以及 52 位尾数位。

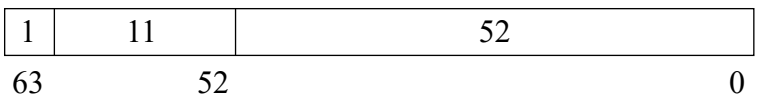

图 1-4 IEEE 64 位浮点数结构

其表示的数值为

$$(-1)^{\text{sign}}\left(1+\sum_{i=1}^{52}\text{fraction}_{52-i}2^{-i}\right)\times 2^{\text{exponent}-1023} \tag{1-13}$$

浮点数加法器与乘法器同样需要实现。还有就是复数的旋转，根据公式，

$$\text{Re}\left\{A\text{e}^{-\text{i}2\pi\frac{k}{s}}\right\}=\text{Re}\{A\}\cos 2\pi\frac{k}{s}+\text{Im}\{A\}\sin 2\pi\frac{k}{s} \tag{1-14}$$

$$\text{Im}\left\{A\text{e}^{-\text{i}2\pi\frac{k}{s}}\right\}=\text{Im}\{A\}\cos 2\pi\frac{k}{s}-\text{Re}\{A\}\sin 2\pi\frac{k}{s} \tag{1-15}$$

将$s=[M,N]$（M为并行数量，为 2 的整数幂）各项保存在内存中则可以结合乘法器求出数值。根据并行计算所需要的表中数值，应当将这些数值存储在不同的内存模块中，以达到并行查询。

1.4 实验目标及环境

完成采样点总数$N=4096$，并行数$M=16$的 FFT 模块，从内存中读取数据，并输出至内存。

实验环境为 Linux，Vivado 2018 以及 Digilent Nexys4 DDR 开发板。

2　模块设计

2.1　模块整体设计

为方面应用于展示效果，模块整体分为顶层模块、FFT 模块、内存模块和信号采集模块。其中 FFT 模块中包含运算器模块与控制器模块。

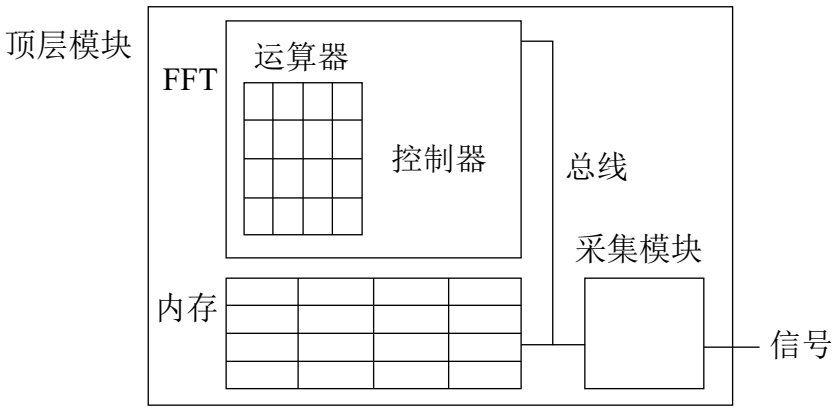

图 2-1 整体模块

2.2　FFT 模块

FFT 模块由于与内存的交互使用到了时序逻辑，与 CPU、GPU 架构类似，包含运算器与控制器。其中运算器为组合逻辑，输入三个双精度复数（6 个 64 位浮点型）a、b、$w=\mathrm{e}^{-\mathrm{i}\omega}$，输出为两个双精度复数 （4 个 64 位浮点型）c、d，示意图如下。a、b、c、d分别代表蝶形运算器的四个输入输出。

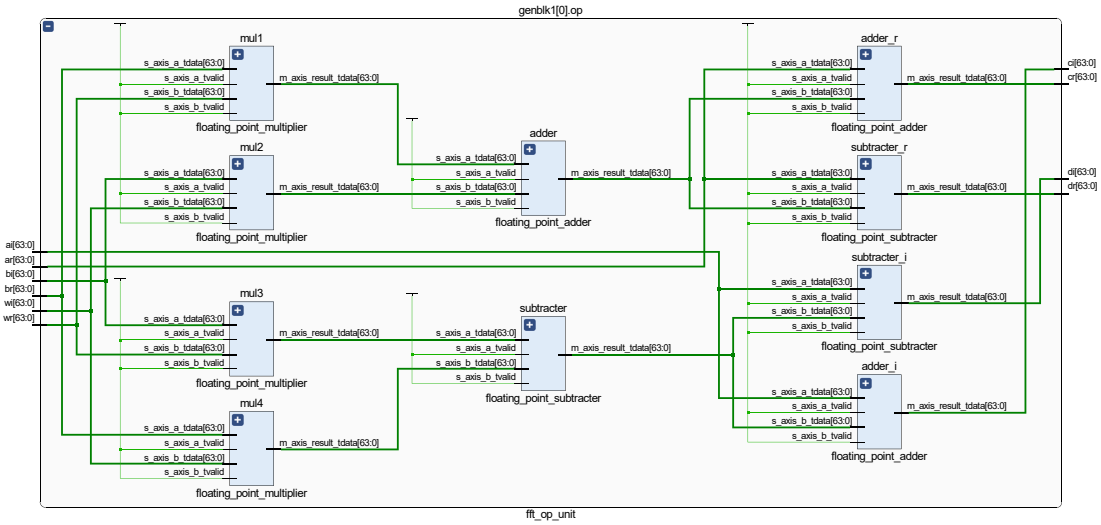

图 2-2 运算器单元示意图

控制器为时序逻辑，在输入信号 sig 的上升沿，将寄存器 busy 置 1，同时开

始将时钟信号输入给内存，从内存中按照 FFT 并行运行的时序依次读写。

首先使用 stage 寄存器对于算法整体的时序进行控制，0 代表将原数据根据按位反转的下标进行重排的操作，1代表使用蝶形运算器计算每一层的过程。在 1 的计算中，round 寄存器存储当前进行到的轮次，共$\log N$次，iter 表示在当前轮次中循环下标。根据 round 与 iter，可以共同确定各个运算器操作数的地址，从而实现循环计算。

图 1-3 中算法示意图如下。算法的意思是s从 1 开始，以2^s项为一组分为$N/2^s$组。随后在每一组内，分为上下两部分，每次从两部分中选取依次进行单元计算，计算输出的值重新写入原位，完成一次计算。可见地址$i_{1,2}$与轮次s（round）存在等间隔的关系

$$i_2 = i_1 + 2^{s-1} \tag{2-1}$$

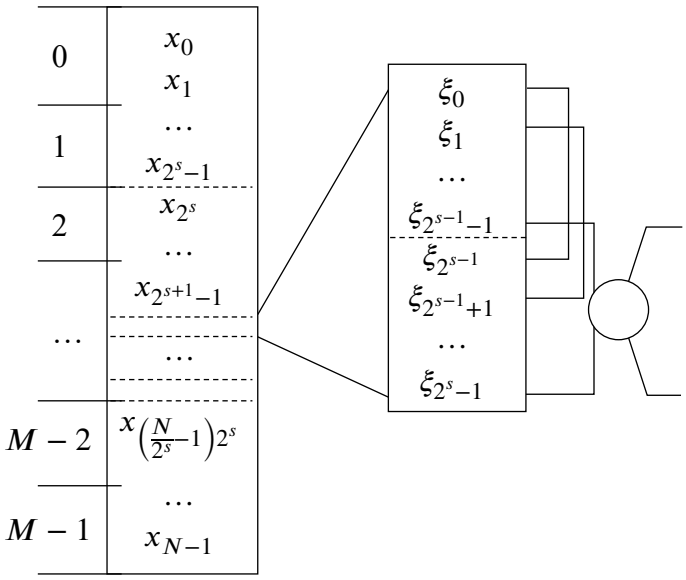

图 2-3

如果将内存分为2^m组，那么当$s \le n-m$时，计算的一组内存集中在内存的一组中。如果$s > n-m$，则一次计算将涉及到内存中的两组，但是由于同一组计算组中任意两组是平行等间隔的，因此每一组内存组中同样最多需要同时读写两个数。

通过上面对所需内存地址分布情况简单的考虑，将内存平均分为$M = 2^m$组，可以得出地址$A_{1,2}$、轮次$r \in (0, \log_2 N] = (0, n]$、循环下标$i \in [0, 2^{n-m-1})$与运算

器下标$k \in [0,2^m)$有如下关系。其中内存组G与组内下标I满足关系

$$G = A \ \mathbf{div}\ 2^{n-m}, I = A \ \mathbf{mod}\ 2^{n-m} \tag{2-2}$$

当$r \le n-m$时，k可以和内存组连续对应，

$$\begin{gathered} G_1 = k, I_1 = \left(i \ \mathbf{div}\ 2^{r-1}\right)2^r + i \ \mathbf{mod}\ 2^{r-1} \\ G_2 = G_1, I_2 = I_1 + 2^{r-1} \end{gathered} \tag{2-3}$$

当$r > n-m$时，i可以和内存组内下标连续对应，

$$\begin{gathered} G_1 = \left(k \ \mathbf{div}\ 2^{r-(n-m)}\right)2^{r-(n-m)} + k \ \mathbf{mod}\ 2^{r-1-(n-m)} \\ I_1 = i + \left(k \ \mathbf{mod}\ 2^{r-(n-m)} \ \mathbf{div}\ 2^{r-1-(n-m)}\right)2^{n-m-1} \\ G_2 = G_1 + 2^{r-1-(n-m)} \\ I_2 = I_1 \end{gathered} \tag{2-4}$$

具体硬件操作可以使用位运算，这样也会更加直观一些。

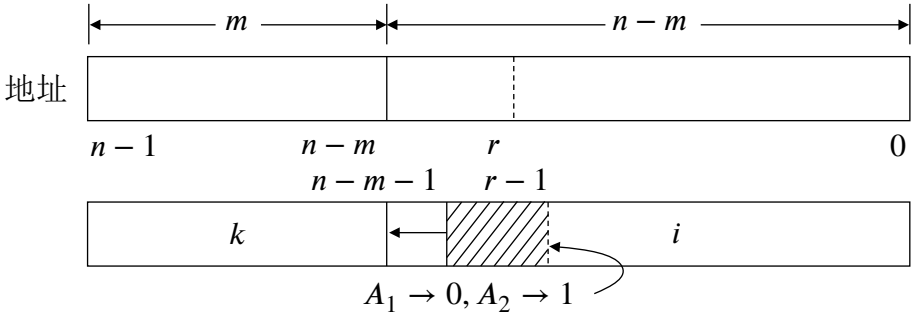

图 2-4 $r \le n-m$情形地址的位运算

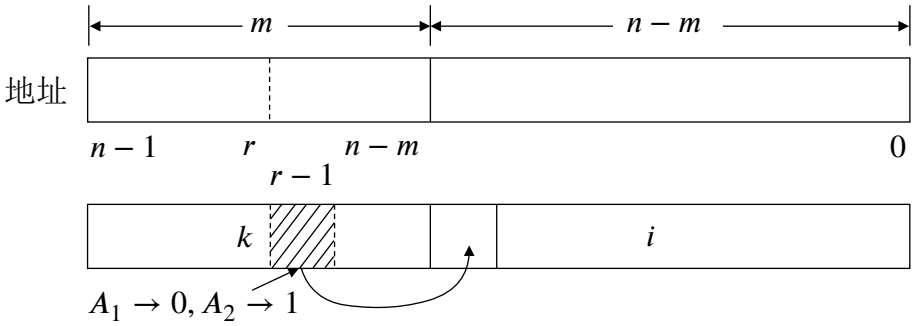

图 2-5 $r > n-m$情形地址的位运算

上面两张图表示的地址将组号k放在高m位，循环下标i放在低$n-m-1$位上。如果$r \le n-m$，则从第$r-1$位开始到$n-m-2$共$n-m-r$位左移 1 位，如果$r > n-m$，则将第$r-1$位直接移至$n-m-1$位上。随后将空出的$r-1$位填上 0 或 1 代表两个操作数的地址。

经过上面的位操作后，对于某一个i的所有地址，即仅k变化，$r \le n-m$时

显然对于不同的高m位，有且仅有两个不同的地址，因为高m位就是k，对应两个不同的补充位。而$r > n - m$时，先排除位操作的两位，高m位是全排列分布的，即每一个数出现且仅出现一次。变化的两位也是全排列的 4 个数，同时有一位在低位不参与高位的变化，因此不考虑顺序，则这 4 个中，两个对应补充 0 的高位地址，两个对应补充 1 的高位地址，因此每个高位地址也是有且仅有两个地址对应。

根据这个特点，直接根据顺序对内存进行分块就是可行的了，每一部分在每一个循环的计算中需要进行两个复数的输入输出，可以根据实际资源的情况进一步划分时序。

2.3 内存与信号采集模块

内存使用芯片中共 4860Kbit 的 BRAM，可以使用 Vivado 中提供的 IP 核完成此模块的构造。输入为地址和输入数据，输出为输出数据。

模块中使用到内存的部分主要为数据 RAM、旋转因子表 ROM。数据 RAM 需要在过程中存储复数的实部与虚部。按照 2.2 中关于控制器地址的讨论，需要将 RAM 进行分组以扩充 IO 并行数量，同时每个 RAM 需要两个端口同时进行两个数据的输入输出。因此对于$N = 4096, M = 16$的情形，需要 32 个宽度为 64，深度为 256 的双端口 RAM。

而以$N/2 = 2048$精度存储$[0, \pi)$正余弦的 ROM，每一个运算器都需要两个以存储所有的正余弦，因此需要 32 个宽度为 64，深度为 2048 的单端口 ROM。

综上所述，共需要$(2 + M)N \cdot 64\ \text{bit} = 4608\ \text{Kbit}$的内存，Nexy 4 DDR 的 BRAM 满足需要求，不需要使用更大且较慢的 SDRAM。实际上，大部分无效的存储在 ROM 上。由于可以只用一个 ROM 存储数据，在初始化时向M个总大小与 ROM 相同的 RAM 中按照每一个模块中写入当前轮次需要使用到的角度。但是这样大部分时间将会花在写入这些RAM上，并没有实际的意义。如果$M > \log_2 N$的话，可以为每一个轮次分配M组 ROM，分别填充不同的数值以分组计算。 还有一种方法就是使用更加一般的 CORDIC 等模块计算正余弦而不是通过查表，那样的话对于硬件资源消耗主要在门电路上，成本也可能会高一些。

而信号的采集，可以使用 Nexys 4 DDR 上的 mic 获取 PDM 信号（Pulse Density Modulation，脉冲密度调制），统计高电平的密度即高电平时钟周期个数，

转换为浮点数存储到 RAM 中，当存储满一次 RAM 后，就可以进行傅里叶变换了，经过计算的最终数据可以通过进一步的处理获得需要的结果。在开发板上直接的光学输出有一个七段数码管，对于最终的结果可以将信号中包含的最大的频率输出。

3 测试与应用

3.1 源代码

将上述各模块使用 Verilog 语言实现，通过 Vivado 仿真与实现，可以完成 FFT 的硬件实现。

完整源代码详见 https://github.com/jalaxy/Nexys-4-DDR-app/tree/main/FFT。

- 顶层模块：

图 3-1 顶层模块接口

- FFT 模块：

```
module fft(clk, rst, sig, we, rev, addr, din, dout);
input clk, rst, sig;
input we, rev;
input [`logN-1:0] addr;
input [`CW-1:0] din;
output [`CW-1:0] dout;
```

图 3-2 FFT 时序控制模块接口

- 按位反转模块：

```
module bit_reverse(din, dout);
input [`logN-1:0] din;
output [`logN-1:0] dout;
genvar i;
for (i = 0; i < `logN; i = i + 1) begin
    assign dout[i] = din[`logN - 1 - i];
end
endmodule
```

图 3-3 地址按位反转计算模块

- 运算器模块：

```
module fft_op_unit(ar, ai, br, bi, cr, ci, dr, di, wr, wi);
input [`FW-1:0] ar, ai, br, bi;
output [`FW-1:0] cr, ci, dr, di;
input [`FW-1:0] wr, wi;
```

图 3-4 单个运算器模块接口

3.2 仿真测试

对于各模块编写 testbench，首先进行前仿真测试。

- 按位下标反转模块：

```
module bit_reverse_tb();
reg [`logN-1:0] din;
wire [`logN-1:0] dout;
bit_reverse inst(din, dout);
initial begin
    din = `logN'd0; #10;
    for (din = 1; din != 0; din = din + 1) #10;
end
endmodule
```

图 3-5 下标反转模块仿真测试

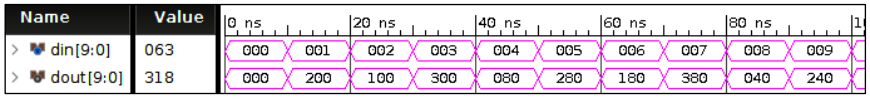

图 3-6 下标反转模块仿真测试波形图

测试使用$N = 10$，上述截取部分波形图正常。

- 内存模块：

```
module bram_dual_tb();
reg clk, wea, web;
reg [7:0] addra, addrb, dina, dinb;
wire [7:0] douta, doutb;
bram_data inst_data(
        .wea(wea), .addra(addra), .dina(dina), .douta(douta), .clka(clk),
        .web(web), .addrb(addrb), .dinb(dinb), .doutb(doutb), .clkb(clk));
initial begin
    clk = 0; wea = 1; web = 1; #10;
    for (addra = 0; addra != 8'd127; addra = addra + 1) begin
        addrb = 256 - addra; dina = addra; dinb = addrb; #20;
    end
    wea = 0; web = 0; #100;
    for (addra = 0; addra != 8'd127; addra = addra + 1) begin
        addrb = 256 - addra; #20;
    end
    $stop();
end
initial begin
    while (1) begin
        clk = ~clk; #5;
    end
end
endmodule
```

图 3-7 内存模块仿真测试

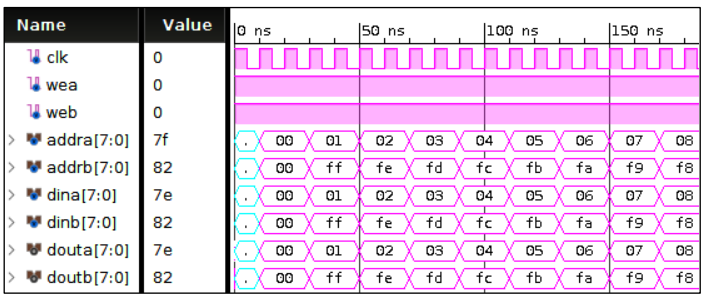

图 3-8 内存写入仿真波形图

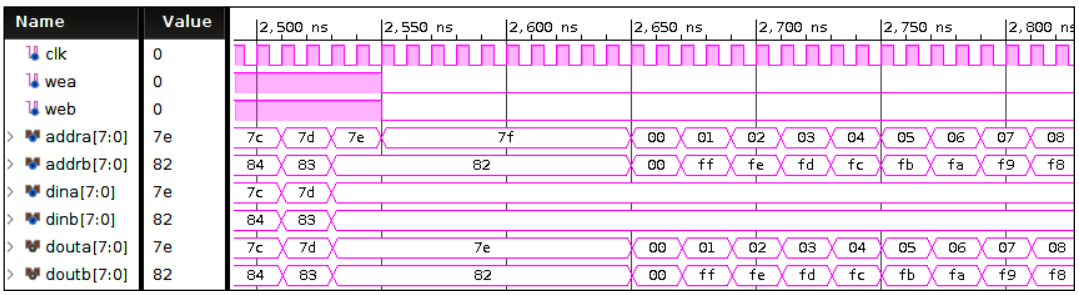

图 3-9 内存读取仿真波形图

BRAM 的大小为 256，可以看见两个端口都可以正常写入。

- 运算器模块：

```
module fft_op_unit_tb();
reg [63:0] ar, ai, br, bi, wr, wi;
wire [63:0] cr, ci, dr, di;
fft_op_unit op(ar, ai, br, bi, cr, ci, dr, di, wr, wi);

integer fpi, fpo, n, i;
initial begin
    fpi = $fopen("C:\\Users\\JXY\\Desktop\\project\\input.txt", "r");
    fpo = $fopen("C:\\Users\\JXY\\Desktop\\project\\output.txt", "w");
    i = $fscanf(fpi, "%d", n);
    $display("%d\n", n);
    for (i = 0; i < n; i = i + 1) begin
        $fscanf(fpi, "%x%x%x%x%x%x", ar, ai, br, bi, wr, wi); #10;
        $fdisplay(fpo, "%x %x %x %x", cr, ci, dr, di);
    end
end
endmodule
```

图 3-10 运算器模块仿真测试

```
100
3FDE74D7B1078A9C BFD83A55410AC450 3FC9B44E52480348 3FE2070AD2DCF4C6 BFE8DCA42B43ADBA 3FC44EEC9F8A4F30
3FE7B4297DA794A6 3FD84A9690D19360 BFE0732CE7990114 BFD421AD604DD008 3FB743E246955030 BFEBACE395B2FD98
BFC6ECDA8A996628 BFE0CC9CFF1E3D46 BF96950834603A00 3FE3969850C9E2E4 BFCF455B4CB631E0 3FA268DCDACED7C0
BFE9F21BB1BF8A22 3FEA2E9D9A8F6A34 BFE2B62E2B48789E BFCE3104F66E7D80 3FD4841050917E14 3FE089302CDC44BA
BFE4EC6E4575A1F6 3FA1CC000D20B760 3FEFB39EA033919C 3FDA92F1EDD31470 BFEAD7FC3F59E70A BFED3A70AD9AAECC
BF921D083DDF8140 BFBB57B61DED2FA0 BF9B095A7FA2E500 BFE5620A33FC96F4 BFD1D601F55E7764 3FE85DBF902FAA3E
3FDF46EEA6E51E24 BFC54E7E0E61A3F8 3FEA1219E711406C BFE9F6E04D573FF0 BFE4652689C159DA 3FEC94E29D426EC0
BFE98BB065BCF124 BFCCA9897BDB9A50 BFDAFAAE0234CF28 BFEB528B7CC6D9A4 BFE38B8ADD46E874 BFC51FB019D93088
BFDA8161EC2C6970 3FD9DFACBCB3E844 BFE0A88A0F163E74 3FED6829CB415C40 BFD8E69D5376A104 BFE615DD79551DDE
3FBC6B997B4C5B70 3FE298448EB2F568 BFBCBCC54EE12060 3FEFBB7E72A41A5A BFC03BDAA985F078 BFD9081257432984
BFE0392CB0E8BEF2 3FED7E293241BF8E BFE1BC518126B4F0 BFCAB95B4F9AEF40 BFE1A163C491758E BFDD6FD4E6BB8C70
BFC4E104EC64DDC8 3FEFDAE802CACFFA 3FEA4E5277A2D704 3FB9D18D9BBFB2F0 3FC8A976CA316AA0 BFEAEF48CE63D5B8
3FC39E6B5A4995A0 3FE97BA7B885D130 BFB2C8F51375B600 BFCA02B117BC9DD8 BFE95082540F2E68 3FD37C7B63516F30
3FEF78034C9BF39C 3FD6CB170C74B9F0 BFC24F4ADC4619D8 3FD3D0879AC5E284 3FC6B806E7065E08 3FDF5DF5949E8604
3FD20809A0556820 3F7E1D85710EC400 3FEC08DE1D37EE18 3FCAE7CC019CE168 3FD1C840DD88E754 3FD9E0AD9B346EE4
```

图 3-11 输入随机文件

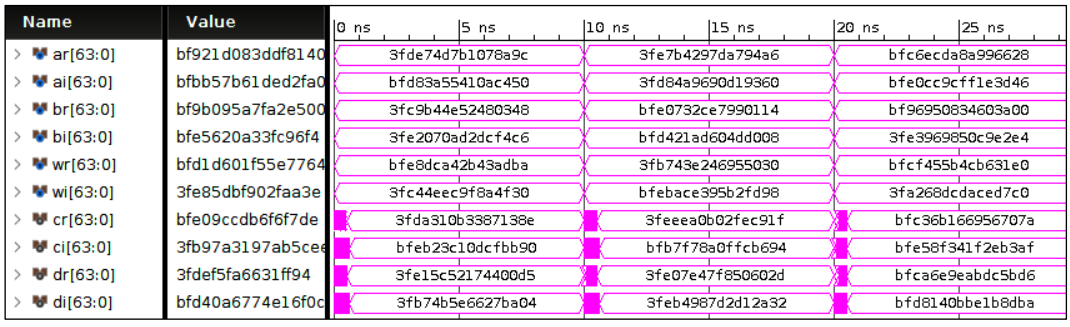

图 3-12 运算器模块仿真测试波形图

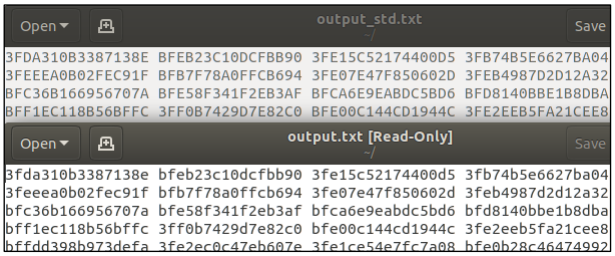

图 3-13 输出文件与标准文件比对

随机生成(−1,1)间的一系列输入的浮点数，并计算正确的输出值。经过比对，模块可以正常工作。

- 控制器模块：

```
module fft_tb();
reg clk, rst, sig, we, rev;
reg [31:0] addr;
reg [127:0] din;
wire [127:0] dout;
fft inst(clk, rst, sig, we, rev, addr, din, dout);
wire clk_run, busy;
wire [16:0] rnd, iter, cycle;
assign clk_run = inst.clk_run;
assign busy = inst.busy;
assign rnd = inst.rnd;
assign iter = inst.iter;
assign cycle = inst.cycle;
initial begin
    clk = 0; sig = 0; addr = 64'd0; rst = 0; #10; rst = 1; #5; rst = 0;
    din = 0; we = 0; rev = 0;
    for (addr = 0; addr < 1024; addr = addr + 1) #20;
    #2000;
    #100;
    sig = 1; #10; sig = 0;
    #100000;
    for (addr = 0; addr < 1024; addr = addr + 1) #20;
    #20;
    $stop();
end
initial begin
    while (1) begin clk = ~clk; #5; end
end
endmodule
```

图 3-14 控制器模块仿真测试

```
memory_initialization_radix=16;
memory_initialization_vector=
0000000000000000
3FE3FED9534556D4
3FEF38F3AC64E589
3FECC1F0F3FCFC5D
```

图 3-15 初始化 COE 文件前 4 行

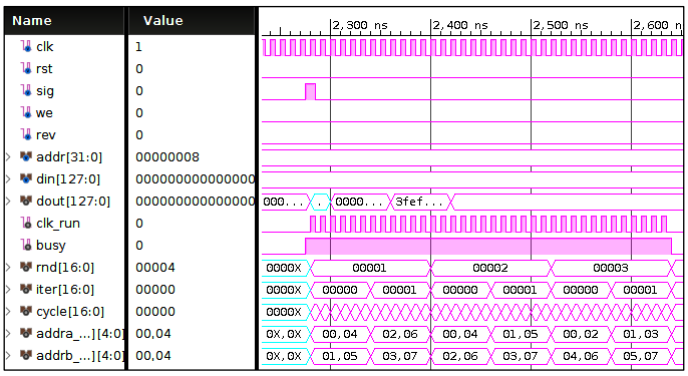

图 3-16 地址计算仿真波形图

为方便测试，测试中取较小的$N=8$，$M=2$。内存中使用 COE 文件预先存储一些$(-1,1)$上的数值。两块内存中为上图中 COE 文件中的前 4 个数，转换为浮点数分别为 0、0.6249、0.9757、0.8987。图 3-16 中最后两行为组合逻辑计算出的两个计算单元对应的地址，可以看见在仿真中是正确的。

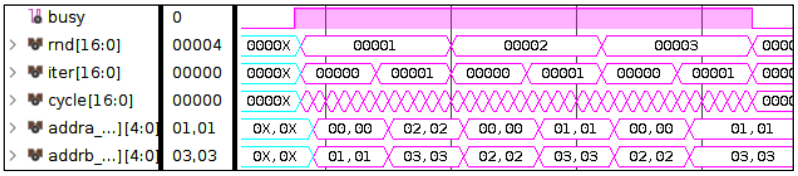

图 3-17 块内地址计算仿真波形图

上图是对于每一个块内两个端口的计算，当$r \le n-m$时，上一步一次运算的两个地址都在一个块内，可以分别赋给端口 A 和端口 B。当$r > n-m$时，则需要根据地址的第$r-1$位分别讨论。实际上只要端口统一并且能够一一对应，就是不会出现地址冲突问题的，上面的两类情况也可以通过地址的前k位是否与k相等判断。如果运算单元的两个地址中第一个地址与k相等，则将其赋给内存对应的第一个地址，如果与k不等，则交叉赋给内存单元上的另一个端口即第二个。对于运算单元的第二个地址同样如此，不等则赋给内存单元的另一个地址。

- 综合测试：

图 3-18 综合随机数据测试$N=8, M=2$

将控制器的时序与运算器结合，可以使用 COE 文件对 RAM 初始化后进行综合测试。$N=8, M=2$的测试如图 3-18，经过单步计算验证是正确的。

- FFT 模块较大规模数据测试：

```
initial begin
    fpi = $fopen("test4096.txt", "r");
    fpo = $fopen("output.txt", "w");
    clk = 0; sig = 0; addr = 64'd0; rst = 0; #10; rst = 1; #5; rst = 0;
    din = 0; we = 1; rev = 1;
    for (addr = 0; addr < 4096; addr = addr + 1) begin
        $fscanf(fpi, "%x", din); #20;
    end
    we = 0; rev = 0;
    for (addr = 0; addr < 4096; addr = addr + 1) #20;
    #2000;
    #100;
    sig = 1; #10; sig = 0;
    for (i = 0; i < 1000000; i = i + 1) #10;
    for (addr = 0; addr < 4096; addr = addr + 1) begin
        #20; $fdisplay(fpo, "%h %h", `r(dout), `i(dout)); #10;
    end
    $fclose(fpo); #200;
    $stop();
end
```

对于N较大的情形，可以使用文件读入进行仿真，以采样率$f_s=8192\text{Hz}$，生成时长$t=0.5\text{s}$，频率$f=440\text{Hz}$与$f=880\text{Hz}$的正弦信号混合。这样傅里叶变换的结果横坐标频率范围为$(0, f_s)=(0,8192\text{Hz})$，分辨率为$f_s/N=2\text{Hz}$。

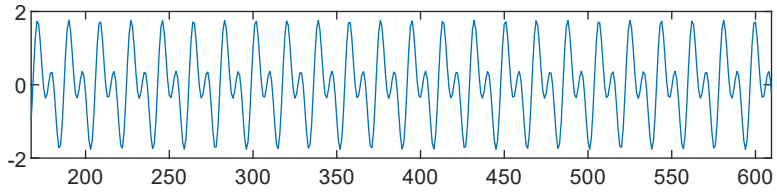

图 3-19 生成周期性数据

```
219    0.000000 + 0.000000 i
220    0.000000 + -0.000000 i
221    4096.000000 + 0.000000 i
222    -0.000000 + -0.000000 i
```

```
439    0.000000 + -0.000000 i
440    0.000000 + -0.000000 i
441    4096.000000 + 0.000000 i
442    -0.000000 + -0.000000 i
```

图 3-20 $N=4096, M=8$周期性数据仿真结果

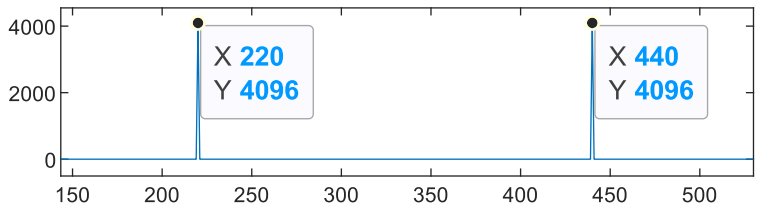

图 3-21 标准计算结果（MATLAB）

可以看见，仿真结果可以在下标为 220 与 440 的位置计算出正确的结果。

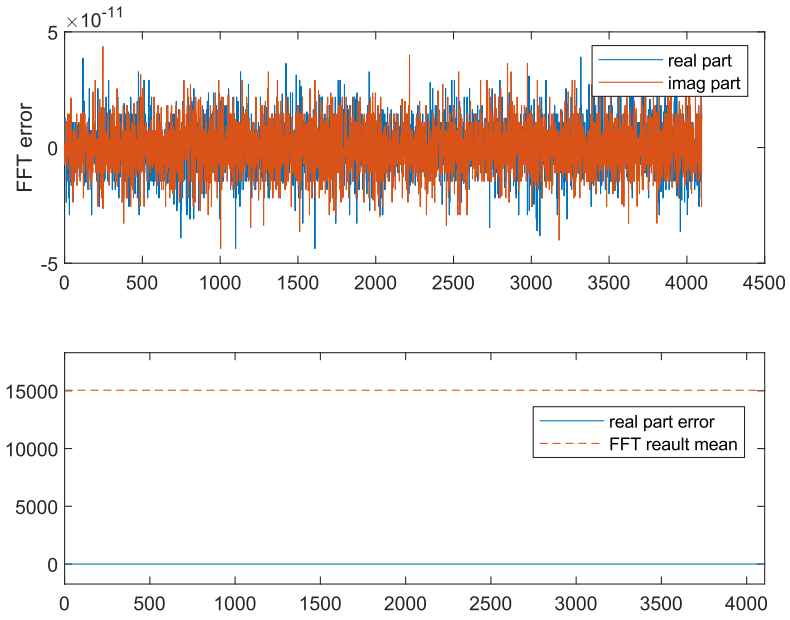

图 3-22 $N = 4096, M = 8$随机数据仿真结果误差

生成$(-1024,1024)$间的随机数据进行$N = 4096$的傅里叶变换，与 MATLAB 中的 FFT 计算结果比较，可以得到如上图的误差。虽然会因为浮点数运算阶段截断尾数等操作有所区别产生一定的误差，但是这个误差十分微小，相差十几个数量级可以忽略。因此，上述规则的数据与不规则的数据，此模块都可以正常仿真运行，正确性得到了验证。

参考文献

[1] Fourier transform, https://en.wikipedia.org/wiki/Fourier_transform.

[2] Fast Fourier transform, https://en.wikipedia.org/wiki/Fast_Fourier_transform.