به نام خدا

محاسبه کننده کانولوشن

آزمایشگاه طراحی سیستم‌های دیجیتال

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

نویسندگان:

رادین چراغی ۴۰۱۱۰۵۸۱۵

امیرمحمد محفوظی ۴۰۱۱۰۶۴۶۹

سیدعلی جعفری ۴۰۰۱۰۴۸۸۹

تاریخ ارائه تکلیف:

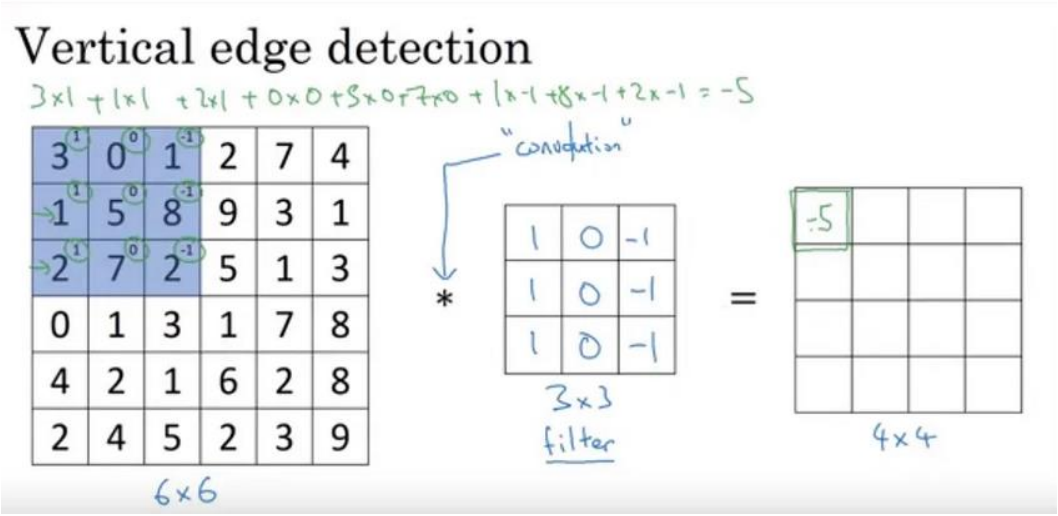
۱۴۰۳/۰۵/۲۲

مقدمه

در این تمرین قصد داریم مدار محاسبه کننده کانولوشن را طراحی و پیاده‌سازی کنیم. این مدار به عنوان ورودی یک ماتریس ورودی مربعی، یک ماتریس فیلتر مربعی با ابعاد کمتر از ماتریس اصلی، ابعاد ماتریس ورودی (عدد ۸ بیتی) و ابعاد ماتریس فیلتر (عدد ۸ بیتی) را دریافت می‌کند و در نهایت خروجی مدار نیز ماتریس حاصل از کانولوشن و ابعاد آن (که ۸ بیت است) می باشد. همچنین درایه‌های ماتریس‌ها اعداد ممیز شناور می‌باشند و به صورت IEEE 754 Single-precision ذخیره می‌شوند که بایستی برای هر یک از آن‌ها ۳۲ در نظر گرفته شود (یک بیت Sign، ۸ بیت Exponent و ۲۳ بیت Mantissa). نکته قابل ملاحظه دیگر در این تمرین این می‌باشد که ماتریس‌های ورودی و خروجی باید به صورت busهای ۳۲ بیتی باشند که در هر کلاک، یک داده ارسال/ دریافت می‌شود و بر اساس ابعاد داده شده داده‌ها مدیریت می‌شوند.

شرح آزمایش

ابتدا توضیح مختصری درباره فرایند محاسبه convolution می‌دهیم. همانطور که گفته شد یک ماتریس ورودی مربعی و یک ماتریس فیلتر مربعی با ابعاد کمتر از ماتریس اصلی داریم. در صورتی که ماتریس اصلی n*n و ماتریس فیلتر m*m باشد، ماتریس خروجی (n – m + 1)*(n – m + 1) خواهد بود که در حقیقت برابر با تعداد زیرماتریس‌ها با ابعاد ماتریس فیلتر در ماتریس اصلی می‌باشد. برای محاسبه کانولوشن در هر مرحله و بدست آوردن درایه‌ای در سطر iام و ستون jام در ماتریس حاصل، ابتدا زیرماتریسی به ابعاد ماتریس فیلتر در ماتریس اصلی در نظر گرفته که خانه گوشه بالا سمت راست آن درایه‌ای در سطر iام و ستون jام ماتریس اصلی باشد. سپس هر عنصر ماتریس فیلتر را در عنصر متناظر خود در زیرماتریس بدست آمده ضرب کرده و در نهایت اعداد بدست آمده را با هم جمع می‌کنیم. این حاصل جمع مقدار درایه در سطر iام و ستون jام در ماتریس حاصل می‌باشد. تصویر زیر بیانگر این عملیات می‌باشد.



اکنون به طراحی ماژول ConvCaluculator می‌پردازیم. ورودی و خروجی‌های این ماتریس به صورت زیر می‌باشند:

ورودی‌ها

- clk: همان کلاک اصلی مدار می‌باشد.
- rst: ریست مدار بوده که active high می‌باشد.
- data_in: داده ۳۲ بیتی ورودی مدار بوده که در هر کلاک یکی از درایه‌های ماتریس اصلی یا فیلتر می‌باشد.
- input_size: عدد ۸ بیتی که ابعاد ماتریس مربعی اصلی را مشخص می‌کند.
- filter_size: عدد ۸ بیتی که ابعاد ماتریس مربعی فیلتر را مشخص می‌کند.

خروجی‌ها

- data_out: داده ۳۲ بیتی خروجی مدار بوده که در هر کلاک یکی از درایه‌های ماتریس خروجی می‌باشد.
- done: خروجی تک بیتی که تمام شدن محاسبات مدار را نشان می‌دهد.
- output_size: عدد ۸ بیتی که ابعاد ماتریس مربعی حاصل را مشخص می‌کند.

حال با استفاده از زبان وریلاگ ماژول را پیاده‌سازی می‌کنیم. ابتدا همانگونه که در بالا توضیح داده شد، ورودی‌ها و خروجی‌های مدار را مشخص می‌کنیم. تصویر زیر این عملیات را نشان می‌دهد.

```
module ConvCalculator(
    input clk,
    input rst,
    input [31:0] data_in,
    input [7:0] input_size,
    input [7:0] filter_size,
    output reg [31:0] data_out,
    output reg done,
    output [7:0] output_size
);
```

از آن‌جایی که محاسبات مدار به صورت ممیز شناور ۳۲ بیتی می‌باشد، نیاز به دو function ضرب و جمع اعداد به صورت IEEE 754 Single-precision داریم. ابتدا function مربوط به جمع اعداد ممیز شناور را طراحی می‌کنیم. این تابع دو عدد ۳۲ بیتی a و b را ورودی گرفته و حاصل جمع را به صورت یک عدد ۳۲ بیتی خروجی می‌دهد. حال ساز و کار این تابع را توضیح می‌دهیم.

در این تابع ابتدا exponentهای دو عدد با یکدیگر مقایسه شده و عدد با exponent کوچکتر را پیدا می‌کنیم. سپس exponent عدد بزرگتر را برای محاسبات در نظر می‌گیریم. بنابراین بایستی mantissa عدد با exponent کمتر را به اندازه اختلاف exponentهای دو عدد شیفت به راست بدهیم. بدین منظور رجیسترهایی برای نگهداری sign، exponent و mantissa اعداد a و b تعریف می‌کنیم. mantissaها را به صورت denormalized در نظر گرفته و آن‌ها را ۲۴ بیتی تعریف می‌کنیم که بیت ۲۴ام آن‌ها برابر با یک می‌باشد. همچنین رجیسترهایی برای شیفت دادن mantissa دو عدد در حین فرایند یکی کردن نمای دو عدد تعریف کرده و نام آن‌ها را aligned_mantissa_a و aligned_mantissa_b می‌گذاریم. سپس تعدادی رجیستر کمکی مانند mantissa_sum و تعدادی رجیستر برای بخش‌های مجزا در عدد نهایی تعریف می‌کنیم. در نهایت فلگ‌هایی برای حالت‌های خاص از جمله صفر بودن a یا b، NaN بودن آن‌ها و infinity بودن هر یک از آن‌ها قرار می‌دهیم. تصویر زیر این عملیات را نشان می‌دهد.

```
function [31:0] fp_add_func;
    input [31:0] a;     // First 32-bit floating-point number
    input [31:0] b;     // Second 32-bit floating-point number

    // Local variable declarations
    reg sign_a, sign_b;
    reg [7:0] exponent_a, exponent_b;
    reg [23:0] mantissa_a, mantissa_b;
    reg [23:0] aligned_mantissa_a, aligned_mantissa_b;
    reg [7:0] exponent_diff;
    reg [7:0] exponent_result;
    reg sign_result;
    reg [24:0] mantissa_sum;
    reg [23:0] mantissa_final;
    reg [7:0] exponent_adjusted;

    // Flags for special cases
    reg a_is_zero, b_is_zero;
    reg a_is_inf, b_is_inf;
    reg a_is_nan, b_is_nan;
```

حال مقادیر مناسب را در هر یک از رجیسترهای فوق قرار می‌دهیم. ابتدا sign، exponent و mantissa دو عدد را از ورودی‌ها استخراج می‌کنیم. سپس فلگ‌ها برای شرایط خاص مقدار دهی می‌کنیم. در ادامه exponent دو عدد را یکی می‌کنیم. تصویر صفحه بعد این عملیات را نشان می‌دهد. همچنین کامنت‌های موجود در کد نیز بیانگر اعمال انجام شده می‌باشند.

```
begin
    // Extract sign, exponent, and mantissa
    sign_a = a[31];
    sign_b = b[31];
    exponent_a = a[30:23];
    exponent_b = b[30:23];
    mantissa_a = (exponent_a == 0) ? {1'b0, a[22:0]} : {1'b1, a[22:0]}; // Handle denormals
    mantissa_b = (exponent_b == 0) ? {1'b0, b[22:0]} : {1'b1, b[22:0]}; // Handle denormals

    // Set flags for special cases
    a_is_zero = (exponent_a == 8'd0 && a[22:0] == 23'd0);
    b_is_zero = (exponent_b == 8'd0 && b[22:0] == 23'd0);
    a_is_inf = (exponent_a == 8'd255 && a[22:0] == 23'd0);
    b_is_inf = (exponent_b == 8'd255 && b[22:0] == 23'd0);
    a_is_nan = (exponent_a == 8'd255 && a[22:0] != 23'd0);
    b_is_nan = (exponent_b == 8'd255 && b[22:0] != 23'd0);

    // Align exponents by shifting the mantissa of the smaller exponent
    exponent_diff = (exponent_a > exponent_b) ? (exponent_a - exponent_b) : (exponent_b - exponent_a);
    aligned_mantissa_a = (exponent_a > exponent_b) ? mantissa_a : (mantissa_a >> exponent_diff);
    aligned_mantissa_b = (exponent_b > exponent_a) ? mantissa_b : (mantissa_b >> exponent_diff);
    exponent_result = (exponent_a > exponent_b) ? exponent_a : exponent_b;
```

حال بر اساس signها، دو رجیستر aligned_mantissa_a و aligned_mantissa_b را با یکدیگر جمع یا تفریق می‌کنیم. در صورتی که علامت‌ها با یکدیگر یکسان باشند، جمع و در غیر این‌صورت تفریق می‌کنیم و حاصل را در mantissa_sum قرار می‌دهیم. از آن‌جایی که mantissa_sum را ۲۵ بیتی در نظر گرفته بودیم و دو عملوند جمع یا تفریق ۲۴ بیتی هستند، بیت ۲۵ ام هر عملوند را صفر قرار می‌دهیم. همچنین sign_result را برابر با علامت عددی قرار می‌دهیم که aligned_mantissa بزرگتری دارد. تصویر زیر این عملیات را نشان می‌دهد. کامنت‌های موجود در کد نیز بیانگر اعمال انجام شده می‌باشند.

```
// Add or subtract mantissas based on sign
mantissa_sum = (sign_a == sign_b) ?
({1'b0, aligned_mantissa_a} + {1'b0, aligned_mantissa_b}) :
                                (aligned_mantissa_a >= aligned_mantissa_b) ?
                                ({1'b0, aligned_mantissa_a} - {1'b0, aligned_mantissa_b}) :
                                   ({1'b0, aligned_mantissa_b} - {1'b0, aligned_mantissa_a});
sign_result = (aligned_mantissa_a >= aligned_mantissa_b) ? sign_a : sign_b;
```

در نهایت بایستی عدد حاصل را normalize کنیم. بدین منظور در صورتی که بیت ۲۵ ام mantissa_sum یک باشد بایستی یک بیت به راست شیفت بخورد و نما یک عدد زیاد شود. در غیر اینصورت چپ ترین یک را در mantissa پیدا کرده و سپس تا جایی شیفت می‌دهیم که بیت پرارزش آن یک شود و exponent را به همان مقدار کم می‌کنیم. در آخر خروجی تابع را مشخص

می‌کنیم و در این حین حالت‌های خاص را نیز در نظر می‌گیریم. تصویر زیر این عملیات را نشان می‌دهد. کامنت‌های موجود در کد نیز بیانگر اعمال انجام شده می‌باشند.

```
// Normalize the result mantissa
mantissa_final = mantissa_sum[24] ? mantissa_sum[24:1] : mantissa_sum[23:0];
exponent_adjusted = mantissa_sum[24] ? exponent_result + 1 : exponent_result;
norm_fac = mantissa_final[23] ? 0 : (mantissa_final[22] ? 1 : (mantissa_final[21] ? 2 : (mantissa_final[20] ? 3 :
(mantissa_final[19] ? 4 : (mantissa_final[18] ? 5 : (mantissa_final[17] ? 6 : (mantissa_final[16] ? 7 :
(mantissa_final[15] ? 8 : (mantissa_final[14] ? 9 : (mantissa_final[13] ? 10 : (mantissa_final[12] ? 11 :
mantissa_final[11] ? 12 : (mantissa_final[10] ? 13 : (mantissa_final[9] ? 14 : (mantissa_final[8] ? 15 :
mantissa_final[7] ? 16 : (mantissa_final[6] ? 17 : (mantissa_final[5] ? 18 : (mantissa_final[4] ? 19 :
mantissa_final[3] ? 20 : (mantissa_final[2] ? 21 : (mantissa_final[1] ? 22 : (mantissa_final[0] ? 23 : 24)))))))))))))))))));
mantissa_final = mantissa_final << norm_fac;
exponent_adjusted = exponent_adjusted - norm_fac;

// Return the result handling special cases
fp_add_func = a_is_nan ? a :
              b_is_nan ? b :
              (a_is_inf & b_is_inf & (sign_a != sign_b)) ? 32'hFFC00000 : // NaN
              a_is_inf ? a :
              b_is_inf ? b :
              a_is_zero ? b :
              b_is_zero ? a :
              {sign_result, exponent_adjusted[7:0], mantissa_final[22:0]};
```

حال تابع مربوط به ضرب اعداد ممیز شناور را پیاده‌سازی می‌کنیم. این تابع دو عدد ۳۲ بیتی a و b را ورودی گرفته و حاصل ضرب را به صورت یک عدد ۳۲ بیتی خروجی می‌دهد. ساز و کار این تابع را توضیح می‌دهیم. ابتدا رجیسترهایی برای نگهداری sign، exponent و mantissa اعداد a و b تعریف می‌کنیم. mantissaها را به صورت denormalized در نظر گرفته و آن‌ها را ۲۴ بیتی تعریف می‌کنیم که بیت ۲۴ام آن‌ها برابر با یک می‌باشد. تعدادی رجیستر برای بخش‌های مجزا در عدد نهایی تعریف می‌کنیم. همچنین یک رجیستر برای overflow قرار می‌دهیم. در نهایت فلگ‌هایی برای حالت‌های خاص از جمله صفر بودن a یا b، NaN بودن آن‌ها و infinity بودن هر یک از آن‌ها قرار می‌دهیم. تصویر زیر این عملیات را نشان می‌دهد.

```
function [31:0] fp_mul_func;
    input [31:0] a;    // First 32-bit floating-point number
    input [31:0] b;    // Second 32-bit floating-point number

    // Local variable declarations
    reg sign_a;
    reg sign_b;
    reg [7:0] exponent_a;
    reg [7:0] exponent_b;
    reg [23:0] mantissa_a;
    reg [23:0] mantissa_b;
    reg sign_result;
    reg [8:0] exponent_result;
    reg [47:0] mantissa_result;
    reg [22:0] mantissa_final;
    reg [7:0] exponent_final;
    reg overflow;

    // Flags for special cases
    reg a_is_zero, b_is_zero;
    reg a_is_inf, b_is_inf;
    reg a_is_nan, b_is_nan;
```

سپس مقادیر مناسب را در هر یک از رجیسترهای فوق قرار می‌دهیم. ابتدا sign، exponent و mantissa دو عدد را از ورودی‌ها استخراج می‌کنیم. در ادامه فلگ‌ها برای شرایط خاص مقدار دهی می‌کنیم. تصویر زیر این عملیات را نشان می‌دهد.

```
begin
    // Extract sign, exponent, and mantissa
    sign_a = a[31];
    sign_b = b[31];
    exponent_a = a[30:23];
    exponent_b = b[30:23];
    mantissa_a = {1'b1, a[22:0]}; // Implicit leading 1
    mantissa_b = {1'b1, b[22:0]}; // Implicit leading 1

    // Set flags for special cases
    a_is_zero = (exponent_a == 8'd0 && a[22:0] == 23'd0);
    b_is_zero = (exponent_b == 8'd0 && b[22:0] == 23'd0);
    a_is_inf = (exponent_a == 8'd255 && a[22:0] == 23'd0);
    b_is_inf = (exponent_b == 8'd255 && b[22:0] == 23'd0);
    a_is_nan = (exponent_a == 8'd255 && a[22:0] != 23'd0);
    b_is_nan = (exponent_b == 8'd255 && b[22:0] != 23'd0);
```

اکنون علامت حاصل را بدست می‌آوریم که XOR علامت‌های دو عدد ورودی است. سپس نمای نهایی را بدست می‌آوریم که جمع دو نمای اعداد وروی منهای Bias (۱۲۷) است. در ادامه حاصل نهایی را بدست آورده و normalize می‌کنیم. همچنین overflow را در صورتی که exponent از ۲۵۴ بیشتر شود فعال می‌کنیم. در نهایت خروجی تابع را مشخص می‌کنیم. تصاویر زیر عملیات فوق را نشان می‌دهد. همچنین کامنت‌های داخل کد شامل توضیحاتی می‌باشند که به فهم مدار کمک می‌کند.

```
// Calculate the result sign
sign_result = sign_a ^ sign_b;

// Add exponents and subtract bias (127)
exponent_result = exponent_a + exponent_b - 8'd127;

// Multiply mantissas
mantissa_result = mantissa_a * mantissa_b;
```

```
// Normalize the result mantissa and adjust exponent
mantissa_final = mantissa_result[47] ? mantissa_result[46:24] : mantissa_result[45:23];
exponent_final = mantissa_result[47] ? exponent_result + 1 : exponent_result;

// Check for overflow
overflow = (exponent_final > 8'd254);

// Return the result handling special cases
fp_mul_func = a_is_nan ? a :
              b_is_nan ? b :
              (a_is_inf & b_is_zero) | (b_is_inf & a_is_zero) ? 32'hFFC00000 : // NaN
              a_is_inf | b_is_inf ? {sign_result, 8'hFF, 23'd0} :
              a_is_zero | b_is_zero ? {sign_result, 31'd0} :
              overflow ? {sign_result, 8'hFF, 23'd0} : // Inf
              {sign_result, exponent_final[7:0], mantissa_final[22:0]};
```

حال بدنه ماژول ConvCalculator را پیاده‌سازی می‌کنیم. ابتدا سه ماتریس رجیستری input_matrix، filter_matrix و output_matrix را تعریف می‌کنیم. سپس رجیسترها و سیگنال‌های مورد نیاز از جمله اندیس‌ها برای پیمایش روی ماتریس‌ها، اندیس‌ها برای ورودی گرفتن و خروجی دادن، state که نشان‌دهنده حالت فعلی مدار است و تعدادی رجیستر کمکی دیگر را قرار می‌دهیم. حال با استفاده از parameterها حالت‌های مدار را مشخص می‌کنیم. سپس خروجی output_size را برابر با اختلاف ابعاد دو ماتریس ورودی به علاوه یک (تعداد زیرماتریس‌ها با ابعاد مشابه با ماتریس فیلتر) مقداردهی می‌کنیم. تصویر زیر اعمال گفته شده را نشان می‌دهد.

```
reg [31:0] input_matrix [0:255][0:255];  // Max size 256x256
reg [31:0] filter_matrix [0:15][0:15];   // Max size 16x16
reg [31:0] output_matrix [0:255][0:255]; // Output matrix

integer i, j, m, n;
reg [7:0] row_count;
reg [7:0] col_count;
reg [7:0] filter_row;
reg [7:0] filter_col;
reg [31:0] sum;
reg [31:0] buff;
reg [7:0] input_index;
reg [7:0] filter_index;
reg [3:0] state;
reg [7:0] output_index;

parameter IDLE = 0,
          READ_INPUT = 1,
          CONVOLVE = 2,
          WRITE_OUTPUT = 3;

assign output_size = input_size - filter_size + 1;
```

حال با استفاده از بلاک always حالت‌های مدار را پیاده‌سازی می‌کنیم. در صورتی که ورودی rst فعال باشد، تمامی رجیسترها و خروجی‌ها را صفر کرده و حالت فعلی را به IDLE تغییر می‌دهیم.

```
always @(posedge clk or posedge rst) begin
    if (rst) begin
        state <= IDLE;
        row_count <= 0;
        col_count <= 0;
        filter_row <= 0;
        filter_col <= 0;
        input_index <= 0;
        output_index <= 0;
        filter_index <= 0;
        done <= 0;
```

این مدار ۴ حالت دارد:

- IDLE: حالت پیش‌فرض مدار می‌باشد. در این حالت مانند زمانی که rst فعال باشد عمل می‌کنیم با این تفاوت که حالت بعدی مدار را READ_INPUT قرار می‌دهیم.

```
case (state)
    IDLE: begin
        row_count <= 0;
        col_count <= 0;
        filter_row <= 0;
        filter_col <= 0;
        input_index <= 0;
        output_index <= 0;
        filter_index <= 0;
        done <= 0;
        state <= READ_INPUT;
    end
```

- READ_INPUT: در این حالت ابتدا با تعداد ابعاد ماتریس‌ها ورودی گرفته و آن‌ها را در درایه‌های ماتریس‌های رجیستری input_matrix و filter_matrix قرار می‌دهیم. در نهایت رجیسترهای مشخص کننده اندیس در ماتریس‌ها را صفر کرده و حالت مدار را به CONVOLVE تغییر می‌دهیم.

```
READ_INPUT: begin
    if (input_index < input_size * input_size) begin
        input_matrix[input_index / input_size][input_index % input_size] = data_in;
        input_index <= input_index + 1;
    end else begin
        if (filter_index < filter_size * filter_size) begin
            filter_matrix[filter_index / filter_size][filter_index % filter_size] = data_in;
            filter_index <= filter_index + 1;
        end else begin
            state <= CONVOLVE;
            filter_index <= 0;
            row_count <= 0;
            col_count <= 0;
        end
    end
end
```

- CONVOLVE: در این حالت حاصل convolution محاسبه می‌شود. همانطور که توضیح داده شد، برای محاسبه کانولوشن در هر مرحله و بدست آوردن درایه‌ای در سطر iام و ستون jام در ماتریس حاصل، ابتدا زیرماتریسی به ابعاد ماتریس فیلتر در ماتریس اصلی در نظر گرفته که خانه گوشه بالا سمت راست آن درایه‌ای در سطر iام و ستون jام ماتریس اصلی باشد. سپس هر عنصر ماتریس فیلتر را در عنصر متناظر خود در زیرماتریس بدست آمده ضرب کرده و در نهایت اعداد بدست آمده را با هم جمع می‌کنیم. این حاصل جمع مقدار درایه در سطر iام و ستون jام در ماتریس حاصل می‌باشد. برای این کار از دو حلقه for استفاده می‌کنیم. در نهایت پس از اتمام فرایند محاسبه به حالت WRITE_OUTPUT می‌رویم و خروجی done را فعال می‌کنیم. تصویر زیر بیانگر عملیات بالا می‌باشد.
- WRITE_OUTPUT: در این استیت در هر کلاک یکی از درایه‌های ماتریس convolution در data_out قرار می‌گیرد.

```
WRITE_OUTPUT: begin
    if (output_index < output_size * output_size) begin
        data_out = output_matrix[output_index / output_size][output_index % output_size];
        output_index <= output_index + 1;
    end
    else begin
        state <= IDLE;
        row_count <= 0;
        col_count <= 0;
    end
end
```

حال نوبت به طراحی testbench مدار می‌شود. ابتدا رجیسترها و سیگنال‌های لازم برای نمونه‌گیری از ConvCalculator را تعریف کرده و از ماژول مذکور نمونه می‌گیریم. سپس در ادامه کلاک را شبیه‌سازی می‌کنیم. همچنین سه ماتریس رجیستری input_matrix، filter_matrix و output_matrix را تعریف می‌کنیم. این عملیات در تصویر زیر مشخص است.

```
module TB();
    reg [31:0] data_in;
    reg [7:0] input_size;
    reg [7:0] filter_size;
    wire [31:0] data_out;
    wire done;
    wire [7:0] output_size;
    // Instantiate the convolution module
    ConvCalculator uut (
        .clk(clk),
        .rst(rst),
        .data_in(data_in),
        .input_size(input_size),
        .filter_size(filter_size),
        .data_out(data_out),
        .done(done),
        .output_size(output_size)
    );
    // Clock generation
    initial begin
        clk = 0;
    end
    always begin
        #5 clk = ~clk;
    end
    // Testbench variables
    integer i, j;
    reg [31:0] input_matrix [3:0][3:0];
    reg [31:0] filter_matrix [1:0][1:0];
    reg [31:0] output_matrix [2:0][2:0];
```

حال تست‌های خود را در مدار قرار می‌دهیم. تصویر زیر تست اول را نشان می‌دهد. ماتریس فیلتر در این تست ماتریس همانی ۲*۲ می‌باشد. ابعاد ماتریس اصلی نیز ۴*۴ است.

```
module TB();
    initial begin
        rst = 1;
        data_in = 0;
        input_size = 4; // 4x4 input matrix
        filter_size = 2; // 2x2 filter matrix

        // Initialize input matrix with IEEE 754 representation
        input_matrix[0][0] = 32'h3fa00000; // 1.25
        input_matrix[0][1] = 32'h40200000; // 2.5
        input_matrix[0][2] = 32'h40400000; // 3.0
        input_matrix[0][3] = 32'h40900000; // 4.5
        input_matrix[1][0] = 32'h40b80000; // 5.75
        input_matrix[1][1] = 32'h3f800000; // 1.0
        input_matrix[1][2] = 32'h40880000; // 4.25
        input_matrix[1][3] = 32'h41100000; // 9.0
        input_matrix[2][0] = 32'h41000000; // 8.0
        input_matrix[2][1] = 32'h40a00000; // 5.0
        input_matrix[2][2] = 32'h40980000; // 4.75
        input_matrix[2][3] = 32'h40e00000; // 7.0
        input_matrix[3][0] = 32'h40000000; // 2.0
        input_matrix[3][1] = 32'h40800000; // 4.0
        input_matrix[3][2] = 32'h3e800000; // 0.25
        input_matrix[3][3] = 32'h3f000000; // 0.5

        // Initialize filter matrix with IEEE 754 representatio
        filter_matrix[0][0] = 32'h3f800000; // 1.0
        filter_matrix[0][1] = 32'h00000000; // 0.0
        filter_matrix[1][0] = 32'h00000000; // 0.0
        filter_matrix[1][1] = 32'h3f800000; // 1.0
```

حاصل convolution باید برابر با ماتریس زیر باشد.

$$\begin{pmatrix} 2.25 & 6.75 & 12 \\ 10.75 & 5.75 & 11.25 \\ 12 & 5.25 & 5.25 \end{pmatrix}$$

تصویر صفحه بعد خروجی transcript پس از شبیه‌سازی مدار در ModelSim را نشان می‌دهد. و همانطور که مشخص است خروجی با ماتریس فوق برابر است.

```
# Output matrix:
# 40100000 40d80000 41400000
# 412c0000 40b80000 41340000
# 41400000 40a80000 40a80000
# output size:   3
```

تصاویر زیر نیز نحوه ورودی دادن و خروجی گرفتن در تست‌بنچ را نشان می‌دهد.

```
// Deassert reset
#10 rst = 0;

// Load input matrix into the module
for (i = 0; i < input_size; i = i + 1) begin
    for (j = 0; j < input_size; j = j + 1) begin
        @(posedge clk);
        data_in = input_matrix[i][j];
    end
end

#10
// Load filter matrix into the module
for (i = 0; i < filter_size; i = i + 1) begin
    for (j = 0; j < filter_size; j = j + 1) begin
        @(posedge clk);
        data_in = filter_matrix[i][j];
    end
end
```

```
// Wait for computation to finish
wait(done);
@(posedge clk)
for (i = 0; i < input_size - filter_size + 1; i = i + 1) begin
    for (j = 0; j < input_size - filter_size + 1; j = j + 1) begin
        @(posedge clk);
        output_matrix[i][j] = data_out;
    end
end


// Print the output matrix
$display("Output matrix:");
for (i = 0; i < input_size - filter_size + 1; i = i + 1) begin
    for (j = 0; j < input_size - filter_size + 1; j = j + 1) begin
        $write("%h ", output_matrix[i][j]); // Print in hexadecimal format
    end
    $display("");
end

$write("output size: %d\n", output_size);

$stop;
```

تصویر زیر ماتریس ورودی تست دوم را نشان می‌دهد. ابعاد ماتریس فیلتر ۶ * ۶ بوده و ماتریس همانی است. ابعاد ماتریس اصلی نیز ۸ * ۸ است.

```
// Initialize input matrix with IEEE 754 representations
input_matrix[0][0] = 32'h3f800000; // 1
input_matrix[0][1] = 32'h3f800000; // 1
input_matrix[0][2] = 32'h3f800000; // 1
input_matrix[0][3] = 32'h3f800000; // 1
input_matrix[0][4] = 32'h3f800000; // 1
input_matrix[0][5] = 32'h3f800000; // 1
input_matrix[0][6] = 32'h3f800000; // 1
input_matrix[0][7] = 32'h3f800000; // 1

input_matrix[1][0] = 32'h3fa00000; // 1.25
input_matrix[1][1] = 32'h3fa00000; // 1.25
input_matrix[1][2] = 32'h3fa00000; // 1.25
input_matrix[1][3] = 32'h3fa00000; // 1.25
input_matrix[1][4] = 32'h3fa00000; // 1.25
input_matrix[1][5] = 32'h3fa00000; // 1.25
input_matrix[1][6] = 32'h3fa00000; // 1.25
input_matrix[1][7] = 32'h3fa00000; // 1.25

input_matrix[2][0] = 32'h3fc00000; // 1.5
input_matrix[2][1] = 32'h3fc00000; // 1.5
input_matrix[2][2] = 32'h3fc00000; // 1.5
input_matrix[2][3] = 32'h3fc00000; // 1.5
input_matrix[2][4] = 32'h3fc00000; // 1.5
input_matrix[2][5] = 32'h3fc00000; // 1.5
input_matrix[2][6] = 32'h3fc00000; // 1.5
input_matrix[2][7] = 32'h3fc00000; // 1.5
```

```
input_matrix[3][0] = 32'h3fe00000; // 1.75
input_matrix[3][1] = 32'h3fe00000; // 1.75
input_matrix[3][2] = 32'h3fe00000; // 1.75
input_matrix[3][3] = 32'h3fe00000; // 1.75
input_matrix[3][4] = 32'h3fe00000; // 1.75
input_matrix[3][5] = 32'h3fe00000; // 1.75
input_matrix[3][6] = 32'h3fe00000; // 1.75
input_matrix[3][7] = 32'h3fe00000; // 1.75

input_matrix[4][0] = 32'h40000000; // 2
input_matrix[4][1] = 32'h40000000; // 2
input_matrix[4][2] = 32'h40000000; // 2
input_matrix[4][3] = 32'h40000000; // 2
input_matrix[4][4] = 32'h40000000; // 2
input_matrix[4][5] = 32'h40000000; // 2
input_matrix[4][6] = 32'h40000000; // 2
input_matrix[4][7] = 32'h40000000; // 2

input_matrix[5][0] = 32'h40100000; // 2.25
input_matrix[5][1] = 32'h40100000; // 2.25
input_matrix[5][2] = 32'h40100000; // 2.25
input_matrix[5][3] = 32'h40100000; // 2.25
input_matrix[5][4] = 32'h40100000; // 2.25
input_matrix[5][5] = 32'h40100000; // 2.25
input_matrix[5][6] = 32'h40100000; // 2.25
input_matrix[5][7] = 32'h40100000; // 2.25
```

```
input_matrix[6][0] = 32'h40200000; // 2.5
input_matrix[6][1] = 32'h40200000; // 2.5
input_matrix[6][2] = 32'h40200000; // 2.5
input_matrix[6][3] = 32'h40200000; // 2.5
input_matrix[6][4] = 32'h40200000; // 2.5
input_matrix[6][5] = 32'h40200000; // 2.5
input_matrix[6][6] = 32'h40200000; // 2.5
input_matrix[6][7] = 32'h40200000; // 2.5

input_matrix[7][0] = 32'h40300000; // 2.75
input_matrix[7][1] = 32'h40300000; // 2.75
input_matrix[7][2] = 32'h40300000; // 2.75
input_matrix[7][3] = 32'h40300000; // 2.75
input_matrix[7][4] = 32'h40300000; // 2.75
input_matrix[7][5] = 32'h40300000; // 2.75
input_matrix[7][6] = 32'h40300000; // 2.75
input_matrix[7][7] = 32'h40300000; // 2.75

// Initialize filter matrix with IEEE 754 representations
filter_matrix[0][0] = 32'h3f800000; // 1.0
filter_matrix[0][1] = 32'h00000000; // 0.0
filter_matrix[0][2] = 32'h00000000; // 0.0
filter_matrix[0][3] = 32'h00000000; // 0.0
filter_matrix[0][4] = 32'h00000000; // 0.0
filter_matrix[0][5] = 32'h00000000; // 0.0
```

حاصل convolution باید برابر با ماتریس زیر باشد.

$$\begin{pmatrix} 9.75 & 9.75 & 9.75 \\ 11.25 & 11.25 & 11.25 \\ 12.75 & 12.75 & 12.75 \end{pmatrix}$$

تصویر صفحه بعد خروجی transcript پس از شبیه‌سازی مدار در ModelSim را نشان می‌دهد. و همانطور که مشخص است خروجی با ماتریس فوق برابر است.

```
# Output matrix:
# 411c0000 411c0000 411c0000
# 41340000 41340000 41340000
# 414c0000 414c0000 414c0000
# output size:   3
```

در تست سوم ماتریس ورودی را نسبت به تست دوم تغییر نمی‌دهیم. ماتریس فیلتر را ماتریس 2I با ابعاد ۸ * ۸ در نظر می‌گیریم. حاصل نهایی برابر با جمع عناصر روی قطر ماتریس اصلی ضرب در ۲ یعنی ۳۰ خواهد بود که در تصویر زیر در خروجی transcript به صورت ممیز شناور و هگزادسیمال نمایش داده شده است.

```
# Output matrix:
# 41f00000
# output size:   1
# ** Note: $stop    : C:/Users/ideapad 5/Desktop/Excersise/TB.v(232)
#    Time: 1285 ps  Iteration: 1  Instance: /TB
```