به نام خدا

دانشگاه صنعتی شریف
دانشکده مهندسی برق

طراحی سیستم های مبتنی بر ASIC/FPGA

تمرین چهارم

امیرحسین یاری
۹۹۱۰۲۵۰۷

۱۷ اردیبهشت ۱۴۰۳

فهرست مطالب

Net Delay و Logic Delay

در ساختار FPGA، Net Delay و Logic Delay دو عامل مهم برای محاسبه بیشترین فرکانس کاری می‌باشند. بطور کلی، تفاوت اصلی بین Logic Delay و Net Delay در FPGAها در مرتبه و ماهیت تأخیرهایی است که توسط آنها نمایان می‌شوند.

Net Delay (تأخیر شبکه): Net Delay به تأخیری اشاره دارد که در انتقال اطلاعات از یک قسمت از FPGA به قسمت دیگر از آن، ایجاد می‌شود. این تأخیر از عوامل مهمی است که باید در نظر گرفته شود زیرا می‌تواند به طور مستقیم تأثیر بر عملکرد سیستم داشته باشد. اصطلاحاً، تأخیر شبکه از تأخیرهای متعددی تشکیل شده است که شامل تأخیرهای Routing و تأخیرهای سیم می‌شود. این تأخیرها ناشی از طول و مسیریابی مسیرهای سیگنالی درون FPGA هستند. در تأخیر شبکه، تأخیرهای مرتبط با انتقال اطلاعات از یک بلوک به بلوک دیگر شامل تأخیرهای مسیریابی، تأخیرهای سیم‌پیچی، تأخیرهای IO و غیره جمع‌آوری می‌شوند.

Logic Delay (تأخیر منطقی): Logic Delay معمولاً به میزان زمانی اشاره دارد که برای انجام عملیات منطقی (مثلاً جمع و یا ضرب) درون یک بلوک منطقی در FPGA لازم است. تأخیر منطقی از فرآیندهایی مانند تأخیر ترانزیستورها، تأخیرهای مسیریابی داخلی در بلوک، تأخیر مسیرهای مجتمع در بلوک و غیره ناشی می‌شود.

پایپ‌لاین کردن یک روش است که می‌تواند بهبود مهمی در کاهش تأخیر شبکه در FPGAها داشته باشد. این بهبود از طریق مدیریت و بهینه‌سازی فرآیند انتقال داده‌ها از یک بلوک به بلوک دیگر در FPGA انجام می‌شود. زیرا Net Delay مربوط به تأخیرهای مسیریابی و تأخیرهای سیم‌پیچی بین بلوک‌های FPGA است، پایپ‌لاین کردن می‌تواند بهبودهای زیر را ایجاد کند:

۱. تقسیم فرآیند انتقال داده: با پایپ‌لاین کردن، فرآیند انتقال داده به چندین مرحله تقسیم می‌شود، هرکدام از این مراحل با یک استیج مشخص متناظر است. این استیج‌ها معمولاً با استفاده از رجیسترها از یکدیگر جدا می‌شوند.

۲. کاهش تأخیر مسیریابی: با تقسیم فرآیند انتقال داده به مراحل کوچک‌تر، مسیریابی مسیرها ساده‌تر و بهینه‌تر می‌شود. به این ترتیب، تأخیرهای مرتبط با Routing کاهش می‌یابد.

۳. کاهش تأخیر سیم‌پیچی: با پایپ‌لاین کردن، طول مسیرهای سیم‌پیچی کاهش می‌یابد زیرا اطلاعات به صورت موازی از یک مرحله به مرحله بعدی منتقل می‌شوند. این کاهش طول مسیرهای سیم‌پیچی می‌تواند تأخیرهای مرتبط با سیم‌پیچی را نیز کاهش دهد.

۴. استفاده از منابع FPGA بهینه‌تر: با تقسیم فرآیند به چندین استیج، امکان استفاده بهینه‌تر از منابع FPGA وجود دارد. به عنوان مثال، می‌توان از منابع محلی (مثل رجیسترها و بلوک‌های منطقی) در هر استیج استفاده کرد تا تأخیر کلی را بهبود بخشید.

هفت برابر کردن ضرب اعداد

الف و ب

ماژول part1 یک ساختار pipeline برای ضرب دو عدد به عنوان ورودی و سپس ضرب نتیجه در عدد ۷ را پیاده‌سازی می‌کند. در ماژول، دو مرحله pipeline وجود دارد:

۱. مرحله اول: در این مرحله، عمل ضرب بین operand1 و operand2 انجام می‌شود و نتیجه آن در متغیر stage1_result ذخیره می‌شود.

۲. مرحله دوم: در این مرحله، نتیجه ضرب از مرحله اول با عدد ۷ ضرب می‌شود و نتیجه در متغیر stage2_result ذخیره می‌شود.

```
module part1(
    input wire clk,
    input wire reset,
    input wire [31:0] operand1,
    input wire [31:0] operand2,
    output reg [63:0] result
);

reg [63:0] stage1_result;
reg [63:0] stage2_result;

always @(posedge clk or posedge reset) begin
    if (reset) begin
        stage1_result <= 0;
        stage2_result <= 0;
        result <= 0;
    end
    else begin
        // Pipeline Stage 1: Multiply operands
        stage1_result <= operand1 * operand2;

        // Pipeline Stage 2: Multiply the result by 7
        stage2_result <= stage1_result * 7;

        // Output the result after latency
        result <= stage2_result;
    end
end

endmodule
```

سپس در ماژول part1_tb صحت عملکرد مدار را مورد بررسی قرار دادم و همانطور که مشاهده می‌کنید به درستی نتیجه می‌دهد.

```
a=        10, b=         5, result=       350
a=        20, b=         3, result=       420
a=         1, b=         4, result=        28
```

همانطور که در تصویر زیر مشاهده می‌کنید، با اعمال محدودیت زمانی، می‌توان گفت فرکانس کاری مدار برابر 80 MHz است. (دقت شود که در این حالت تمامی constraintها met شد.)

```
NET "clk" TNM_NET = clk;
TIMESPEC TS_clk = PERIOD "clk" 12.5 ns HIGH 50%;
```

$$\frac{1}{12.5 \times 10^{-9}} = 80MHz$$

منابع مصرفی مدار را در فایل Design Summary of Part1.pdf می‌توانید مشاهده کنید.

ج

ماژول part2 یک ساختار پایپ‌لاین شده برای ضرب دو عدد ۳۲ بیتی را پیاده‌سازی می‌کند. ورودی‌های این ماژول شامل سیگنال‌های clk و reset و دو عدد ۳۲ بیتی operand1 و operand2 به عنوان عددهایی که می‌خواهیم ضرب کنیم، می‌باشد. خروجی این ماژول یک عدد ۶۴ بیتی به نام result است که نتیجه ضرب را نمایش می‌دهد.

در مرحله اول پایپ‌لاین، عمل ضرب دو عدد ورودی انجام می‌شود و نتیجه آن در stage1_result ذخیره می‌شود. در مرحله دوم، نتیجه محاسبه‌شده از مرحله قبل ابتدا به وسیلهٔ انتقال به چپ ۳ بیتی هشت برابر می‌شود و سپس از نتیجه اصلی کم می‌شود. این کار باعث می‌شود نتیجه نهایی هفت برابر نتیجه مرحله اول باشد و مطابق با خواسته مسئله باشد.

```
module part2(
    input clk,
    input reset,
    input [31:0] operand1,
    input [31:0] operand2,
    output reg [63:0] result
);

reg [63:0] stage1_result;
reg [63:0] stage2_result;

always @(posedge clk or posedge reset) begin
    if (reset) begin
        stage1_result <= 0;
        stage2_result <= 0;
        result <= 0;
    end
    else begin
        // Pipeline Stage 1: Multiply operands
        stage1_result <= operand1 * operand2;

        // Pipeline Stage 2: Multiply the result by 7 using addition and shifting
        stage2_result <= (stage1_result << 3) - stage1_result;

        // Output the result after latency
        result <= stage2_result;
    end
end

endmodule
```

سپس در ماژول part2_tb صحت عملکرد مدار را مورد بررسی قرار دادم و همانطور که مشاهده می‌کنید به درستی نتیجه می‌دهد.

```
a=        10, b=         5, result=       350
a=        20, b=         3, result=       420
a=         1, b=         4, result=        28
```

همانطور که در تصویر زیر مشاهده می‌کنید، با اعمال محدودیت زمانی، می‌توان گفت فرکانس کاری مدار برابر 222 MHz است.(دقت شود که در این حالت تمامی constraintها met شد.)

```
NET "clk" TNM_NET = clk;
TIMESPEC TS_clk = PERIOD "clk" 4.5 ns HIGH 50%;
```

$$\frac{1}{4.5 \times 10^{-9}} = 222MHz$$

منابع مصرفی مدار را در فایل Design Summary of Part2.pdf می‌توانید مشاهده کنید.

با مقایسه منابع مصرفی مدارهای part1 و part2 مشاهده می‌کنیم که تعداد register ها و LUT ها تقریبا باهم برابر بوده اما تعداد DSP48 های استفاده شده در part1 چهارتا از part2 بیشتر بوده که مطابق انتظارمان است.

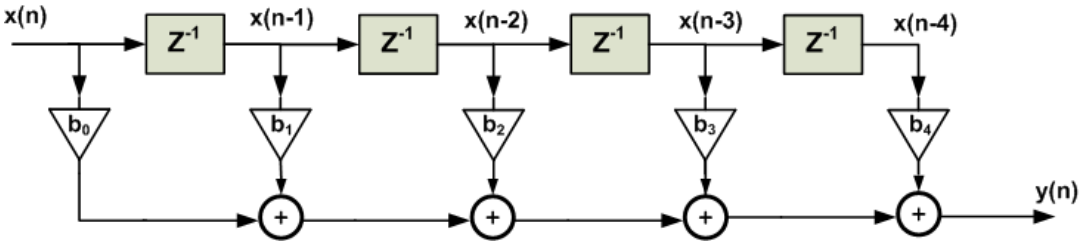
فیلتر FIR

الف

ساختارهای متداول فیلتر FIR را می‌توان به دو دسته اصلی تقسیم کرد: ساختار مستقیم (Direct) و ساختار Transpose. این دو ساختار در روش‌های مختلف محاسبه فیلتر FIR بکار می‌روند و هر کدام ویژگی‌ها و مزایای خاص خود را دارند.

ساختار مستقیم: در این ساختار، هر عضو فیلتر با وزن‌های مخصوص به خود به ترتیب از ورودی‌های فیلتر گذر می‌کند و خروجی تولید می‌کند. عملکرد این ساختار بسیار مستقیم و قابل فهم است. با این حال، این ساختار ممکن است در صورت داشتن تعداد زیادی وزن ورودی، حجم زیادی از محاسبات را مستلزم شود که ممکن است به مشکل محاسباتی منجر شود.

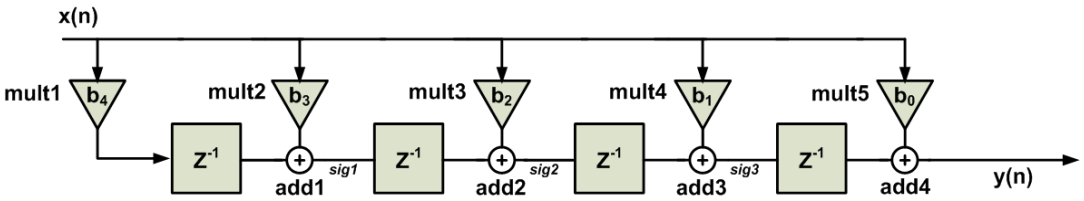
- در این ساختار، خروجی فیلتر با استفاده از ترکیب خطی مستقیم از ورودی‌ها و ضرایب فیلتر محاسبه می‌شود.
- این ساختار معمولاً مناسب برای فیلترهای با تعداد ضرایب کم و مقیاس کوچک است.
- این ساختار برای محاسبات ساده‌تر و مستقیم‌تر به نظر می‌رسد و عموماً مناسب برای سیستم‌های با محدودیت پردازشی است.



$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N]$$
$$= \sum_{i=0}^{N} b_i \cdot x[n-i],$$

ساختار transpose: این ساختار از اتصالات متقارن و وزن‌های متقارن برای کاهش تعداد عملیات محاسباتی استفاده می‌کند. با استفاده از این ساختار، می‌توان تعداد محاسبات را به شدت کاهش داد. این ساختار به خصوص مفید است زمانی که تعداد زیادی از وزن‌ها صفر هستند، چرا که با استفاده از وزن‌های غیر صفر، تعدادی از محاسبات اضافی را حذف می‌کند. اما این ساختار ممکن است پیچیدگی محاسباتی را افزایش دهد.

- در این ساختار، محاسبات به طور موازی انجام می‌شوند و خروجی‌های میانی در هر مرحله برای محاسبه‌های بعدی استفاده می‌شوند.
- این ساختار عموماً برای فیلترهای با تعداد زیادی ضریب و مقیاس بزرگ مناسب است.
- از آنجا که این ساختار بهینه‌سازی‌های موازی را فراهم می‌کند، برای سیستم‌های با پردازش موازی یا سیستم‌های با محدودیت زمانی مناسب است.



ب

ماژول FIR Filter یک فیلتر FIR با ۱۰ تپ است که برای پردازش سیگنال‌های ورودی به کار می‌رود. ورودی‌های این ماژول شامل سیگنال کلاک (Clk)، سیگنال ریست (Reset)، یک ورودی ۸ بیتی (InputData) که به صورت استریم به ماژول وارد می‌شود، یک ورودی ۴ بیتی (CoefficientIndex) که شماره ضریب مورد نظر را مشخص می‌کند، یک ورودی ۸ بیتی (NewCoefficientValue) که مقدار جدید برای ضریب را تعیین می‌کند، و یک ورودی تک بیتی (CoefficientWriteEnable) که فعال کردن نوشتن ضریب جدید را تعیین می‌کند. همچنین یک خروجی ۲۵ بیتی (FilteredOutput) وجود دارد که به صورت استریم از ماژول خارج می‌شود.

ماژول FIR Filter از طریق پارامتر numberOfTaps که برابر با ۱۰ است، تعداد تپ‌های فیلتر را مشخص می‌کند. این ماژول با استفاده از یک آرایه از ضرایب و یک بافر برای داده‌های ورودی عملکرد خود را انجام می‌دهد. در هر لبه کلاک، ضرایب به روزرسانی می‌شوند، داده‌های ورودی جدید وارد بوفر می‌شوند، و سپس عملیات فیلتر FIR با استفاده از ضرایب و داده‌های ورودی در هم ضرب و جمع انجام می‌شود تا خروجی تولید شود.

```
parameter numberOfTaps = 10;                  // Number of filter taps

integer tapIndex;                             // Loop index for taps
integer coefficientIndex;                     // Loop index for coefficients
reg signed[7:0] Coefficients[numberOfTaps - 1:0]; // Coefficient values
reg signed[7:0] InputFrame[numberOfTaps - 1:0]; // Input data buffer
reg signed[24:0] OutputResult;                // Intermediate result

always @(posedge Clk) begin
    if(!Reset) begin                          // Reset condition
        for(tapIndex = 0; tapIndex < numberOfTaps; tapIndex = tapIndex + 1) begin // Initialization
            InputFrame[tapIndex] <= 8'd0;     // Reset input buffer
            Coefficients[tapIndex] <= 8'd0;   // Reset coefficients
        end
        FilteredOutput <= 25'd0;              // Reset output
    end
    else begin
        if(CoefficientWriteEnable)            // If coefficient write enable is asserted
            Coefficients[CoefficientIndex] <= NewCoefficientValue; // Update coefficient
        InputFrame[0] <= InputData;           // New input
        for(tapIndex = 1; tapIndex < numberOfTaps; tapIndex = tapIndex + 1) begin
            InputFrame[tapIndex] <= InputFrame[tapIndex-1]; // Shift input buffer
        end
        FilteredOutput <= OutputResult;       // Update output
    end
end

always @(*) begin
    OutputResult = 0;                         // Reset intermediate result
    for(coefficientIndex = 0; coefficientIndex < numberOfTaps; coefficientIndex = coefficientIndex + 1) begin // FIR filter operation
        OutputResult = OutputResult + Coefficients[coefficientIndex] * InputFrame[coefficientIndex];
    end
end
```

ماژول FIR_Filter_tb تست بنچ برای اعتبارسنجی ماژول FIR Filter استفاده می‌شود. در این تست بنچ، ابتدا سیگنال‌های ورودی از جمله کلاک (Clk) و سیگنال ریست (Reset) مقداردهی اولیه می‌شوند. سپس ضرایب FIR Filter با مقادیر مختلف مقداردهی می‌شوند تا فیلتر آماده به کار شود. سپس یک مقدار ورودی به فیلتر داده می‌شود و خروجی آن بررسی می‌شود.

در طول اجرای تست بنچ، خروجی FIR Filter با استفاده از عبارت FilterOutput بررسی می‌شود. اگر خروجی ماژول با مقادیر مورد انتظار مطابقت نداشته باشد، خطایی گزارش می‌شود و تعداد خطاها ثبت می‌شود. پس از پایان اجرای تست بنچ، تعداد کل خطاها گزارش می‌شود و شبیه‌سازی متوقف می‌شود.

```
# Starting testbench simulation...
# Reset signal asserted.
# Setting coefficient[ 0] to    1
# Setting coefficient[ 1] to    2
# Setting coefficient[ 2] to    3
# Setting coefficient[ 3] to    4
# Setting coefficient[ 4] to    5
# Setting coefficient[ 5] to    4
# Setting coefficient[ 6] to    3
# Setting coefficient[ 7] to    2
# Setting coefficient[ 8] to    1
# Coefficient initialization completed.
# Inputting value 1 to the filter.
# Output[          0] is correct:         1
# Output[          1] is correct:         2
# Output[          2] is correct:         3
# Output[          3] is correct:         4
# Output[          4] is correct:         5
# Output[          5] is correct:         4
# Output[          6] is correct:         3
# Output[          7] is correct:         2
# Output[          8] is correct:         1
# Testbench simulation completed.
# Number of errors =           0
```

همانطور که مشاهده می‌کنید تعداد خطاها صفر می‌باشد که صحت عملکرد ماژول ما را تایید می‌کند.

ب

زمانی که ضرایب یک فیلتر FIR به صورت متقارن هستند، به عنوان یک فیلتر symmetric FIR شناخته می‌شوند. در این حالت، رابطه زیر بین ضرایب برقرار است:

$$\forall i : 0 \leq i \leq N : b_i = b_{N-i}$$

بنابراین، ضرایب فیلتر در موقعیت i و $N - i$ مقادیر یکسانی دارند. این به این معناست که ضرایب از مرکز به لبه‌ها به صورت متقارن تغییر می‌کنند.

در این حالت، انجام محاسبات برای محاسبه خروجی فیلتر ساده‌تر می‌شود. به عنوان مثال، اگر فیلتر با تعداد تپ فرد داشته باشیم، می‌توانیم از ویژگی متقارن بودن ضرایب استفاده کنیم تا از انجام محاسبات تکراری در حساب جمع و ضرب خودداری کنیم. این ویژگی باعث بهینه‌سازی عملیات محاسباتی می‌شود و بهبود عملکرد و سرعت فیلتر می‌دهد.

در ماژول FIR_Filter_Symmetric تغییرات فوق را اعمال کرده و با همان تست بنچ قسمت الف که در فایل FIR_Filter_Symmetric_tb.v موجود می‌باشد، صحت عملکرد ماژول را بررسی کردیم.

```
always @(posedge Clk) begin
    if(!Reset) begin                    // Reset condition
        for(tapIndex = 0; tapIndex < 9; tapIndex = tapIndex + 1) // Initialization
            InputFrame[tapIndex] <= 8'd0;    // Reset input buffer
        for(coefficientIndex = 0; coefficientIndex < 5; coefficientIndex = coefficientIndex + 1)
            Coefficients[coefficientIndex] <= 8'd0; // Reset coefficients
        FilterOutput <= 25'd0;              // Reset output
    end
    else begin
        if(CoefficientWriteEnable)          // If coefficient write enable is asserted
            Coefficients[CoefficientIndex] <= NewCoefficientValue; // Update coefficient
        InputFrame[0] <= InputData;         // New input
        for(tapIndex = 1; tapIndex < 9; tapIndex = tapIndex + 1) begin
            InputFrame[tapIndex] <= InputFrame[tapIndex-1]; // Shift input buffer
        end
        FilterOutput <= OutputResult;       // Update output
    end
end

always @(*) begin
    OutputResult = Coefficients[4] * InputFrame[4]; // Center tap operation
    for(coefficientIndex = 0; coefficientIndex < 4; coefficientIndex = coefficientIndex + 1) begin
        OutputResult = OutputResult + Coefficients[coefficientIndex] * (InputFrame[coefficientIndex] + InputFrame[8 - coefficientIndex]); // Symmetric taps operation
    end
end
```

```
# Starting testbench simulation...
# Reset signal asserted.
# Setting coefficient[ 0] to    1
# Setting coefficient[ 1] to    2
# Setting coefficient[ 2] to    3
# Setting coefficient[ 3] to    4
# Setting coefficient[ 4] to    5
# Setting coefficient[ 5] to    4
# Setting coefficient[ 6] to    3
# Setting coefficient[ 7] to    2
# Setting coefficient[ 8] to    1
# Coefficient initialization completed.
# Inputting value 1 to the filter.
# Output[          0] is correct:         1
# Output[          1] is correct:         2
# Output[          2] is correct:         3
# Output[          3] is correct:         4
# Output[          4] is correct:         5
# Output[          5] is correct:         4
# Output[          6] is correct:         3
# Output[          7] is correct:         2
# Output[          8] is correct:         1
# Testbench simulation completed.
# Number of errors =           0
```

همانطور که مشاهده می‌کنید تعداد خطاها صفر می‌باشد که صحت عملکرد ماژول ما را تایید می‌کند.

ج

- کاهش محاسبات تکراری: یکی از مزایای بزرگ ساختار متقارن این است که محاسبات تکراری را به شدت کاهش می‌دهد. زیرا ضرایب متقارن با یکدیگر همانند آینه در مرکز ایستاده‌اند. بنابراین، به جای محاسبه خروجی فیلتر برای هر تپ، می‌توان فقط نیمی از آنها را محاسبه کرد و سپس خروجی نهایی را با تکرار آینه آن محاسبات به دست آورد.
- بهینه‌سازی عملیات جمع و ضرب: با توجه به متقارن بودن ضرایب، عملیات جمع و ضرب می‌تواند به طور موازی انجام شود. این بهینه‌سازی موجب افزایش سرعت عملیاتی و کارایی فیلتر می‌شود.
- کاهش پیچیدگی سخت‌افزاری: به علت ساده‌تر بودن عملیات محاسباتی و از پیش پردازش‌ها، ساختار متقارن به عنوان یک راهکار سخت‌افزاری ساده‌تر و کارآمدتر محسوب می‌شود. این سادگی و کارآیی می‌تواند منجر به کاهش نیاز به منابع سخت‌افزاری و مصرف توان باشد.

د

در این مرحله با فرض اینکه ضرایب فقط مقادیر ۰، ۱، و -۱ را به خود می‌گیرند، عملیات ضرب ماتریسی لازم برای FIR فیلتر از بین می‌رود و صرفا با استفاده از عملیات جمع و تفریق می‌توانیم توان خروجی فیلتر را محاسبه کنیم. این به معنای این است که برای هر نمونه ورودی، ما فقط نیاز به یک عملیات جمع یا تفریق داریم. این عملیات باعث کاهش تعداد عملیات مورد نیاز برای پردازش و همچنین کاهش پیچیدگی سخت‌افزاری می‌شود و در نتیجه پیاده‌سازی آن بر روی FPGA هم ساده‌تر و کارا تر می‌شود.

```
parameter numberOfTaps = 10;               // Number of filter taps

integer tapIndex;                          // Loop index for taps
integer coefficientIndex;                  // Loop index for coefficients
reg signed[1:0] Coefficients[numberOfTaps - 1:0]; // Coefficient values
reg signed[7:0] InputFrame[numberOfTaps - 1:0]; // Input data buffer
reg signed[18:0] OutputResult;             // Intermediate result

always @(posedge Clk) begin
    if(!Reset) begin                       // Reset condition
        for(tapIndex = 0; tapIndex < numberOfTaps; tapIndex = tapIndex + 1) begin // Initialization
            InputFrame[tapIndex] <= 8'd0;    // Reset input buffer
            Coefficients[tapIndex] <= 2'd0;  // Reset coefficients
        end
        FilterOutput <= 25'd0;             // Reset output
    end
    else begin
        if(CoefficientWriteEnable)         // If coefficient write enable is asserted
            Coefficients[CoefficientIndex] <= NewCoefficientValue[1:0]; // Update coefficient
        InputFrame[0] <= InputData;        // New input
        for(tapIndex = 1; tapIndex < numberOfTaps; tapIndex = tapIndex + 1) begin
            InputFrame[tapIndex] <= InputFrame[tapIndex-1]; // Shift input buffer
        end
        FilterOutput <= OutputResult;      // Update output
    end
end

always @(*) begin
    OutputResult = 0;                      // Reset intermediate result
    for(coefficientIndex = 0; coefficientIndex < numberOfTaps; coefficientIndex = coefficientIndex + 1) begin
        if(Coefficients[coefficientIndex][0])
            OutputResult = OutputResult + (Coefficients[coefficientIndex][1] ? -InputFrame[coefficientIndex] : InputFrame[coefficientIndex]);
    end
end
```

سپس توسط ماژول FIR_Filter_One_Coef_tb عملکرد ماژول خود را بررسی می‌کنیم.

```
# Starting testbench simulation...
# Setting coefficient[ 0] to    0
# Setting coefficient[ 1] to   -1
# Setting coefficient[ 2] to   -1
# Setting coefficient[ 3] to   -1
# Setting coefficient[ 4] to    1
# Setting coefficient[ 5] to    1
# Setting coefficient[ 6] to   -1
# Setting coefficient[ 7] to    0
# Setting coefficient[ 8] to    1
# Setting coefficient[ 9] to    1
# Output[          0] is correct:       0
# Output[          1] is correct:      -1
# Output[          2] is correct:      -1
# Output[          3] is correct:      -1
# Output[          4] is correct:       1
# Output[          5] is correct:       1
# Output[          6] is correct:      -1
# Output[          7] is correct:       0
# Output[          8] is correct:       1
# Output[          9] is correct:       1
# Testbench simulation completed.
# Number of errors =           0
```

همانطور که مشاهده می‌کنید تعداد خطاها صفر می‌باشد که صحت عملکرد ماژول ما را تایید می‌کند.

ه

همانطور که در تصویر زیر مشاهده می‌کنید، با اعمال محدودیت زمانی، می‌توان گفت فرکانس کاری مدار برابر 80 MHz است.(دقت شود که در این حالت تمامی constraintها met شد.)

```
NET "clk" TNM_NET = clk;
TIMESPEC TS_clk = PERIOD "clk" 12.5 ns HIGH 50%;
```

$$\frac{1}{12.5 \times 10^{-9}} = 80MHz$$

و

ماژول FIR_Filter_Symmetric_Pipeline ورژن پایپ‌لاین شده‌ی قسمت ب سوال است که در آخر کار بجای محاسبه‌ی combinational خروجی، با استفاده از الگوریتم reduce بصورت پایپ‌لاین پیاده‌سازی کردیم.

```
always @(posedge Clk) begin
    // Stage 1: Center tap operation
    partialSums[0] <= Coefficients[4] * InputFrame[4];

    // Stage 2: Symmetric taps operation (1st half)
    partialSums[1] <= Coefficients[0] * (InputFrame[0] + InputFrame[8]) +
                      Coefficients[1] * (InputFrame[1] + InputFrame[7]);

    // Stage 3: Symmetric taps operation (2nd half)
    partialSums[2] <= Coefficients[2] * (InputFrame[2] + InputFrame[6]) +
                      Coefficients[3] * (InputFrame[3] + InputFrame[5]);

    // Stage 4: Reduction for final sum
    FilterOutput <= partialSums[0] + partialSums[1] + partialSums[2];
end
```

اکنون به کمک ماژول FIR_Filter_Symmetric_Pipeline_tb صحت عملکرد مدار خود را به کمک ISIM بررسی می‌کنیم.

```
Starting testbench simulation...
Finished circuit initialization process.
Reset signal asserted.
Setting coefficient[ 0] to  1
Setting coefficient[ 1] to  2
Setting coefficient[ 2] to  3
Setting coefficient[ 3] to  4
Setting coefficient[ 4] to  5
Setting coefficient[ 5] to  4
Setting coefficient[ 6] to  3
Setting coefficient[ 7] to  2
Setting coefficient[ 8] to  1
Coefficient initialization completed.
Inputting value 1 to the filter.
Output[         0] is correct:         1
Output[         1] is correct:         2
Output[         2] is correct:         3
Output[         3] is correct:         4
Output[         4] is correct:         5
Output[         5] is correct:         4
Output[         6] is correct:         3
Output[         7] is correct:         2
Output[         8] is correct:         1
Testbench simulation completed.
Number of errors =         0
```

همانطور که مشاهده می‌کنید تعداد خطاها برابر صفر است و مدار ما به درستی عمل می‌کند.
با اعمال محدودیت زمانی که در تصویر زیر مشاهده می‌کنید، می‌توان گفت فرکانس کاری مدار برابر 133 MHz است.(دقت شود که در این حالت تمامی constraintها met شد.)

```
NET "clk" TNM_NET = clk;
TIMESPEC TS_clk = PERIOD "clk" 7.5 ns HIGH 50%;
```

$$\frac{1}{7.5 \times 10^{-9}} = 133MHz$$

با استفاده از پایپ‌لاینینگ، عملکرد کد بهبود یافته است. این به معنای این است که فرآیند پردازش اطلاعات به چندین مرحله تقسیم شده است که هر مرحله تنها انجام یک قسمت از محاسبات را به‌صورت همزمان با سایر مراحل انجام می‌دهد. این بهینه‌سازی باعث افزایش سرعت عملکرد و کارایی کلی فیلتر می‌شود.

در ماژول قسمت ب، تمامی محاسبات در یک حلقه واحد انجام می‌شدند، در حالی که در کد بهبود یافته، محاسبات به چندین مرحله تقسیم شده‌اند. این تقسیم بندی سبب می‌شود که محاسبات هر مرحله زمان کمتری برای انجام داشته باشند و بتوانند با فرکانس کلاک بالاتری هماهنگ شوند.

ط

ماژول پیاده سازی شده یک فیلتر FIR است که با استفاده از تکنیک Resource Sharing بهینه سازی شده است. در این ماژول، داده های ورودی با فاصله های ۸ کلاک به ماژول اعمال می شوند. برای هر داده ورودی، داده های قبلی در یک بافر ذخیره می شوند و با استفاده از آنها خروجی فیلتر محاسبه می شود. همچنین، مقادیر ضریب فیلتر نیز در یک آرایه ذخیره می شوند و می توان آنها را به روز کرد. برای محاسبه خروجی فیلتر، هر ضریب با داده متناظرش در بافر ضرب می شود و سپس نتایج حاصل از ضرب ها جمع می شوند. این عملیات ضرب و جمع در ۱۰ مرحله انجام می شود که در هر مرحله، یک ضرب و یک جمع انجام می شود. این به معنی استفاده مجدد از منابع برای انجام ضرب و جمع است، که منجر به کاهش مصرف منابع می شود. در نهایت، خروجی فیلتر در هر مرحله به خروجی قبلی اضافه می شود و در پایان همه مراحل، خروجی نهایی فیلتر حاصل می شود. این ماژول همچنین قابلیت Reset و به روزرسانی ضرایب فیلتر را دارد.
در ماژول FIR_Filter_Resource_Share_tb صحت عملکرد مدار را بررسی و تایید می‌کنیم.

```
Starting testbench simulation...
Finished circuit initialization process.
Reset signal asserted.
Setting coefficient[ 0] to   1
Setting coefficient[ 1] to   2
Setting coefficient[ 2] to   3
Setting coefficient[ 3] to   4
Setting coefficient[ 4] to   5
Setting coefficient[ 5] to   6
Setting coefficient[ 6] to   7
Setting coefficient[ 7] to   8
Setting coefficient[ 8] to   9
Setting coefficient[ 9] to  10
Coefficient initialization completed.
Inputting value 1 to the filter.
Output[      10] is:       1
Output[      11] is:       1
Output[      12] is:       1
Output[      13] is:       1
Output[      14] is:       1
Output[      15] is:       1
Output[      16] is:       1
Output[      17] is:       1
Output[      18] is:       1
Output[      19] is:       1
Output[      20] is:       1
Testbench simulation completed.
Number of errors =       0
```

منابع مصرفی مدار این قسمت را در فایل resource share utilization.pdf و بخش ب را در simple utilization.pdf می‌توانید مشاهده کنید.
با مقایسه منابع مصرفی مشاهده می‌کنیم که تعداد DSP48 های بخش resource share یک عدد ولی در بخش ب ۱۰ عدد است و همانطور که انتظارش را داشتیم، با کاهش تعداد جمع کننده‌ها و ضرب

کننده‌ها، منابع مصرفی مدار کاهش یافت.

ح

ماژول FIR_Filter_Dual_Input یک فیلتر FIR دو ورودی است که در هر لبه کلاک، دو ورودی را دریافت و پردازش می‌کند. این ماژول حافظه‌ای برای ذخیره ضرایب و داده‌های ورودی دارد. در حالت ریست، ماژول در حالت اولیه قرار می‌گیرد. در حالت بعدی، اگر coefficient_write_enable فعال باشد، ضرایب و داده‌های ورودی در حافظه ذخیره می‌شوند. در غیر این صورت، ماژول به حالت بعدی می‌رود که در آن خروجی‌های FIR محاسبه می‌شوند.

صحت این ماژول توسط ماژول FIR_Filter_Dual_Input_tb مورد بررسی قرار گرفت و تایید شد. خروجی تست‌بنچ در حالتی که ورودی ۱و۲و۳و۴و۵و۶ باشد را در تصویر زیر مشاهده می‌کنید.

```
# First Output:        4, Second Output:        1
# First Output:        7, Second Output:        3
# First Output:       11, Second Output:        6
# First Output:       16, Second Output:       10
# First Output:       26, Second Output:       17
# First Output:       39, Second Output:       27
# First Output:       55, Second Output:       40
# First Output:       74, Second Output:       56
# First Output:       96, Second Output:       75
# First Output:      112, Second Output:       97
# First Output:      128, Second Output:      111
# First Output:      146, Second Output:      127
# First Output:      166, Second Output:      145
# First Output:      166, Second Output:      165
# First Output:      170, Second Output:      167
# First Output:      176, Second Output:      171
# First Output:      184, Second Output:      177
# First Output:      194, Second Output:      185
# First Output:      206, Second Output:      195
# First Output:      222, Second Output:      207
# First Output:      238, Second Output:      221
# First Output:      256, Second Output:      237
# First Output:      276, Second Output:      255
```