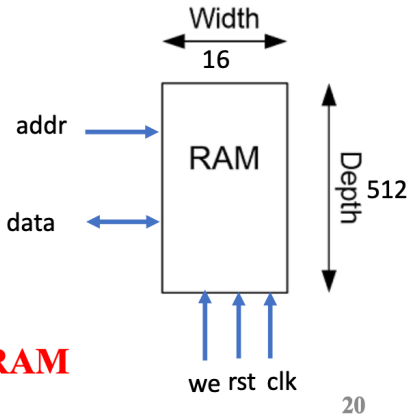
سوال چهارم)

در این سوال، از ما خواسته شده است که یک حافظه با دسترسی تصادفی یا RAM بسازیم. به منظور این کار، ابتدا ماژول RAM درون اسلایدها (اسلاید هشتم) را بررسی می‌کنیم.

```
module mem(input [8:0] addr, input we, rst, clk, inout [15:0] data);

        reg [15:0] mem[0:511];
        reg [15:0] read_data;

        assign data = !we ? read_data : 16'bz;

        always @(posedge clk) begin
                if (rst) read_data <= {16{1'b0}};
                else if (we) mem[addr] <= data;
                else read_data <= mem[addr];
        end

endmodule
```



دقت کنید که برای طراحی RAM خواسته شده درون سوال، واضحا باید تغییراتی در این ماژول نمونه بدهیم.

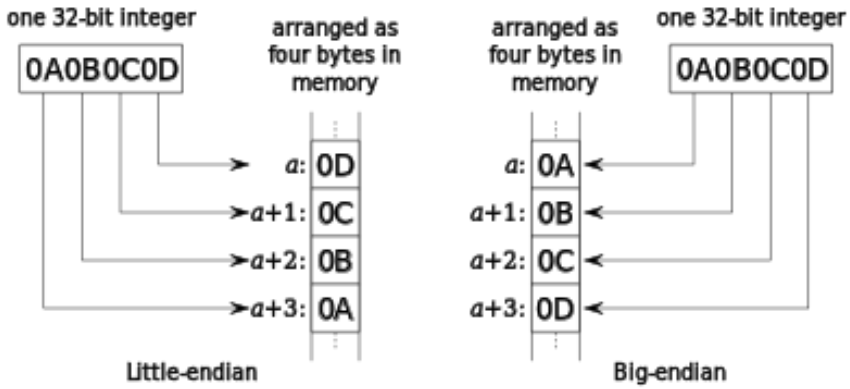
ابتدا دقت کنید که در نظر می‌گیریم این حافظه word addressable است و هر word شامل چهار byte است. با توجه به اینکه این حافظه 16 بیت آدرس دارد، در نتیجه دارای 4 * 2^16 بایت می‌باشد که RAM ما را 256kB می‌کند. به منظور پیاده سازی خود حافظه، از دستور زیر استفاده می‌کنیم:

```
reg [7:0] mem[0:4 * 2^16 - 1];
```

اینجا یک آرایه دو بعدی تشکیل می‌شود که دارای عمق 256k و عرض یک byte است که در کل شامل 64k کلمه است.

در ادامه، همانطور که در صورت سوال گفته شده‌است، این RAM باید سنکرون باشد، در نتیجه خواندن و نوشتن از این RAM در لبه بالارونده کلاک انجام می‌شود.

نکته حائز اهمیت در مورد بخش اول سوال این است که از ما خواسته شده‌است یک RAM که big endian است بسازیم. در حافظه‌های big endian، اگر یک کلمه درون حافظه ذخیره شود، MSByte آن کلمه در آدرس کوچکتر و LSByte آن در آدرس بزرگتر ذخیره می‌شود. در شکل زیر، این موضوع بهتر دیده می‌شود:



در این شکل، علاوه‌بر ذخیره‌سازی big endian، ذخیره‌سازی little endian نیز که دقیقا برعکس ذخیره‌سازی big endian است نیز نمایش داده شده‌است. در حافظه‌های little endian، اگر یک کلمه درون حافظه ذخیره شود، MSByte آن کلمه در آدرس بزرگتر و LSByte آن در آدرس کوچکتر ذخیره می‌شود.

برای پیاده‌سازی big endian RAM، از ماژول زیر استفاده می‌کنیم.

```
module big_endian_mem(
    input [15:0] addr,
    input [1:0] byte_sel,
    input we, re, clk,
    input [31:0] data,
    output reg [31:0] read_data,
    output reg [7:0] data_out_byte
);

reg [7:0] mem[0:4 * 2 ** 16 - 1];

wire [17:0] base_addr;
assign base_addr = addr << 2;

always @(posedge clk) begin
    if (we) begin
        mem[base_addr + 0] <= data[31:24];
        mem[base_addr + 1] <= data[23:16];
        mem[base_addr + 2] <= data[15:8];
        mem[base_addr + 3] <= data[7:0];
    end else if (re) begin
        read_data[31:24] <= mem[base_addr + 0];
        read_data[23:16] <= mem[base_addr + 1];
        read_data[15:8]  <= mem[base_addr + 2];
        read_data[7:0]   <= mem[base_addr + 3];

        case (byte_sel)
            2'b00: data_out_byte <= mem[base_addr + 0];
            2'b01: data_out_byte <= mem[base_addr + 1];
            2'b10: data_out_byte <= mem[base_addr + 2];
            2'b11: data_out_byte <= mem[base_addr + 3];
            default: data_out_byte <= 8'bz;
        endcase
    end else begin
        read_data <= 32'bz;
        data_out_byte <= 8'bz;
    end
end

endmodule
```

ورودی‌های این ماژول، شامل address، data و read_data است که به ترتیب آدرس، داده‌ای که می‌خواهیم در حافظه بنویسیم و داده خروجی از حافظه است. بدین منظور، ورودی های read_enable = re و write_enable = we را نیز داریم. همانطور که در بخش قبل توضیح داده شده بود، آدرس 16 بیتی این حافظه به یک word اشاره می‌کند و برای مشخص کردن آدرس واقعی (که به یک بایت اشاره می‌کند) باید address را در 4 ضرب کنیم. (دو واحد به چپ شیفت دهیم)

این حافظه اولویت را به نوشتن داده نسبت به خواندن داده می‌دهد. یعنی در صورتی که هم we و هم re فعال باشند، این حافظه نوشتن داده را انتخاب می‌کند. با توجه به اینکه این حافظه big endian می‌باشد، هنگامی که داده‌ی 4 بایتی برای نوشتن به آن داده می‌شود، MSByte آن کلمه در آدرس کوچکتر و LSByte آن در آدرس بزرگتر ذخیره می‌کند. در بخش زیر از ماژول، این مورد مشهود است.

```
mem[base_addr + 0] <= data[31:24];
mem[base_addr + 1] <= data[23:16];
mem[base_addr + 2] <= data[15:8];
mem[base_addr + 3] <= data[7:0];
```

خروجی read_data این ماژول که همان data_out تعریف شده درون صورت سوال است، با توجه به endianness تعیین می‌شود. بدین ترتیب که word خروجی داده شده که شامل 4 بایت است، با همان ترتیبی که ابتدا به حافظه ورودی داده شده بود نشان داده می‌شود. در بخش زیر از ماژول، این مورد مشهود است.

```
read_data[31:24] <= mem[base_addr + 0];
read_data[23:16] <= mem[base_addr + 1];
read_data[15:8]  <= mem[base_addr + 2];
read_data[7:0]   <= mem[base_addr + 3];
```

برای اینکه endianness این ماژول را تعیین کنیم، ورودی ۲ بیتی byte_sel به ماژول اضافه می‌کنیم. این ورودی بدین صورت کار می‌کند که هنگامی که از حافظه مقداری را می‌خوانیم، علاوه بر اینکه کلمه خواسته شده را خروجی می‌دهیم، بایت point شده درون ورودی byte_sel را نیز نشان می‌دهیم. این بایت که در واقع بایت درون آدرس address<<2+byte_sel است را در خروجی data_out_byte قرار می‌دهد. در بخش زیر از ماژول، این مورد مشهود است.

```
case (byte_sel)
     2'b00: data_out_byte <= mem[base_addr + 0];
     2'b01: data_out_byte <= mem[base_addr + 1];
     2'b10: data_out_byte <= mem[base_addr + 2];
     2'b11: data_out_byte <= mem[base_addr + 3];
```

که base_addr در واقع همان address<<2 است.

در صفحه بعد، طراحی little_endian_mem توضیح داده شده‌است.

در ادامه، little endian RAM را طراحی می‌کنیم. این ماژول عملا تفاوتی با big endian RAM ندارد و تنها در ترتیب ذخیره‌سازی و خواندن داده متفاوت است. ماژول زیر، ماژول طراحی شده برای little endian RAM است.

```
module little_endian_mem(
    input [15:0] addr,
    input [1:0] byte_sel,
    input we, re, clk,
    input [31:0] data,
    output reg [31:0] read_data,
    output reg [7:0] data_out_byte
);

reg [7:0] mem[0:4 * 2 ** 16 - 1];

wire [17:0] base_addr;
assign base_addr = addr << 2;

always @(posedge clk) begin
    if (we) begin
        mem[base_addr + 0] <= data[7:0];
        mem[base_addr + 1] <= data[15:8];
        mem[base_addr + 2] <= data[23:16];
        mem[base_addr + 3] <= data[31:24];
    end else if (re) begin
        read_data[7:0]   <= mem[base_addr + 0];
        read_data[15:8]  <= mem[base_addr + 1];
        read_data[23:16] <= mem[base_addr + 2];
        read_data[31:24] <= mem[base_addr + 3];

        case (byte_sel)
            2'b00: data_out_byte <= mem[base_addr + 0];
            2'b01: data_out_byte <= mem[base_addr + 1];
            2'b10: data_out_byte <= mem[base_addr + 2];
            2'b11: data_out_byte <= mem[base_addr + 3];
            default: data_out_byte <= 8'bz;
        endcase
    end
end

endmodule
```

تفاوت این ماژول با ماژول big endian در این قسمت است که little endian بودن را مشهود می‌کند.

```
    if (we) begin
        mem[base_addr + 0] <= data[7:0];
        mem[base_addr + 1] <= data[15:8];
        mem[base_addr + 2] <= data[23:16];
        mem[base_addr + 3] <= data[31:24];
    end else if (re) begin
        read_data[7:0]   <= mem[base_addr + 0];
        read_data[15:8]  <= mem[base_addr + 1];
        read_data[23:16] <= mem[base_addr + 2];
        read_data[31:24] <= mem[base_addr + 3];
```

برای تست این ماژول‌ها، از یک اسکریپت پایتون به اسم generator.py استفاده می‌کنیم. این اسکریپت، با دستور زیر ۲ فایل تست بنچ برای RAM های little endian و big endian تولید می‌کند. این تست‌بنچ‌ها تست‌های کاملا یکسانی دارند که رندوم تولید شده است. تکه‌کد تولید testهای رندوم در زیر آمده‌است:

```
def generate_random_test(num_tests):
    addr = [random.randint(0, 65535) for _ in range(num_tests)]
    data = [random.randint(0, 4294967295) for _ in range(num_tests)]
    byte_sel = [random.randint(0, 3) for _ in range(num_tests)]

    test_code = generate_test(addr, data, byte_sel)
    return test_code

def generate_test(addr, data, byte_sel):
    test = []
    for i in range(len(addr)):
        test.append(f"addr = 16'h{addr[i]:04X};")
        test.append(f"data = 32'h{data[i]:08X};")
        test.append("we = 1;")
        test.append("#10;")
        test.append("we = 0;")
        test.append(f"$display(\"Written %h to address %h\", data, addr);")
        test.append("#10;")

    for i in range(len(addr)):
        test.append(f"addr = 16'h{addr[i]:04X};")
        test.append(f"byte_sel = 2'b{byte_sel[i]:02b};")
        test.append("re = 1;")
        test.append("#10;")
        test.append(f"$display(\"Read full data at address %h: %h\", addr, read_data);")
        test.append(f"$display(\"Byte %h at address %h: %h\", byte_sel, addr, data_out_byte);")
        test.append("re = 0;")
        test.append("#10;")

    return '\n'.join(test)
```

پس از تولید این ماژول‌های تست‌بنچ، این فایل‌های بستر آزمون با استفاده از دستور زیر ران می‌شوند و خروجی آن‌ها در فایل‌های جدا قرار می‌گیرند:

```
iverilog -o tb little_endian_mem_tb.v
vvp tb >> littleOutput.txt
iverilog -o tb big_endian_mem_tb.v
vvp tb >> bigOutput.txt
```

سپس با استفاده از دستور diff، تفاوت این فایل‌های output را مشاهده می‌کنیم:

```
diff -y bigOutput.txt littleOutput.txt >> diff.txt
```

مجموعه دستورات یادشده درون فایل script.sh قرار گرفته‌اند که با دستور زیر، می‌توان به تعداد num_test برای ماژول‌های تست‌بنچ تست جنریت کرد و سپس، خروجی آن‌ها را مقایسه کرد:

```
./script.sh num_test
```

به عنوان مثال، با اجرای دستور زیر، خروجی مشاهده شده در صفحه بعد را می‌بینیم:

```
./script.sh 10
```

تفاوت‌های مشاهده شده، به دلیل endianness حافظه‌ها بوده که برای مثال در تفاوت اول، در بیت سوم (index = 2) آدرس f64e3fe0، در حافظه big endian مقدار 3f و در پردازنده little endian مقدار 4e ذخیره شده‌است که به درستی نشانگر تفاوت این دو حافظه است.

در فایل diff.text موجود درون فولدر q4، همین فایل خروجی با num_test = 1000 آورده شده‌است. دقت کنید این تست‌ها مطابق خواسته استاد، کاملا رندوم جنریت شده‌اند و تفاوت خروجی‌های فایل‌های بستر آزمون little endian و big endian مشهود است.

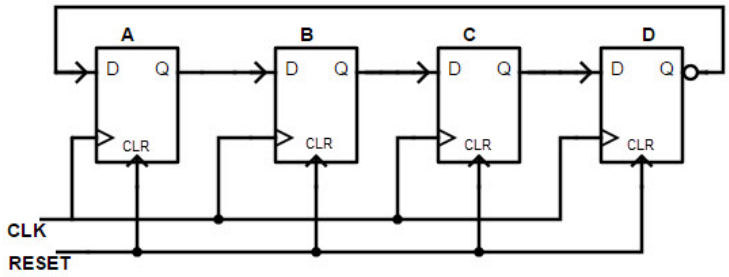
q4 > diff.txt

```
Written f64e3fe0 to address 2563                    Written f64e3fe0 to address 2563
Written 71cbe429 to address 41f4                    Written 71cbe429 to address 41f4
Written 2cc74c08 to address 991a                    Written 2cc74c08 to address 991a
Written 25c24ef7 to address 6bdb                    Written 25c24ef7 to address 6bdb
Written afba848b to address 2a7a                    Written afba848b to address 2a7a
Written 0f583a07 to address d8a8                    Written 0f583a07 to address d8a8
Written 9fc652ac to address c274                    Written 9fc652ac to address c274
Written ea834519 to address c914                    Written ea834519 to address c914
Written db7fc0ee to address 86ea                    Written db7fc0ee to address 86ea
Written 47479db0 to address d97a                    Written 47479db0 to address d97a
Read full data at address 2563: f64e3fe0                Read full data at address 2563: f64e3fe0
Byte 2 at address 2563: 3f                  |       Byte 2 at address 2563: 4e
Read full data at address 41f4: 71cbe429                Read full data at address 41f4: 71cbe429
Byte 2 at address 41f4: e4                  |       Byte 2 at address 41f4: cb
Read full data at address 991a: 2cc74c08                Read full data at address 991a: 2cc74c08
Byte 3 at address 991a: 08                  |       Byte 3 at address 991a: 2c
Read full data at address 6bdb: 25c24ef7                Read full data at address 6bdb: 25c24ef7
Byte 3 at address 6bdb: f7                  |       Byte 3 at address 6bdb: 25
Read full data at address 2a7a: afba848b                Read full data at address 2a7a: afba848b
Byte 1 at address 2a7a: ba                  |       Byte 1 at address 2a7a: 84
Read full data at address d8a8: 0f583a07                Read full data at address d8a8: 0f583a07
Byte 1 at address d8a8: 58                  |       Byte 1 at address d8a8: 3a
Read full data at address c274: 9fc652ac                Read full data at address c274: 9fc652ac
Byte 0 at address c274: 9f                  |       Byte 0 at address c274: ac
Read full data at address c914: ea834519                Read full data at address c914: ea834519
Byte 3 at address c914: 19                  |       Byte 3 at address c914: ea
Read full data at address 86ea: db7fc0ee                Read full data at address 86ea: db7fc0ee
Byte 0 at address 86ea: db                  |       Byte 0 at address 86ea: ee
Read full data at address d97a: 47479db0                Read full data at address d97a: 47479db0
Byte 3 at address d97a: b0                  |       Byte 3 at address d97a: 47
Test completed.                            Test completed.
big_endian_mem_tb.v:187: $finish called at 400 (1s)     |       little_endian_mem_tb.v:187: $finish called at 400 (1s)
```

تمامی فایل‌ها، از جمله فایل‌های پایتون و بش درون فولدر q4 موجود می‌باشد.

سوال ششم)

در این سوال، از ما خواسته شده است که شمارنده جانسون با استفاده از DFF طراحی کنیم. قبل از هرچیزی، به توضیح شمارنده جانسون می‌پردازیم. این شمارنده، شامل N تا فلیپ‌فلاپ نوع d یا DFF است که به صورت سری به هم متصل هستند. ورودی DFF شماره i به طور مستقیم به خروجی DFF شماره i-1 متصل است. البته ورودی DFF اول به معکوس خروجی DFF آخر متصل می‌باشد. شماتیک این مدار برای N=4 به این صورت است:



ابتدا به طراحی DFF می‌پردازیم.
ماژول DFF:

```
module d_flip_flop (
    input d,
    input clk,
    input reset,
    output reg q
);

always @(posedge clk, posedge reset)
begin
    if (reset)
        q <= 1'b0;
    else
        q <= d;
end

endmodule
```

توضیح ماژول: این ماژول شامل ورودی و خروجی d و q است که همان ورودی و خروجی اصلی DFF ما می‌باشند. همچنین این ماژول ورودی‌های clk و reset را دارد. reset این مدار به صورت آسنکرون کار می‌کند و active high است. به این ترتیب که در هر زمانی که reset از 0 به 1 تبدیل شود، درون بلاک always می‌رویم. همچنین در هر لبه بالارونده کلاک، ورودی d به خروجی q منتقل می‌شود.

پس از طراحی ماژول DFF، ماژول johnson counter را طبق توضیحات اول سوال طراحی می‌کنیم. برای این کار، در ابتدای ماژول، پارامتر N را تعیین می‌کنیم. دقت کنید که در اینجا مقدار default value = 4 را برای N تعریف کردیم اما هنگام گرفتن instance از این ماژول در ماژول test bench، می‌توانیم این پارامتر را تغییر دهیم. سپس با استفاده از بلوک generate، به تعداد N تا DFF تولید می‌کنیم (instance می‌گیریم) و ورودی‌های هر کدام از این DFFها را مشخص می‌کنیم.

همانطور که در اول سوال توضیح داده شد، طراحی شمارنده جانسون بسیار ساده‌است. کافی‌ست که ورودی clk و reset یکسان به همه DFFها بدهیم. همچنین ورودی DFF شماره i را از خروجی DFF شماره i-1 می‌گیریم، مگر DFF شماره اول که ورودی آن، معکوس خروجی DFF شماره N است. با استفاده از رابطه زیر، ورودی هر DFF مشخص می‌شود.

$$\text{input}(\text{DFF}[i]) = \text{output}(\text{DFF}[i-1]) \text{ for } \forall i; 1 \leq i < N$$
$$\text{input}(\text{DFF}[0]) = \sim\text{output}(\text{DFF}[N-1])$$

در صفحه بعد، ماژول را مشاهده می‌کنید.

```
`include "d_flip_flop.v"

module johnson_counter #(
    parameter N = 4
) (
    input clk,
    input reset,
    output [N-1:0] q
);

genvar i;
generate
    for (i = 0; i < N; i=i+1) begin : dff_loop
        d_flip_flop dff (
            .d(i == 0 ? ~q[N-1] : q[i-1]),
            .clk(clk),
            .reset(reset),
            .q(q[i])
        );
    end
endgenerate

endmodule
```

در این ماژول، ورودی clk و reset را داریم که به طور یکسان به همه‌ی DFFهای تولید شده در بلاک generate ورودی داده می‌شوند. همچنین خروجی q را داریم که مجموع خروجی‌های DFFها است. همچنین پارامتر N مشخص شده‌است که به هنگام instance گرفتن از این ماژول، آن را تعیین می‌کنیم. در ادامه نیز یک حلقه تعریف کردیم که به تعداد N تا DFF تولید می‌کند. (instance می‌گیرد)

در ادامه، اولین ماژول test bench را برای این شمارنده طراحی می‌کنیم. در این فایل بستر آزمون، بررسی می‌کنیم که پارامتر N برای ساخت شمارنده جانسون به درستی استفاده شده باشد. برای این‌کار، دو پارامتر CLK_PERIOD و SIMULATION_TIME را در ابتدای ماژول آورده‌ایم. سپس، به ازای N=4,8,16,32 از ماژول شمارنده جانسون (johnson_counter) تولید می‌کنیم. (instance می‌گیریم)

فایل parameter_tb:

```
`include "johnson_counter.v"

module parameter_tb;

parameter CLK_PERIOD    = 10;
parameter SIM_DURATION  = 1000;

reg clk     = 0;
reg reset   = 0;

wire [3:0]  q1;
wire [7:0]  q2;
wire [15:0] q3;
wire [31:0] q4;

johnson_counter #(.N(4)) dut4 (
    .clk(clk),
    .reset(reset),
    .q(q1)
);
```

```
johnson_counter #(.N(8)) dut8 (
    .clk(clk),
    .reset(reset),
    .q(q2)
);

johnson_counter #(.N(16)) dut16 (
    .clk(clk),
    .reset(reset),
    .q(q3)
);

johnson_counter #(.N(32)) dut32 (
    .clk(clk),
    .reset(reset),
    .q(q4)
);

always #((CLK_PERIOD / 2)) clk = ~clk;

initial begin
    $dumpfile("waveform.vcd");
    $dumpvars(0, parameter_tb);
    reset = 1;
    #10
    reset = 0;
end

always @(posedge clk) begin
    $display("q1 = %b, q2 = %b, q3 = %b, q4 = %b", q1 ,q2 ,q3, q4);
    if ($time >= SIM_DURATION)
        $finish;
end

endmodule
```

خروجی این فایل test bench پس از ران شدن در فایل output_parameter.txt در پیوست آمده‌است. همانطور که مشاهده می‌شود، با استفاده از یک ماژول و با پارامتر N، شمارنده‌های جانسون متفاوتی ساختیم و همزمان از آن‌ها استفاده کردیم.

چند خط ابتدای output_parameter.txt:

```
q1 = 0000, q2 = 00000000, q3 = 0000000000000000, q4 = 00000000000000000000000000000000
q1 = 0000, q2 = 00000000, q3 = 0000000000000000, q4 = 00000000000000000000000000000000
q1 = 0001, q2 = 00000001, q3 = 0000000000000001, q4 = 00000000000000000000000000000001
q1 = 0011, q2 = 00000011, q3 = 0000000000000011, q4 = 00000000000000000000000000000011
q1 = 0111, q2 = 00000111, q3 = 0000000000000111, q4 = 00000000000000000000000000000111
q1 = 1111, q2 = 00001111, q3 = 0000000000001111, q4 = 00000000000000000000000000001111
q1 = 1110, q2 = 00011111, q3 = 0000000000011111, q4 = 00000000000000000000000000011111
q1 = 1100, q2 = 00111111, q3 = 0000000000111111, q4 = 00000000000000000000000000111111
q1 = 1000, q2 = 01111111, q3 = 0000000001111111, q4 = 00000000000000000000000001111111
q1 = 0000, q2 = 11111111, q3 = 0000000011111111, q4 = 00000000000000000000000011111111
q1 = 0001, q2 = 11111110, q3 = 0000000111111111, q4 = 00000000000000000000000111111111
q1 = 0011, q2 = 11111100, q3 = 0000001111111111, q4 = 00000000000000000000001111111111
q1 = 0111, q2 = 11111000, q3 = 0000011111111111, q4 = 00000000000000000000011111111111
q1 = 1111, q2 = 11110000, q3 = 0000111111111111, q4 = 00000000000000000000111111111111
q1 = 1110, q2 = 11100000, q3 = 0001111111111111, q4 = 00000000000000000001111111111111
```

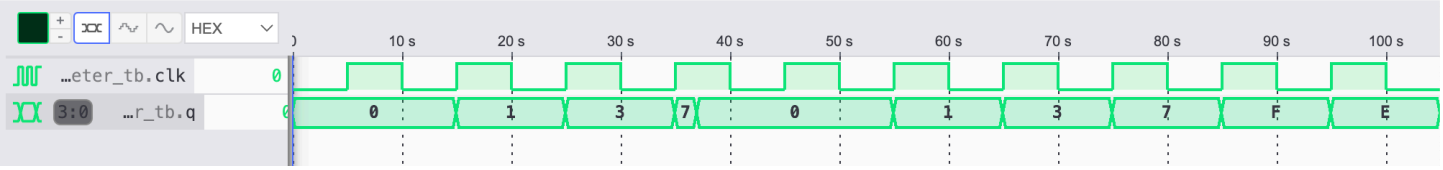
حال که از کارکرد parameter N در ماژول johnson_counter مطمئن شدیم، در تست‌بنچ بعدی، ورودی آسنکرون reset را بررسی می‌کنیم. بدین منظور، در هر زمانی که reset از 0 به 1 تبدیل شود، مقادیر خروجی DFFها باید صفر بشود. بدین منظور، از یک شمارنده جانسون با N=4 استفاده می‌کنیم که در مضارب 10، کلاک آن تغییر می‌کند. سپس فایل waveform خروجی را بررسی می‌کنیم.

فایل reset_tb:

```
`include "johnson_counter.v"

module parameter_tb;

parameter CLK_PERIOD    = 10;
parameter SIM_DURATION  = 100;

reg clk     = 0;
reg reset   = 0;

wire [3:0]  q;

johnson_counter #(.N(4)) dut (
    .clk(clk),
    .reset(reset),
    .q(q)
);


always #((CLK_PERIOD / 2)) clk = ~clk;

initial begin
    $dumpfile("waveform.vcd");
    $dumpvars(0, parameter_tb);
    reset = 1;
    #10
    reset = 0;
    #27
    reset = 1;
    #15
    reset = 0;

end

always @(posedge clk) begin
    if ($time >= SIM_DURATION)
        $finish;
end

endmodule
```

با توجه به اینکه از ابزار iverilog برای کامپایل‌کردن و ران گرفتن از کد وریلاگ استفاده شده‌است، برای گرفتن خروجی waveform نیز از دستورات ابتدای بلاک initial استفاده می‌کنیم

در صفحه بعد، فایل waveform خروجی که با یک اکتنشن از vscode باز شده‌است را مشاهده می‌کنید.



تحلیل فایل waveform:

ابتدا برای 10 ثانیه، reset فعال است. در ثانیه 15 که لبه بالارونده کلاک است، شمارنده شروع به کار می‌کند. پس از گذشت 27 ثانیه از ثانیه 10، که یعنی به ثانیه 37 رسیدیم، reset فعال می‌شود. چون ورودی آسنکرون است، در همان ثانیه 37 مقدار خروجی 0000 می‌شود. پس از گذشت 13 ثانیه، دوباره reset غیر فعال می‌شود و در ثانیه 55 در لبه بالارونده کلاک، دوباره شمارنده شروع به کار می‌کند.

این تحلیل نشان می‌دهد که ورودی reset به صورت آسنکرون خروجی را تغییر می‌دهد.

. دقت کنید دستورات برای کامپایل کردن و ران شدن این فایل به ترتیب به این صورت است:

```
iverilog -o tb johnson_counter_tb.v
vvp tb
```

فایل‌های نام‌برده شده، در فولدر q6 موجود هستند.