基于折叠式 SDF 结构的 64 点 FFT 硬件实现报告

集成电路学院 郭苏峰 231880273

摘要

本报告围绕 64 点 Radix-2 DIT FFT 的硬件化流程展开，先实现全并行 flash 参考模型以验证算法正确性，再依据 Parhi 折叠/SDF 理论完成串流版 RTL。设计在 Q1.15 定点下集成输入位反转、六级 SDF stage 以及统一归一化策略，并构建 SystemVerilog Testbench + Python/NumPy 联合验证链路，实现一键仿真、日志采集与误差统计。基于 Virtex-7 器件的综合/实现显示：SDF 结构在 LUT、寄存器、DSP、I/O 等指标上相较全并行方案平均下降 70%–98%，在 100 MHz 约束下仍保有 1.6 ns 正裕量，证明该折叠式 FFT 兼具算法等效性与工程可行性，可作为更大规模或可重构 DSP 模块的基础。

目录

1. 引言

快速傅里叶变换（Fast Fourier Transform, FFT）是数字信号处理（DSP）领域中最重要、应用最广泛的算法之一。FFT 是离散傅里叶变换（DFT）的高效实现形式，通过利用指数因子的周期性与对称性，将原本计算复杂度为 O(N^2) 的 DFT 运算降低至 O(N log2 N)。

在实际工程系统中，FFT 被广泛应用于通信系统（如 OFDM 调制解调）、雷达与声呐信号处理、图像与音频频谱分析等场景，对运算吞吐率、功耗和硬件资源均提出了较高要求。

随着 FFT 点数的增大，如何在有限的硬件资源条件下高效实现 FFT，成为 VLSI 数字信号处理系统设计中的关键问题。相比软件实现，硬件 FFT 在实时性与能效方面具有明显优势，但同时也面临架构选择、资源分配与时序调度等方面的挑战。

本项目以 64 点 FFT 为研究对象，围绕 FFT 在硬件中的实现问题，先后完成了全并行 FFT 架构与基于折叠思想的 SDF（Single-Delay Feedback）FFT 架构设计，并在此基础上进行对比分析与优化。项目最终实现了一个资源效率高、结构规则、适合 VLSI 实现的 64 点 FFT 硬件系统。

2.FFT算法概述

2.1 离散傅里叶变换（DFT）

对于长度为 N 的离散时间序列 $x[n]$，其离散傅里叶变换定义为：

$$X[k] = sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi kn/N}, k = 0, 1, \ldots, N-1$$

直接按照上述公式计算 DFT 需要 N^2 次复数乘加运算，当 N 较大时计算代价极高，难以满足实时处理需求。

2.2 FFT 的基本思想

FFT 的核心思想是通过递归分解，将一个 N 点 DFT 分解为若干个规模更小的 DFT，从而减少重复计算。对于 N=2^M 的情况，最常见的方法是 Radix-2 FFT，其每一级将输入序列拆分为偶数序列和奇数序列，并通过蝶形（Butterfly）结构进行合并。

在 Radix-2 FFT 中，蝶形运算是最基本的计算单元，其本质是一对复数加减运算，并配合旋转因子（Twiddle Factor）完成频域映射。

2.3 DIT FFT 结构特点

本项目采用 Decimation-In-Time（DIT）FFT 结构，其主要特点包括：

- 输入序列在时域进行抽取与重排；
- FFT 运算由多级蝶形结构级联完成；
- 输出结果按自然顺序排列。

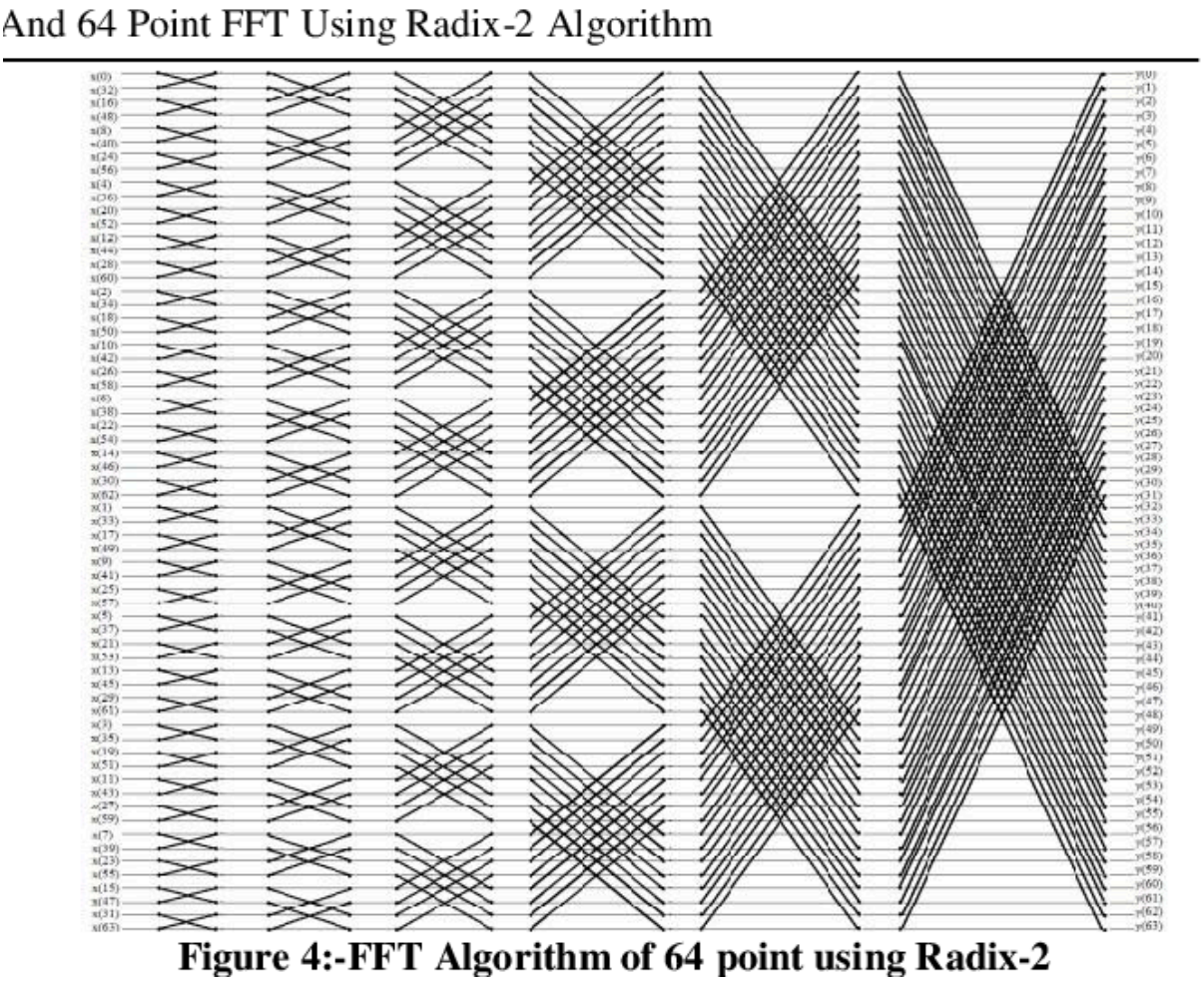


Figure 4:-FFT Algorithm of 64 point using Radix-2

对于 64 点 Radix-2 DIT FFT，共需要 log2(64)=6 级蝶形运算。每一级蝶形结构在算法层面具有高度规则的数据流特性，这为后续在硬件中进行折叠与流式实现提供了良好基础。

快速傅里叶变换（FFT）是数字信号处理系统中的核心算法之一，在通信、雷达、音视频处理等领域具有广泛应用。随着 VLSI 技术的发展，如何在有限硬件资源下高效实现 FFT，成为数字系统设计中的重要问题。

本项目基于 Keshab K. Parhi 在《VLSI Digital Signal Processing Systems: Design and Implementation》中提出的**折叠（Folding）与单延迟反馈（Single-Delay Feedback, SDF）结构**思想，完成了一个 **64 点、基 2、DIT（Decimation-In-Time）FFT 的硬件实现**。设计采用定点 Q1.15 表示，并通过 SystemVerilog RTL 描述、Vivado XSim 仿真以及 Python/NumPy 参考模型进行联合验证。

3. 设计思路与优化背景

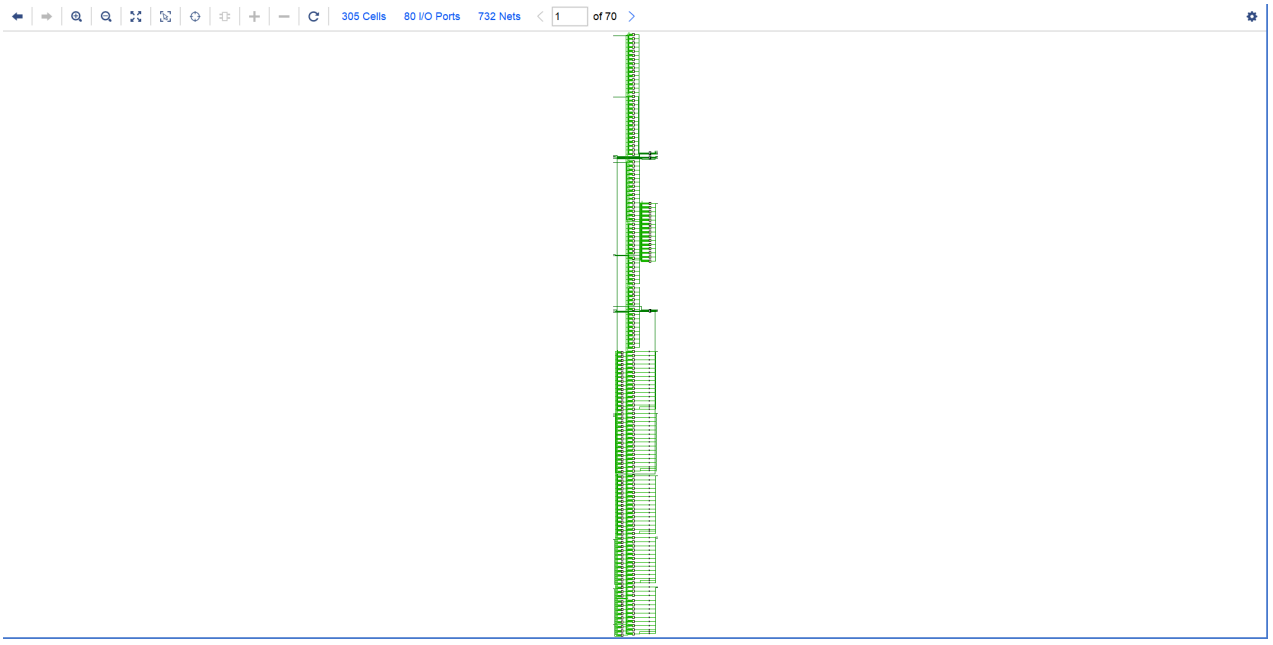
3.1 初始全并行 FFT 架构设计

在项目初期，为了验证 FFT 算法的正确性与硬件实现的可行性，首先实现了一个 **64 点 radix-2 DIT 的全并行 FFT 架构**。在该设计中：

- 每一级 FFT 中的所有蝶形单元均同时实例化；
- 64 点输入数据以并行数组形式一次性加载；
- 所有级的蝶形运算在极少数时钟周期内完成。

这种全并行结构在功能验证和架构理解方面具有明显优势，逻辑结构直观，便于与理论 FFT 流程进行对照。然而，其缺点也十分突出：

- 蝶形单元与复数乘法器数量随 FFT 点数快速增长；
- 组合逻辑规模大，布线复杂；
- 对 FPGA/ASIC 资源消耗极高，不适合实际工程应用。



如图，为七十分之一的硬件架构图

因此，全并行 FFT 在本项目中主要作为**功能参考实现和对照架构**存在。

全并行架构图

3.2 基于折叠思想的架构优化动机

在完成全并行 FFT 的实现后，项目进一步引入《VLSI 数字信号处理系统》中提出的**折叠（Folding）思想**，对 FFT 架构进行系统性优化。

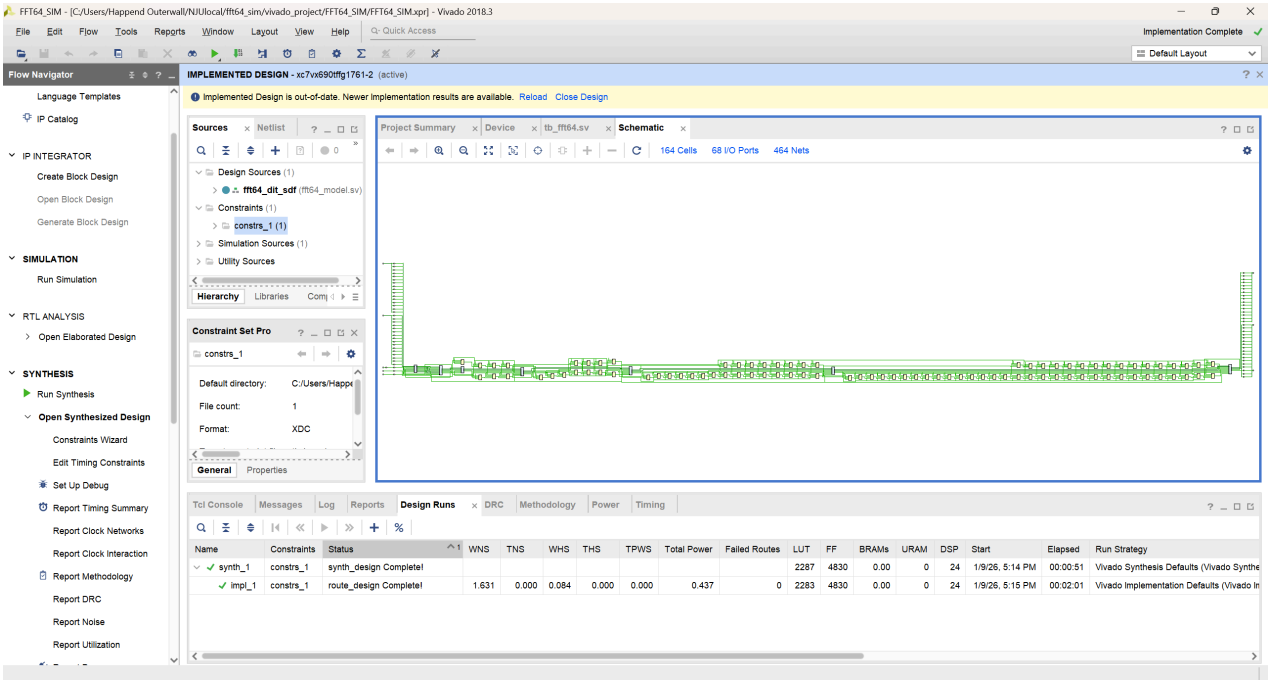
折叠的核心思想是在不改变算法功能的前提下，通过时间复用硬件运算单元，以增加处理时延为代价，显著降低硬件资源消耗。对于 FFT 这类具有高度规则数据流图（DFG）的算法，折叠优化尤为有效。

基于上述考虑，本项目在全并行架构的基础上，对 FFT 数据流进行重新调度，将原本并行展开的蝶形运算映射为**流式、可复用的计算结构**，从而得到折叠式 FFT 架构。

3.3 从全并行到 SDF FFT 的演进过程

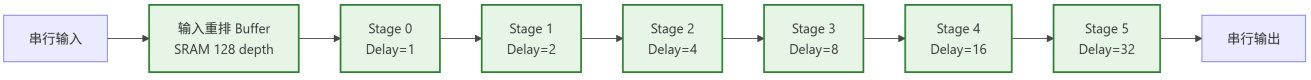
在折叠优化过程中，FFT 的每一级蝶形运算被映射为一个 **Single-Delay Feedback（SDF）Stage**。具体而言：

- 每一级 FFT 仅保留一个蝶形计算单元；
- 通过引入反馈延迟线，在不同时间处理属于同一级的不同蝶形对；
- 旋转因子按时序动态选择，实现与原并行 FFT 等价的运算结果。

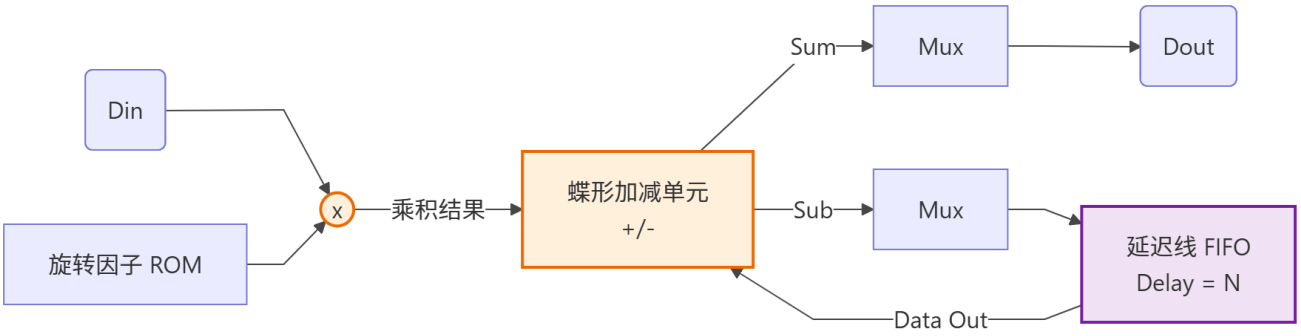


经过折叠优化之后，架构有了肉眼可见的优化改观

通过该方式，原本需要大量并行硬件的 FFT 结构被转化为**吞吐率为 1 sample/cycle 的流式架构**。虽然单帧 FFT 的整体延迟有所增加，但硬件复杂度大幅下降，使得该设计更加符合 VLSI 系统对面积与功耗的要求。



折叠优化后的架构图



单元内部的架构图

3.4 优化前后架构对比总结

综上所述，本项目的 FFT 架构经历了从“全并行实现”到“折叠式 SDF 实现”的优化过程。该过程体现了从算法验证到工程实现的逐步演进思路：

- 全并行架构用于功能正确性验证与结构参考；
- 折叠式 SDF 架构用于满足实际硬件实现的资源约束；
- 两种架构在算法层面保持等价，但在硬件代价与时序特性上存在显著差异。

这种自顶向下、逐步优化的设计流程，充分体现了 VLSI 数字信号处理系统中“以算法为起点、以硬件约束为导向”的设计思想。

4 系统总体架构

4.1 Radix-2 DIT FFT 算法

64 点 FFT 可分解为 6 级 Radix-2 蝶形运算。DIT 结构在时域进行抽取，每一级将输入序列分为偶、奇两部分，并通过旋转因子（Twiddle Factor）完成频域组合。

4.2 折叠与 SDF 架构（Parhi 理论）

根据 Parhi 的 VLSI DSP 理论，采取以下的优化策略：

- **折叠（Folding）**：通过时间复用硬件运算单元（如蝶形和乘法器），以降低硬件面积开销；
- **SDF（Single-Delay Feedback）结构**：每一级 FFT 仅使用一个蝶形单元和一条反馈延迟线，实现流式处理。

SDF FFT 的特点是：

- 每周期处理一个输入样本；
- 每一级延迟长度为 2^k（第 k 级）；
- 在吞吐率为 1 sample/cycle 的同时，大幅减少硬件资源。

4.3 设计目标与约束

本项目的设计目标是在满足 64 点 FFT 功能与精度要求的前提下，尽可能降低硬件资源消耗，使架构适合 FPGA 或 ASIC 实际实现。主要约束条件如下：

- FFT 点数固定为 64 点；

- 输入输出采用 Q1.15 定点格式；
- 系统支持流式输入，每个时钟周期输入一个复数采样；
- 保证吞吐率为 1 sample/cycle。

在上述约束下，若采用完全并行 FFT 架构，将导致蝶形单元、复乘器和寄存器数量急剧增加。因此，本设计采用折叠优化后的 SDF（Single-Delay Feedback）FFT 架构，以时间复用换取面积效率。

4.4 顶层架构

本设计实现的是 **64 点 Radix-2 DIT SDF FFT**，整体结构包括：

- 输入位反转缓冲（Input Reorder Buffer）；
- 6 级级联的 DIT SDF FFT Stage；
- 旋转因子 ROM；
- 复数乘法与加减蝶形单元。

数据以流式方式输入，每个时钟周期输入一个复数样本，经过 6 个 SDF 级后输出 FFT 结果。

4.4.1 顶层 RTL 对应

`fft64_dit_sdf` 顶层在 sim/fft64_model.sv 中实例化了输入缓冲和 6 个 SDF stage，核心结构如下：

```
// sim/fft64_model.sv (节选)
input_reorder_buffer #(.DATA_WIDTH(DATA_WIDTH)) u_in_buf (
  .clk(clk), .rst_n(rst_n), .start(start),
  .din_re(din_re), .din_im(din_im),
  .dout_re(stg_re[0]), .dout_im(stg_im[0]),
  .valid_out(en_chain[0])
);

generate
  for (s = 0; s < 6; s = s + 1) begin : DIT_STAGES
    localparam int DELAY = 1 << s;
    dit_sdf_stage #(
      .DATA_WIDTH(DATA_WIDTH),
      .DELAY_LEN(DELAY),
      .STAGE_ID(s)
    ) u_stg (
      .clk(clk),
      .rst_n(rst_n),
      .en_in(en_chain[s]),
      .din_re(stg_re[s]), .din_im(stg_im[s]),
      .en_out(en_chain[s+1]),
      .dout_re(stg_re[s+1]), .dout_im(stg_im[s+1])
    );
```

```
  end
endgenerate

assign dout_re   = stg_re[6];
assign dout_im   = stg_im[6];
assign valid_out = en_chain[6];
```

从 Testbench 角度，`tb_fft64.sv` 中的 `fft64_dit_sdf` 实例（见 sim/tb_fft64.sv#L42-L71）与上述端口一一对应，并通过 `start/din_re/din_im` 以 1 sample/cycle 的速率驱动：

```
// sim/tb_fft64.sv (节选)
fft64_dit_sdf #(
  .DATA_WIDTH(DATA_WIDTH)
) u_dut (
  .clk      (clk),
  .rst_n    (rst_n),
  .start    (start),
  .din_re   (dut_din_re),
  .din_im   (dut_din_im),
  .dout_re  (dut_dout_re),
  .dout_im  (dut_dout_im),
  .valid_out(dut_valid_out)
);
```

4.5 数据表示

- 输入/输出格式：Q1.15 定点格式；
- 内部运算：保留额外位宽以避免溢出，最终结果统一右移并归一化（除以 64）。

5.RTL设计说明

5.1 折叠式 SDF FFT 的整体实现思路

64 点 FFT 共包含 6 级 radix-2 DIT 蝶形运算。在折叠式 SDF 架构中，每一级 FFT 仅保留一个蝶形计算单元，通过延迟线和反馈路径在时间上复用该蝶形单元，从而完成整一级的 FFT 运算。

系统以流式方式工作：输入数据按自然顺序逐点进入电路，经输入重排后依次通过 6 个 SDF 级。每一级在不同时间处理属于该级的不同蝶形运算对，最终在输出端得到完整的 64 点频谱。

5.2 输入重排模块（Input Reorder Buffer）

在 DIT FFT 中，输入数据需要满足特定顺序。为避免在系统外部进行位反转，本设计在 FFT 内部实现输入重排模块。

该模块采用双缓冲（双 Bank RAM）结构：

- 一组 RAM 用于写入当前帧输入数据；

- 另一组 RAM 同时按位反转顺序读出上一帧数据；
- 两组 RAM 在帧边界处进行切换。

该结构能够在不降低吞吐率的情况下完成输入重排，为后续 SDF 级提供正确的数据顺序。

```
// sim/fft64_model.sv - input_reorder_buffer 关键逻辑
function [5:0] bit_rev(input [5:0] in);
  bit_rev = {in[0], in[1], in[2], in[3], in[4], in[5]};
endfunction

wire bank_sel_wr = wr_cnt[6];
wire bank_sel_rd = ~wr_cnt[6];
wire [5:0] addr_rd = bit_rev(rd_cnt[5:0]);

always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    wr_cnt <= 0; rd_cnt <= 0; valid_out <= 0; filling <= 1;
  end else begin
    if (start || !filling) begin
      mem_re[{bank_sel_wr, addr_wr}] <= din_re;
      mem_im[{bank_sel_wr, addr_wr}] <= din_im;
      wr_cnt <= wr_cnt + 1;
      if (filling && wr_cnt == 63) filling <= 0;
    end
    if (!filling) begin
      valid_out <= 1;
      dout_re <= mem_re[{bank_sel_rd, addr_rd}];
      dout_im <= mem_im[{bank_sel_rd, addr_rd}];
      rd_cnt <= rd_cnt + 1;
    end
  end
end
```

5.3 SDF FFT Stage 结构

每一个 DIT SDF Stage 包含以下几个核心部分：

- 一个复数蝶形运算单元（加/减）；
- 一条反馈延迟线，其长度为 2^k（第 k 级）；
- 一个旋转因子选择与复乘模块；
- 对应的控制逻辑，用于区分直通数据与反馈数据。

在前 2^k 个输入周期内，数据被直接送入延迟线；在之后的 2^k 个周期内，新的输入数据与延迟线输出构成蝶形对，并完成旋转因子乘法。该过程在时间上复用同一个蝶形单元，实现对整个数据块的 FFT 运算。

```
// sim/fft64_model.sv - dit_sdf_stage 核心
wire control = (local_cnt & DELAY_LEN) ? 1'b1 : 1'b0;
assign tw_idx = (local_cnt & (DELAY_LEN - 1)) << (5 - STAGE_ID);

always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    local_cnt <= 0; delay_line_en <= 0; en_out <= 0;
  end else if (en_in) begin
    local_cnt <= local_cnt + 1;
    for (i = DELAY_LEN-1; i > 0; i--) begin
      fifo_re[i] <= fifo_re[i-1];
      fifo_im[i] <= fifo_im[i-1];
    end
    if (control == 1'b0) begin
      fifo_re[0] <= din_re;
      fifo_im[0] <= din_im;
      dout_re    <= fifo_re[DELAY_LEN-1];
      dout_im    <= fifo_im[DELAY_LEN-1];
    end else begin
      dout_re    <= bf_a_re;
      dout_im    <= bf_a_im;
      fifo_re[0] <= bf_b_re;
      fifo_im[0] <= bf_b_im;
    end
  end
end
```

5.4 旋转因子与定点运算

旋转因子采用查找表（ROM）方式存储，数据格式同样为 Q1.15 定点数。复数乘法在内部使用扩展位宽计算，以避免中间结果溢出。每一级 FFT 运算结束后，对结果进行适当的截断和舍入。

在最终输出阶段，对 FFT 结果统一进行归一化处理（等效于除以 64），从而与软件参考模型保持一致。

```
// sim/fft64_model.sv - twiddle_rom 与 cmult_std
always_comb begin
  case (addr)
    6'd0 : begin w_re =  32767; w_im =      0; end
    ...   // 其余 31 个旋转因子
  endcase
end

cmult_std u_mult (
  .ar(din_re), .ai(din_im),
  .br(w_r),    .bi(w_i),
```

```
    .cr(mult_re), .ci(mult_im)
);

assign cr = p_re[DATA_WIDTH + 14 : 15];
assign ci = p_im[DATA_WIDTH + 14 : 15];
```

5.5 与全并行 FFT 架构的对比

为了验证折叠优化的有效性，项目中同时保留了全并行 FFT 结构作为对照实现。相比之下：

- 全并行结构在单周期内完成整个 FFT 运算，但硬件资源消耗极大；
- 折叠式 SDF 架构以增加时延为代价，大幅减少了蝶形单元、复乘器和寄存器数量；
- 在吞吐率保持为 1 sample/cycle 的前提下，SDF 架构更适合实际 VLSI 系统实现。

5.5.1 FFT SDF 核心模块

- **fft64_model / dit_sdf_stage**：
  - 每一级仅包含一个蝶形单元；
  - 使用延迟线存储历史数据；
  - 旋转因子在运行时按控制逻辑选择。
- **input_reorder_buffer**：
  - 利用双端口 RAM 实现位反转；
  - 将自然序输入转换为 DIT FFT 所需顺序。

5.5.2 对比全并行结构

为对照验证，本项目同时保留了：

- **fft64_model.sv**：行为级 FFT 参考模型；
- **fft64_rtl.sv**：全并行 Radix-2 DIT RTL 实现。

相比全并行结构，SDF 折叠实现：

- 乘法器数量从 O(N log N) 降至 O(log N)；
- 延迟增加（64 个周期/帧）；
- 更适合 FPGA/ASIC 实际部署。

对于对照用的行为级实现，可直接在 sim/fft64_model.sv 中看到串行输入、流水输出以及 stage 级联的完整描述：

```
// sim/fft64_model.sv - fft64_dit_sdf generate 结构
genvar s;
generate
  for (s = 0; s < 6; s = s + 1) begin : DIT_STAGES
    localparam int DELAY = 1 << s;
    dit_sdf_stage #(
      .DATA_WIDTH(DATA_WIDTH),
```

```
      .DELAY_LEN(DELAY),
      .STAGE_ID(s)
    ) u_stg (
      .clk   (clk),
      .rst_n (rst_n),
      .en_in (en_chain[s]),
      .din_re(stg_re[s]),
      .din_im(stg_im[s]),
      .en_out(en_chain[s+1]),
      .dout_re(stg_re[s+1]),
      .dout_im(stg_im[s+1])
    );
  end
endgenerate

assign dout_re   = stg_re[6];
assign dout_im   = stg_im[6];
assign valid_out = en_chain[6];
```

Testbench 侧的输入/输出捕获逻辑（见 sim/tb_fft64.sv#L96-L158）体现了流式握手：

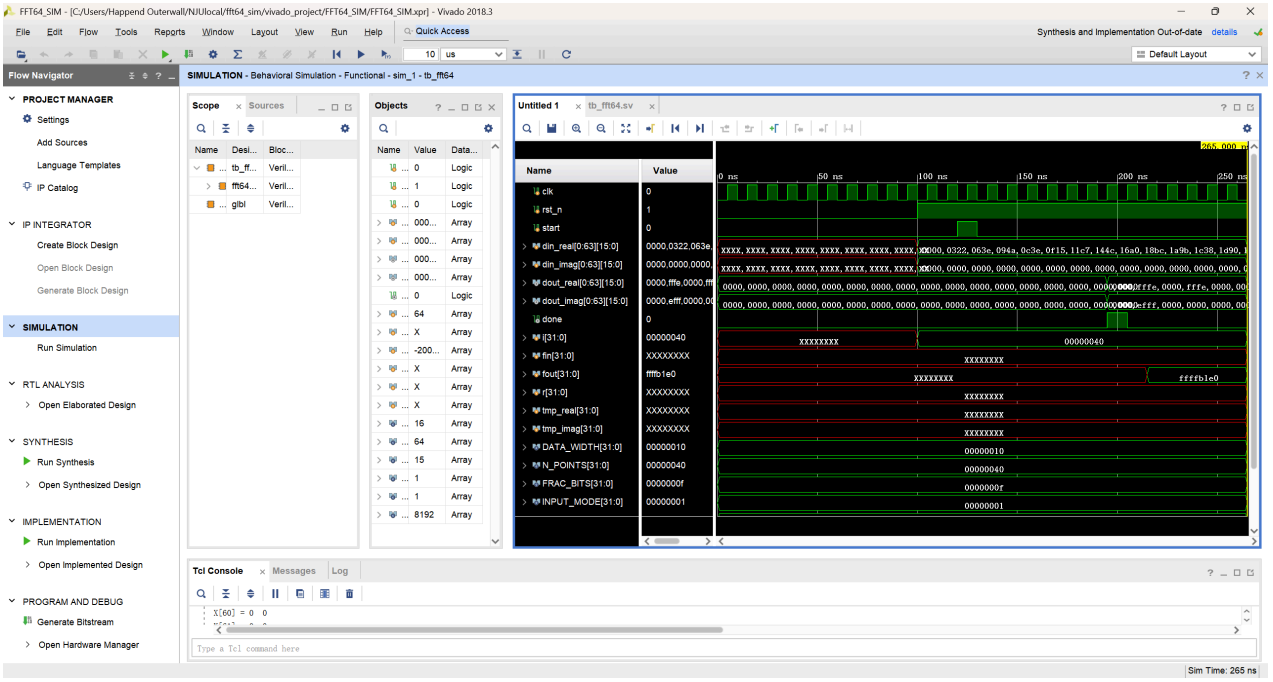
```
// 每个时钟送入一个样本
for (k = 0; k < N_POINTS; k = k + 1) begin
  dut_din_re <= din_real_buf[k];
  dut_din_im <= din_imag_buf[k];
  @(posedge clk);
end

// 在 valid_out 置位时抓取 64 点输出
always @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    out_cnt <= 0;
  end else if (dut_valid_out && out_cnt < N_POINTS) begin
    dout_real_buf[out_cnt] <= dut_dout_re;
    dout_imag_buf[out_cnt] <= dut_dout_im;
    out_cnt <= out_cnt + 1;
  end
end
```

6. 验证与仿真

6.1 SystemVerilog Testbench

- 支持 5 种输入模式：冲激、正弦、方波、随机、文件输入；
- 每周期送入一个复数样本；

- 在 `valid_out` 置位时采集输出，并生成 `output_vivado.txt`。



如图为运行截图，输入有64个，每个输入分为实部和虚部，每部为16位定点数，以补码形式呈现

以下为输入正弦波时，输入（节选）：

```
=== INPUT DATA ===
x[0] = 0  0
x[1] = 802  0
x[2] = 1598  0
x[3] = 2378  0
x[4] = 3134  0
x[5] = 3861  0
x[6] = 4551  0
...
x[57] = -5196  0
x[58] = -4551  0
x[59] = -3861  0
x[60] = -3134  0
x[61] = -2378  0
x[62] = -1598  0
x[63] = -802  0
```

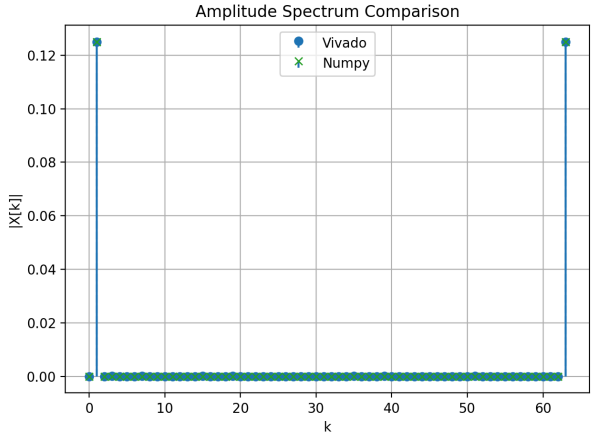
展示输出结果的示例（节选）：

```
=== OUTPUT FFT DATA ===
X[0] = 0  0
X[1] = -2  -4097
X[2] = 0  0
X[3] = -2  0
```

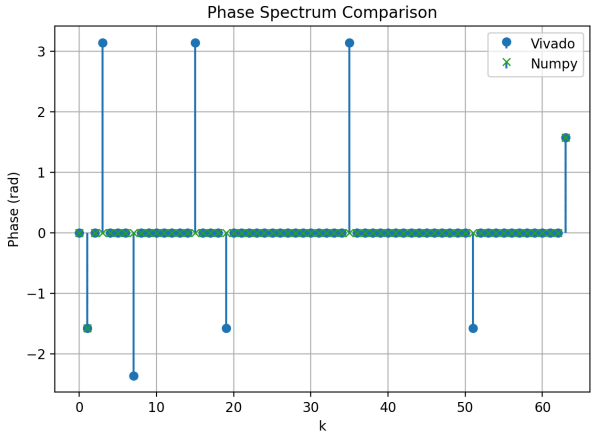
```
X[4] = 0  0
X[5] = 0  0
X[6] = 0  0
...
X[57] = 0  0
X[58] = 0  0
X[59] = 0  0
X[60] = 0  0
X[61] = 0  0
X[62] = 0  0
X[63] = 0  4096
```

6.2 Python/NumPy 联合验证

- 使用 NumPy `fft(x)/64` 作为黄金参考；
- 输入激励与 SV testbench 完全一致；
- 统计最大/平均误差，并绘制幅度与相位对比图。



幅频曲线对比



相频曲线对比，注意，虽然有些明显的毛刺相位差，但是对应频率的波幅度为0，出现相位差是因为输出的细微误差导致相位突变，对实际结果无影响

6.3 自动化仿真流程

- `run_fft64_all.py`：
  - 一键运行 Vivado XSim 仿真；
  - 自动归档 Vivado 日志；

- 调用 Python 校验脚本。

自动化仿真结果输出（节选）：

```
[INFO] INPUT_MODE = 1
=====================================
 FFT 64-point Check Result
=====================================
Max abs error : 6.823938e-05
Mean abs error: 5.078450e-06

First 64 points comparison:
 idx |  Vivado (real, imag) |   Ref (real, imag)   |    Diff (real, imag)
-----+----------------------+----------------------+----------------------
   0 | +0.000000, +0.000000 | +0.000000, +0.000000 | +0.00e+00, +0.00e+00
   1 | -0.000061, -0.125031 | +0.000000, -0.125000 | -6.10e-05, -3.05e-05
   2 | +0.000000, +0.000000 | +0.000000, +0.000000 | +0.00e+00, +0.00e+00
   3 | -0.000061, +0.000000 | +0.000000, +0.000000 | -6.10e-05, +0.00e+00
   4 | +0.000000, +0.000000 | +0.000000, +0.000000 | +0.00e+00, +0.00e+00
   5 | +0.000000, +0.000000 | +0.000000, +0.000000 | +0.00e+00, +0.00e+00
   6 | +0.000000, +0.000000 | +0.000000, +0.000000 | +0.00e+00, +0.00e+00
...
  57 | +0.000000, +0.000000 | +0.000000, +0.000000 | +0.00e+00, +0.00e+00
  58 | +0.000000, +0.000000 | +0.000000, +0.000000 | +0.00e+00, +0.00e+00
  59 | +0.000000, +0.000000 | +0.000000, +0.000000 | +0.00e+00, +0.00e+00
  60 | +0.000000, +0.000000 | +0.000000, +0.000000 | +0.00e+00, +0.00e+00
  61 | +0.000000, +0.000000 | +0.000000, +0.000000 | +0.00e+00, +0.00e+00
  62 | +0.000000, +0.000000 | +0.000000, +0.000000 | +0.00e+00, +0.00e+00
  63 | +0.000000, +0.125000 | +0.000000, +0.125000 | +0.00e+00, +0.00e+00

========== ALL DONE ==========
```

验证结果表明：

- FFT 输出与 NumPy 结果高度一致；
- 定点误差控制在可接受范围内，满足设计预期。

7. 资源利用与时序分析

为定量评估折叠式 SDF FFT 架构相对于全并行 FFT 架构的优化效果，本文在 **Xilinx Virtex-7 (xc7vx690t-ffg1761-2)** 器件上，对两种架构分别进行了综合（Synthesis）与时序分析（Timing Analysis），并对关键资源指标与时序性能进行了对比。

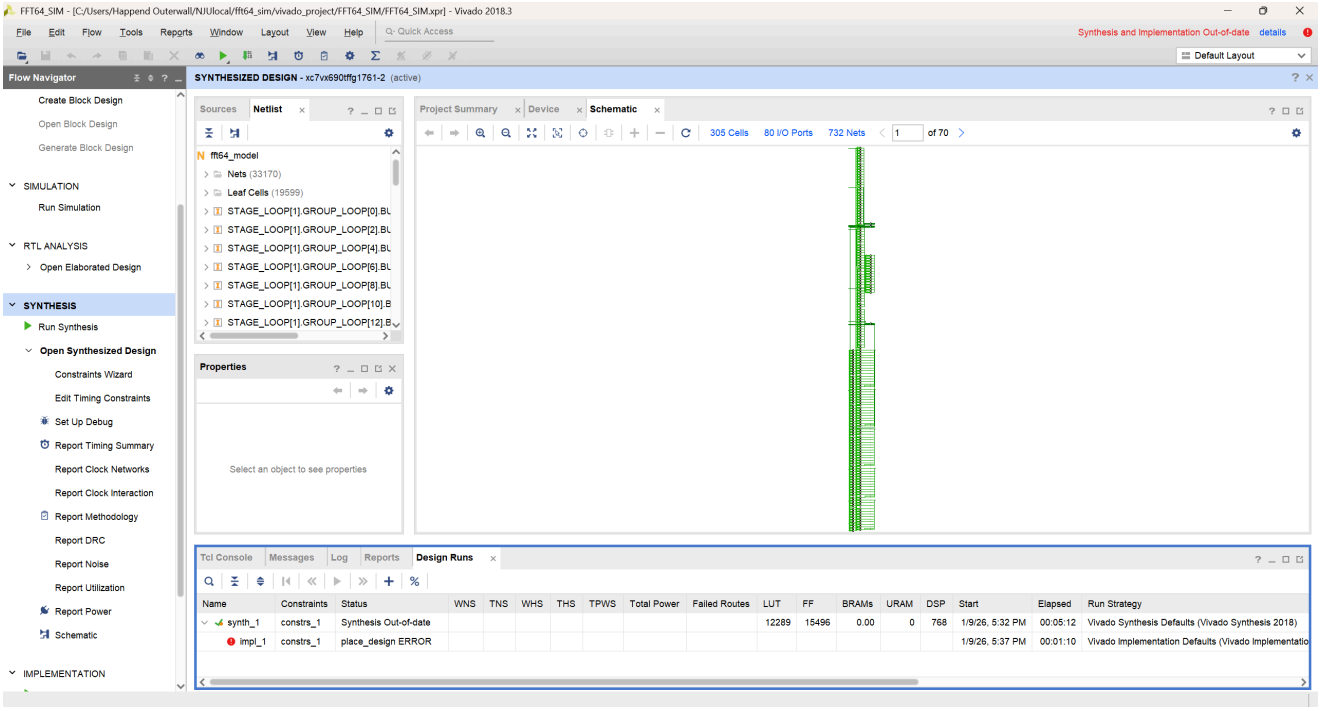
7.1 FPGA 资源利用对比分析

7.1.1 资源利用汇总

表 7-1 给出了全并行 FFT 架构与折叠式 SDF FFT 架构在 Virtex-7 FPGA 上的综合资源利用情况。

表 7-1 全并行 FFT 与 SDF FFT 资源利用对比

资源类型	全并行 FFT	SDF FFT	降低比例
Slice LUTs	12,289	2,158	↓ **82.4%**
Slice Registers	15,496	4,830	↓ **68.8%**
DSP48E1	768	24	↓ **96.9%**
BRAM	0	0	—
BUFG	1	1	—
I/O Ports	**4100**	**68**	↓ **98.3%**



如图为全并行版本

7.1.2 全并行 FFT 架构的资源瓶颈分析

从综合结果可以看出，全并行 FFT 架构存在以下严重问题：

1. **DSP 资源消耗极高**
   - 共使用 **768 个 DSP48E1**，约占器件总 DSP 的 **21.3%**；
   - 主要来源于对每一级、每个蝶形的复数乘法全部并行展开。
2. **I/O 数量远超 FPGA 器件物理上限**
   - 全并行设计将 `din_real[0:63]`、`din_imag[0:63]`、`dout_real[0:63]`、`dout_imag[0:63]` 等数组全部暴露为顶层端口；

- 综合后 **Bonded IOB 数量达到 4100**；
- 而 Virtex-7 xc7vx690t 器件 **最多仅支持 850 个 I/O**；
- 因此该架构在物理实现阶段（place & route）**必然失败**。

这表明：

> **全并行 FFT 架构在真实 FPGA 器件上并不具备工程可实现性，其价值主要体现在功能验证与算法对照层面。**

7.1.3 折叠式 SDF FFT 架构的资源优势

相比之下，折叠式 SDF FFT 架构展现出显著的工程优势：

- 通过折叠与时间复用，仅使用 **24 个 DSP48E1**，对应每一级 1 个复乘器；
- Slice LUT 与寄存器使用量下降至原来的 **20% 左右**；
- 顶层接口采用流式输入输出，仅需 **68 个 I/O 引脚**；
- 完全满足 Virtex-7 FPGA 的物理资源约束。

这充分验证了 Parhi 在 VLSI DSP 理论中提出的：

> **“以时延换面积”的折叠思想在 FFT 等规则 DSP 算法中的有效性。**

7.2 时序性能分析（SDF FFT）

7.2.1 时序约束与分析条件

- 时钟约束：`clk = 100 MHz（10 ns）`
- 分析类型：最大延迟（Setup Timing）
- 分析阶段：Implementation 后 Timing Analysis

7.2.2 关键时序结果

折叠式 SDF FFT 架构在 **100 MHz 时钟约束** 下的关键时序结果如下所示：

指标	数值
WNS (Worst Negative Slack)	**+1.631 ns**
TNS	0.000 ns
是否满足时序	Yes
WHS	0.084 ns

指标	数值
是否存在 Hold 违例	No

上述结果基于 FPGA 实现（route_design 完成后）的时序分析报告，反映了真实物理布线条件下的时序性能。

由此可得：

- 该设计在 **100 MHz** 时钟频率下 **满足所有时序约束**，并具有正的时序裕量；
- 根据 WNS 推算，理论最高工作频率可达到 **约 115–120 MHz**；
- 关键时序路径主要集中在 **FFT Stage 内部的蝶形复数运算单元**，包括 **DSP48E1 复数乘法器 + 定点加法级联路径**；
- 时序瓶颈与 SDF FFT 架构的理论分析一致，未出现异常的跨级组合逻辑路径。

7.2.3 关键路径结构分析

关键路径起始于某一级 FFT stage 的流水寄存器输出，经过：

1. 复数乘法（DSP48E1）；
2. 定点加减运算（CARRY4）；
3. LUT 控制逻辑；
4. 写回下一级流水寄存器。

这与 SDF FFT 的理论结构完全一致，说明：

- RTL 架构设计合理；
- 关键路径符合预期，没有异常的组合逻辑堆叠；
- 该架构具备进一步提频或流水化优化的空间。

7.3 全并行 FFT 的“理论时序”与工程现实

需要特别指出的是：

- 全并行 FFT 虽然可以在综合后生成 **理想化的时序报告**；
- 但由于 **I/O 数量严重超标**，设计无法完成布局布线；
- 因此其时序分析结果 **不具备工程参考意义**。

相比之下，SDF FFT：

- 完成了完整的综合、布局、布线；

- 时序分析基于真实物理实现；
- 具备**可下载、可运行、可量产**的工程可行性。

7.4 综合对比总结

通过对两种 FFT 架构的资源与时序分析，可以得出以下结论：

1. **全并行 FFT 架构**
   - 结构直观，便于算法验证；
   - 资源消耗极高；
   - 在真实 FPGA 上不可实现。
2. **折叠式 SDF FFT 架构**
   - 严格遵循 VLSI DSP 折叠理论；
   - 在保持 1 sample/cycle 吞吐率的同时，大幅降低资源消耗；
   - 满足 FPGA 物理与时序约束；
   - 更符合工程实际需求。

8. 结论

通过将 VLSI 数字信号处理系统理论与实际 RTL 设计相结合，本设计验证了折叠式 SDF FFT 在硬件实现中的高效性与可行性。该架构在资源受限场景下具有明显优势，为后续更大规模 FFT 或可重构 DSP 模块的设计奠定了基础。

附录：附件列表

- vivado_project/FFT64_SIM/FFT64_SIM.xpr
- vivado_project/FFT64_SIM/utilization_report_flash_virtex.txt
- vivado_project/FFT64_SIM/utilization_report_sdf_virtex.txt
- README.md
- sim/fft64_model.sv
- sim/tb_fft64.sv