**به نام خدا**



**عنوان**

**پروژه درس مدارهای واسط**

**استاد**

**دکتر فصحتی**

**نام و شماره دانشجویی**

**مهدی بهرامیان 401171593**

**محمد علیزاده 401106244**

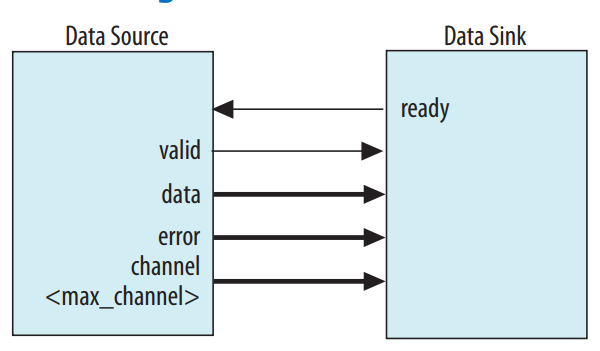
در این پروژه، هدف ما پیاده‌سازی اینترکانکشن‌های مذکور و مقایسه عملکرد آن‌ها با یکدیگر بود. به دلیل سختی محک زدن عملکرد و مقایسه آن میان دو اینترکانکشن متفاوت، این مقایسه با توجه به روش پیاده‌سازی انجام می‌گیرد، چرا که فرکانس برد با توجه به مشترک بودن ثابت است و تنها اختلاف در پیاده سازی اینترکانکشن‌هاست. در واقع اینترکانکشنی که چرخه‌های بیشتری نیاز داشته باشد، واضحا عملکرد بدتری دارد. در ادامه در هربخش پیاده سازی هرکدام از اینتکانکشن‌ها به همراه منابع و توضیح مختصر پروتکل قرار داده شده است.

1. **Avalon stream**

بر طبق منبع [mnl\_avalon\_spec-683091-667068.pdf](../../../../../../C:/Users/surface/Downloads/mnl_avalon_spec-683091-667068.pdf) و همچنین [5.2. Avalon® Streaming Interface Signal Roles](https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/streaming-interface-signal-roles.html)، این پروتکل را پیاده سازی کردیم. ویژگی کلی این پروتکل به این صورت است که پیام‌ها صرفا به صورت یکطرفه ارسال می‌شوند. به منظور رفع این کاستی، ما دو بار این پروتکل را پیاده‌سازی کردیم که یکی از مستر به اسلیو و دیگری از اسلیو به مستر است. روند کلی ارسال داده نیز در آن به این فرم است که از طرف مستر، برای تنظیم مقادیر A و B، ابتدا یک مقدار فرستاده می‌شود که مشخص می‌کند می‌خواهیم A را تنظیم کنیم یا B را. این مقدار برای ورودی A برابر 1 و برای متغیر B برابر 2 است. در ادامه، داده به صورت 8 بیت 8 بیت از جایگاه پرارزش به کم‌ارزش ارسال می‌شود. در بخش اول، سیگنال استارت پکت و ولید روشن شده و پس از این بخش استارت پکت خاموش می‌شود ولی ولید تا زمان انتهای ارسال روشن می‌ماند. در انتها نیز یک سیگنال اند پکت ارسال می‌شود که با توجه به ثابت بودن طول بسته‌های ما عملا بلا استفاده است. کد زده شده برای این بخش در دو فایل avalon\_st\_slave\_wrapper.sv و avalon\_st\_master\_wrapper.sv قرار گرفته است. تست بنچ مربوطه نیز با نام avalon\_st\_tb.sv قرار داده شده است.

سیگنال‌های موجود در این پروتکل به شرح ذیل می‌باشند (برای هر طرف جریان داده، یعنی در واقع از هرکدام از این سیگنال‌ها یک جفت به صورت رفت و برگشتی در پیاده سازی ما وجود دارد):

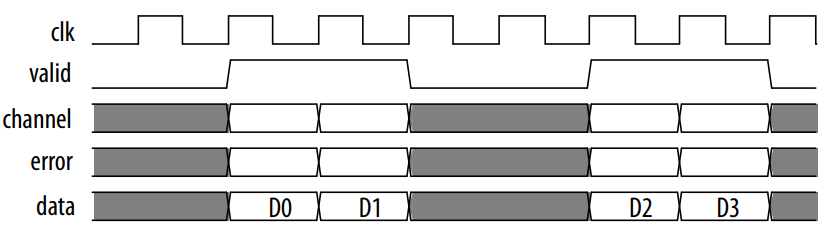
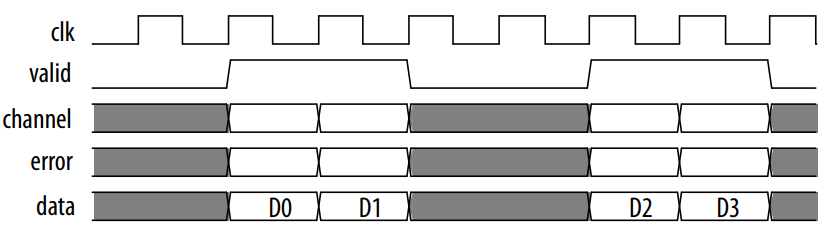
1. کلاک (فرستنده به گیرنده)
2. باس 8 بیتی داده (فرستنده به گیرنده)
3. سیگنال ready (گیرنده به فرستنده)
4. سیگنال valid (فرستنده به گیرنده)
5. سیگنال startofpacket (فرستنده به گیرنده)
6. سیگنال endofpacket (فرستنده به گیرنده)



پیاده سازی متداول این پروتکل به شکل بالاست، اما در اینجا ما به دو بخش ارور و چنل نیاز نداشتیم، لذا در پیاده‌سازی خود صرفا بخش‌های اجباری مورد نیاز را قرار دادیم.

روند کلی به ترتیب مراحل زیر می‌باشد:

1. فعال شدن سیگنال ready از سمت گیرنده
2. قرار دادن داده برروی باس داده که در اینجا 8 بیتی است
3. فعال کردن خروجی valid از سمته فرستنده
4. خواندن داده توسط گیرنده
5. همچنین برای تشخیص ابتدا و انتهای پکت‌های بیشتر از 8 بیت، در اولین 8 بیت سیگنال خروجی startofpacket و در آخرین نوبت سیگنال endofpacket فعال می‌شود.



شکل بالا به خوبی روند این پروتکل را نشان می‌دهد. صرفا توجه داشته باشید که ما از دو سیگنال اختیاری startofpacket و endofpacket برای ارسال بایت‌های متوالی دیتای خود استفاده کردیم. همانطور که ملاحظه می‌شود و پیش‌تر توضیح داده شد، این پروتکل داده را به صورت بخش‌های 8 بیتی متوالی و در کلاک‌های متوالی ارسال می‌کند.

در ادامه صرفا برخی از بخش‌های مهم را قرار داده و توضیح می‌دهیم:

    reg [7:0] send\_state;

    reg [7:0] receive\_state;

    assign out\_clk = clk;

    always @(posedge clk, negedge \_rst) begin

        if (!\_rst) begin

            send\_state <= 0;

            startofpacket\_out <= 0;

            endofpacket\_out <= 0;

            data\_out <= 0;

            valid\_out <= 0;

        end

        else begin

            // Sending A

            if (send\_state == 0) begin

                if (ready\_in == 1) begin

                    send\_state <= 1;

                    startofpacket\_out <= 1;

                    endofpacket\_out <= 0;

                    data\_out <= 1;

                    valid\_out <= 1;

                end

            end

            else if (send\_state == 1) begin

                if (ready\_in == 1) begin

                    send\_state <= 2;

                    startofpacket\_out <= 0;

                    endofpacket\_out <= 0;

                    data\_out <= A[31:24];

                    valid\_out <= 1;

                end

                else begin

                    send\_state <= 0;

                    startofpacket\_out <= 0;

                    endofpacket\_out <= 0;

                    data\_out <= 0;

                    valid\_out <= 0;

                end

            end

            else if (send\_state == 2) begin

                if (ready\_in == 1) begin

                    send\_state <= 3;

                    startofpacket\_out <= 0;

                    endofpacket\_out <= 0;

                    data\_out <= A[23:16];

                    valid\_out <= 1;

                end

                else begin

                    send\_state <= 0;

                    startofpacket\_out <= 0;

                    endofpacket\_out <= 0;

                    data\_out <= 0;

                    valid\_out <= 0;

                end

            end

            else if (send\_state == 3) begin

                if (ready\_in == 1) begin

                    send\_state <= 4;

                    startofpacket\_out <= 0;

                    endofpacket\_out <= 0;

                    data\_out <= A[15:8];

                    valid\_out <= 1;

                end

                else begin

                    send\_state <= 0;

                    startofpacket\_out <= 0;

                    endofpacket\_out <= 0;

                    data\_out <= 0;

                    valid\_out <= 0;

                end

            end

            else if (send\_state == 4) begin

                if (ready\_in == 1) begin

                    send\_state <= 5;

                    startofpacket\_out <= 0;

                    endofpacket\_out <= 1;

                    data\_out <= A[7:0];

                    valid\_out <= 1;

                end

                else begin

                    send\_state <= 0;

                    startofpacket\_out <= 0;

                    endofpacket\_out <= 0;

                    data\_out <= 0;

                    valid\_out <= 0;

                end

            end

            else if (send\_state == 5) begin

                if (ready\_in == 1) begin

                    send\_state <= 6;

                    startofpacket\_out <= 0;

                    endofpacket\_out <= 0;

                    data\_out <= 0;

                    valid\_out <= 0;

                end

                else begin

                    send\_state <= 0;

                    startofpacket\_out <= 0;

                    endofpacket\_out <= 0;

                    data\_out <= 0;

                    valid\_out <= 0;

                end

            end

همانطور که از بخش فوق پیداست، ارسال مقادیر به صورت استیت به استیت صورت می‌گیرد. این استیت در رجیستری با نام send\_state نگهداری شده و در هر مرحله بروز می‌شود.

always @(posedge clk, negedge \_rst) begin

        if (!\_rst) begin

            receive\_state <= 0;

            ready\_res <= 0;

            RES <= 0;

            ready\_out <= 1;

        end

        else begin

            // Reading Result

            if (receive\_state == 0) begin

                if (valid\_in == 1) begin

                    if (startofpacket\_in == 1) begin

                        receive\_state <= 1;

                        ready\_res <= 0;

                        RES[63:56] <= data\_in;

                        ready\_out <= 1;

                    end

                end

                else begin

                    receive\_state <= 0;

                    ready\_res <= 0;

                    RES <= 0;

                    ready\_out <= 1;

                end

            end

            if (receive\_state == 1) begin

                if (valid\_in == 1) begin

                    receive\_state <= 2;

                    ready\_res <= 0;

                    RES[55:48] <= data\_in;

                    ready\_out <= 1;

                end

                else begin

                    receive\_state <= 0;

                    ready\_res <= 0;

                    RES <= 0;

                    ready\_out <= 1;

                end

            end

            if (receive\_state == 2) begin

                if (valid\_in == 1) begin

                    receive\_state <= 3;

                    ready\_res <= 0;

                    RES[47:40] <= data\_in;

                    ready\_out <= 1;

                end

                else begin

                    receive\_state <= 0;

                    ready\_res <= 0;

                    RES <= 0;

                    ready\_out <= 1;

                end

            end

            if (receive\_state == 3) begin

                if (valid\_in == 1) begin

                    receive\_state <= 4;

                    ready\_res <= 0;

                    RES[39:32] <= data\_in;

                    ready\_out <= 1;

                end

                else begin

                    receive\_state <= 0;

                    ready\_res <= 0;

                    RES <= 0;

                    ready\_out <= 1;

                end

            end

            if (receive\_state == 4) begin

                if (valid\_in == 1) begin

                    receive\_state <= 5;

                    ready\_res <= 0;

                    RES[31:24] <= data\_in;

                    ready\_out <= 1;

                end

                else begin

                    receive\_state <= 0;

                    ready\_res <= 0;

                    RES <= 0;

                    ready\_out <= 1;

                end

            end

            if (receive\_state == 5) begin

                if (valid\_in == 1) begin

                    receive\_state <= 6;

                    ready\_res <= 0;

                    RES[23:16] <= data\_in;

                    ready\_out <= 1;

                end

                else begin

                    receive\_state <= 0;

                    ready\_res <= 0;

                    RES <= 0;

                    ready\_out <= 1;

                end

            end

            if (receive\_state == 6) begin

                if (valid\_in == 1) begin

                    receive\_state <= 7;

                    ready\_res <= 0;

                    RES[15:8] <= data\_in;

                    ready\_out <= 1;

                end

                else begin

                    receive\_state <= 0;

                    ready\_res <= 0;

                    RES <= 0;

                    ready\_out <= 1;

                end

            end

            if (receive\_state == 7) begin

                if (valid\_in == 1) begin

                    receive\_state <= 0;

                    ready\_res <= 1;

                    RES[7:0] <= data\_in;

                    ready\_out <= 1;

                end

                else begin

                    receive\_state <= 0;

                    ready\_res <= 0;

                    RES <= 0;

                    ready\_out <= 1;

                end

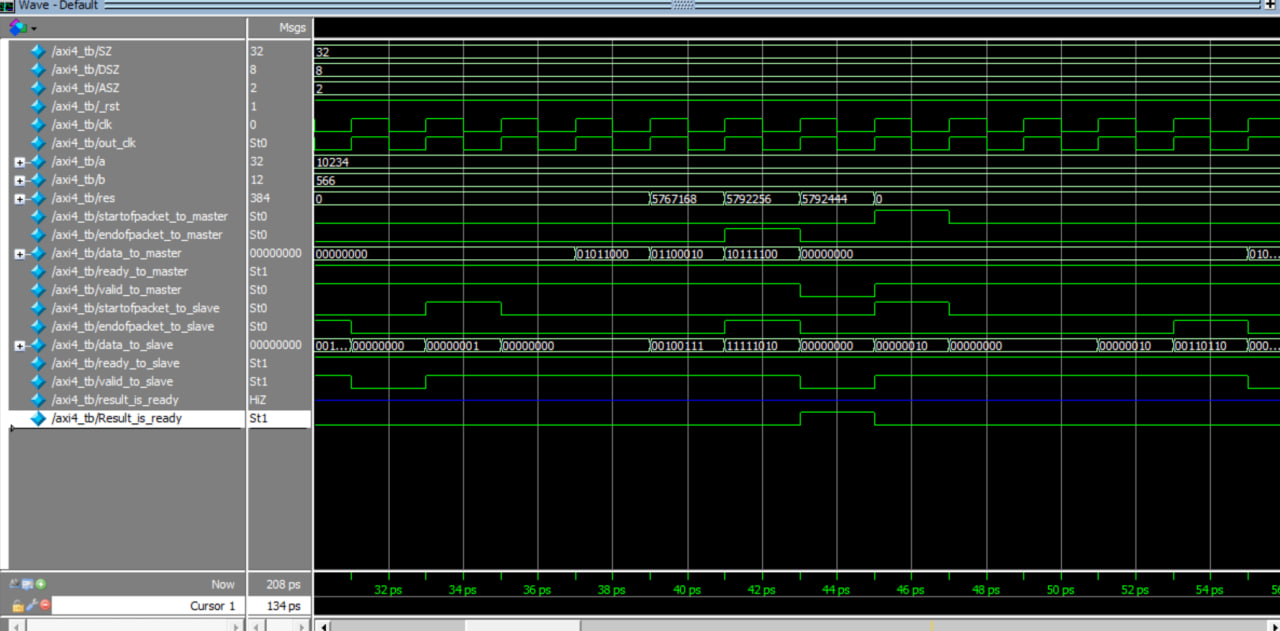
            end

        end

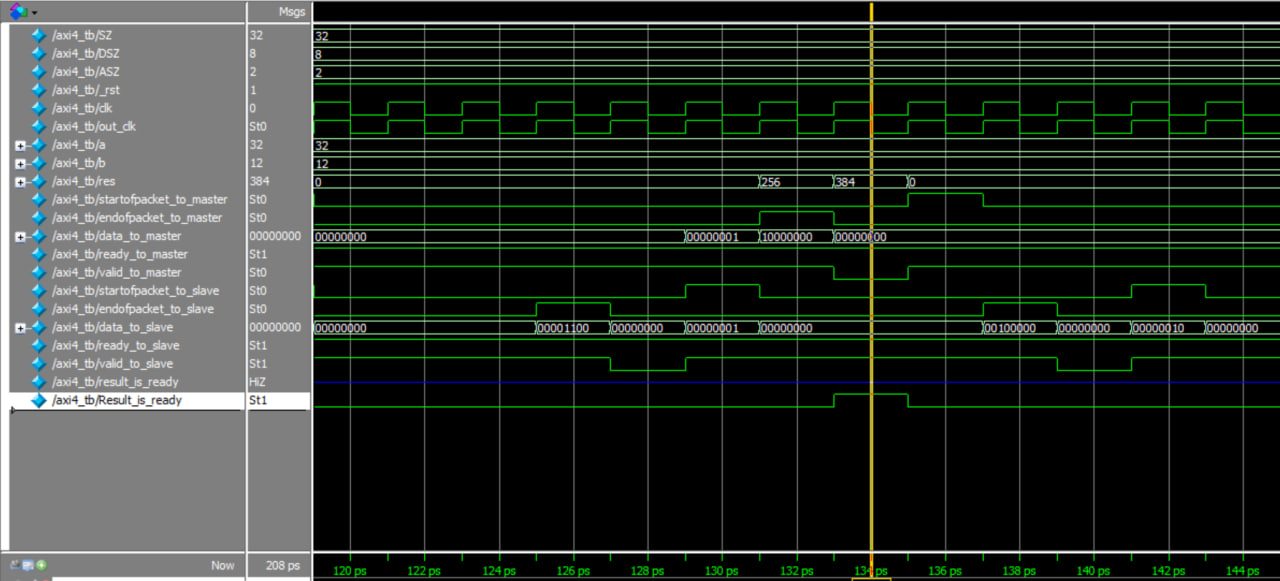
    end

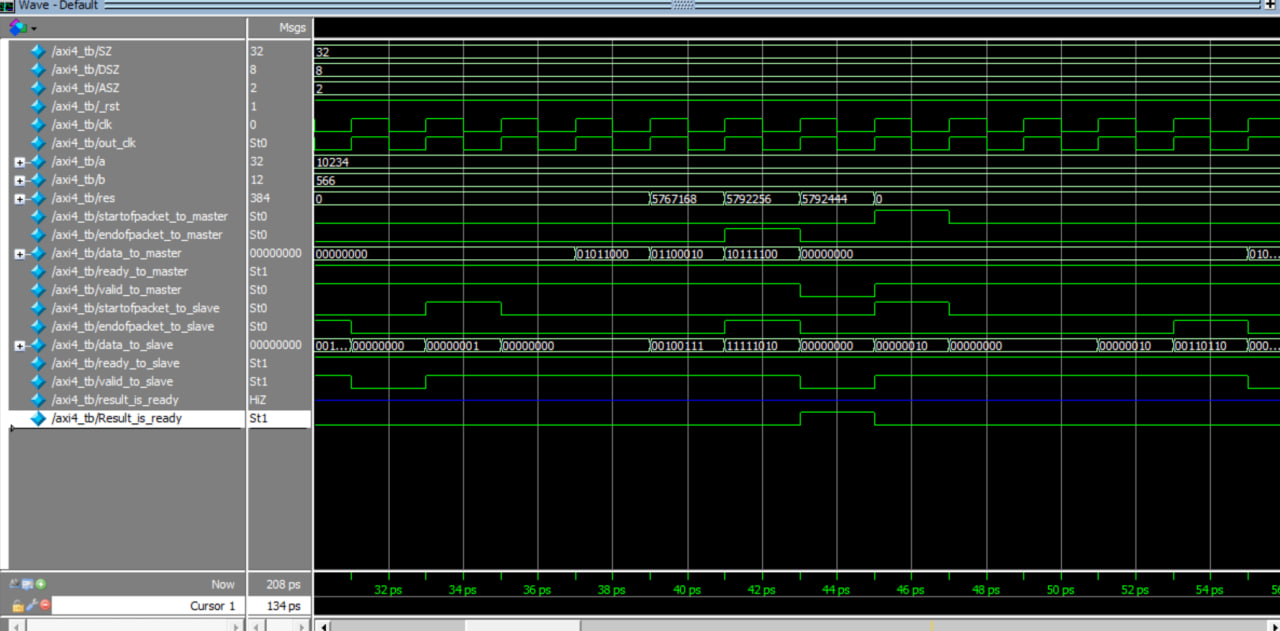
خواندن داده نیز به صورت بالا هندل شده است که به خوبی نمایانگر ساختار هندشیک گونه این اینترکانکشن است. تنها نکته حائز اهمیت این است که این کدها متعلق به مستر بودند. همانطور که گفتیم این پروتکل رویکرد تک جهته‌ای دارد. لذا پیاده سازی صورت گرفته در اسلیو عینا به همین شکل است تا امکان ارسال اسلیو به مستر داده را فراهم سازد. (همچنین ساختار دریافت پیام نیز مشترک است، و در اسلیو ساختاری عینا مشابه ساختار بالا داریم.)

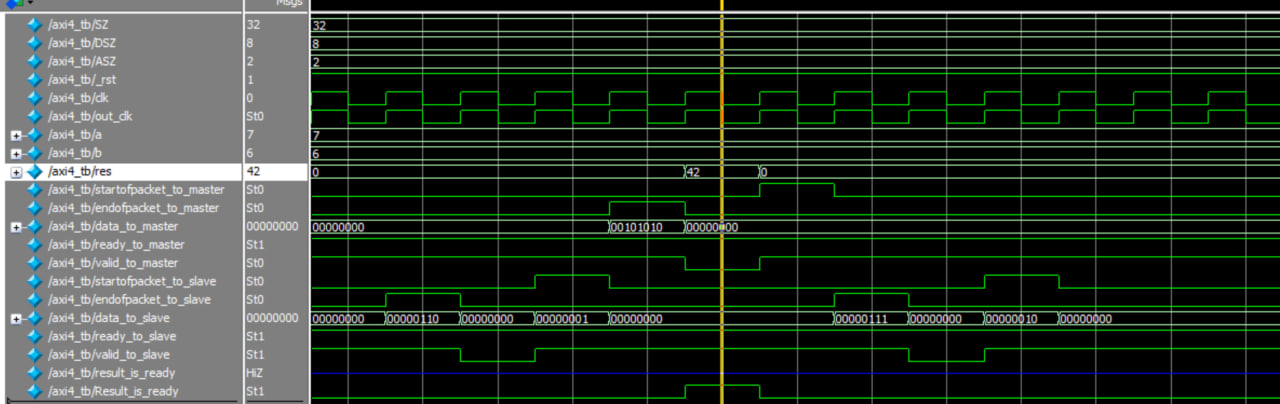
مطابق شکل زیر، این پروتکل در زمان 43 اولین ریزالت خود را محاسبه می‌کند. با توجه به اینکه از زمان 8 از حالت ریست خارج شده و در حال پردازش است، متوجه می‌شویم زمان مورد نیاز بری انجام این محاسبه در بدترین حالت 35 واحد طول کشید.



**در ادامه چند نمونه عملیات ضرب صورت گرفته به کمک این اینترکانکشن را بررسی می‌کنیم.**







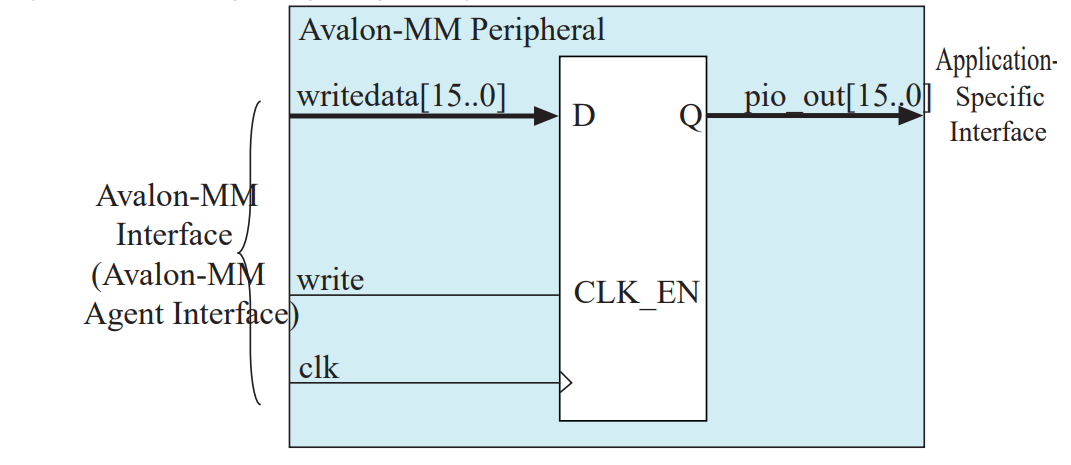
1. **Avalon memory-mapped**

این پروتکل بشدت شبیه پروتکل قبلی است با این تفاوت که دیگر یکطرفه نبوده و خودش ذاتا ویزگی‌های مورد نیاز ما را هندل کرده است. برای این پروتکل از دو داک [3.2. Avalon® Memory Mapped Interface Signal Roles](https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/memory-mapped-interface-signal-roles.html) و [mnl\_avalon\_spec-683091-667068.pdf](../../../../../../C:/Users/surface/Downloads/mnl_avalon_spec-683091-667068.pdf) استفاده شده است. روند کلی این پروتکل نیز اینگونه است که داده با توجه به سیگنال عملیات خورده شده برروی باس متناسب قرار می‌گیرد. برای مثال برای خواندن داده، آدرس روی باس آدرس قرار می‎گیرد و پس از فعال شدن سیگنال read، مقدار خوانده شده در باس read قرار داده می‌شود. برای نوشتن نیز، داده در حال نوشته شدن برروی باس نوشتن قرار گرفته و پس از فعال شدن سیگنال نوشتن، داده نوشته می‌شود.

پیاده سازی مستر و اسلیو مربوطه در فایل avalon\_mm\_slave\_wrapper.sv قرار دارد. تست بنچ مربوط به این فایل نیز با نام avalon\_mm\_tb.sv قرار گرفته است.

سیگنال‌های موجود از سمت مستر به شرح ذیل است:

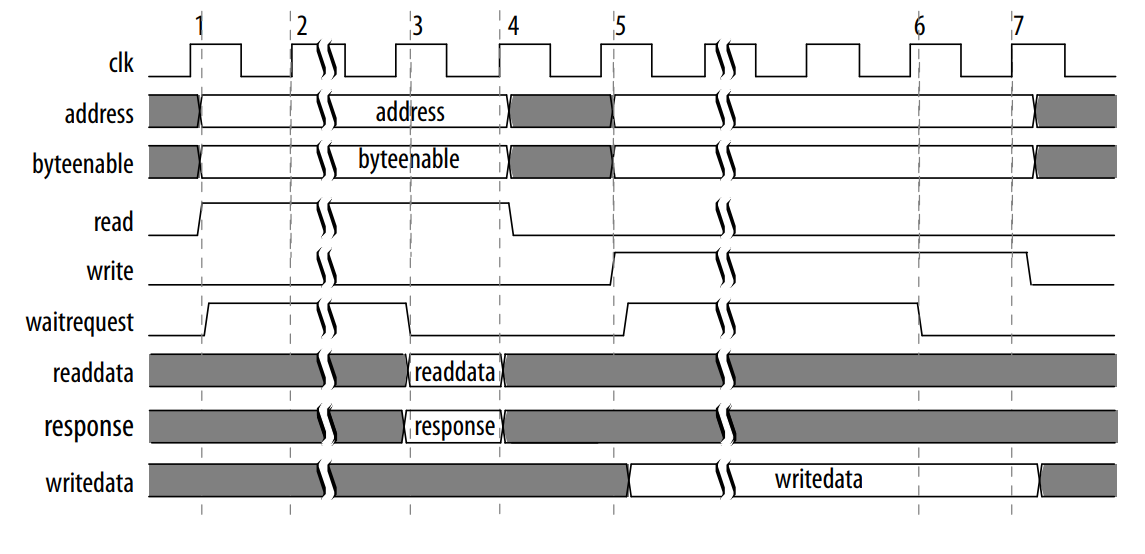
1. کلاک
2. آدرس (در اینجا 4 بیتی گرفتیم)
3. read\_data 16 بیتی برای خواندن داده از اسلیو
4. write\_data 16 بیتی برای نوشتن داده در اسلیو
5. سیگنال write برای نوشتن
6. سیگنال read برای خواندن



شمای پیاده سازی مینیمالیستی پروتکل مذکور به شکل بالاست که ما بنا به نیاز هم عرض باس را کم کرده و هم از سیگنال read بهره بردیم.

روند کلی استفاده از این پروتکل توسط ما:

1. نوشتن مقدار A و B در آردس مپ شده به این مقادیر در ماژول ضرب کننده. در واقع با توجه به 16 بیتی بودن باس‌ نوشتن، این کار را برای هرکدام به این صورت باید انجام دهیم که نیمه بالا و پایین را جدا ارسال کنیم.
2. خواندن نتیجه از نقاط مپ شده به نتیجه در ماژول ضرب کننده. مجددا با توجه به طول 16 بیتی باس خواندن، این مقدار را به صورت 16 بیت 16 بیت و در 4 مرحله باید بخوانیم.



روند کلی این پروتکل در شکل بالا قرار داده شده است. صرفا توجه داشته باشید که از برخی از سیگنال‌های اختیاری این شکل استفاده نکرده‌ایم.

محتویات این فایل به شکل زیر است که در آن هر مقدار به تعدادی خانه حافظه پنداشته شده و مانند خانه‌های حافظه با آن‌ها برخورد می‌شود.

module avalon\_mm\_slave\_wrapper #(

    parameter int SZ = 32

) (

    input \_rst,

    input clk,

    input [3:0] addr,  // A1 A0 B1 B0 C3 C2 C1 C0

    input read,

    input write,

    input [15:0] write\_data,

    output reg [15:0] read\_data

);

    reg  [  SZ-1:0] areg;

    reg  [  SZ-1:0] breg;

    wire [2\*SZ-1:0] res;

    wire ready, start;

    mult #(

        .SZ(SZ)

    ) main\_module (

        .a(areg),

        .b(breg),

        .res(res),

        .start(start),

        .\_rst(\_rst),

        .clk(clk),

        .ready(ready)

    );

    always @(posedge clk, negedge \_rst) begin

        if (!\_rst) begin

            areg <= 0;

            breg <= 0;

            read\_data <= 0;

        end

        else begin

            if (read) begin

                if (addr == 4) begin

                    read\_data <= res[15:0];

                end

                else if (addr == 5) begin

                    read\_data <= res[31:16];

                end

                else if (addr == 6) begin

                    read\_data <= res[47:32];

                end

                else if (addr == 7) begin

                    read\_data <= res[63:48];

                end

                else begin

                    read\_data <= 0;

                end

            end

            else if (write) begin

                if (addr == 0) begin

                    areg[15:0] <= write\_data;

                end

                else if (addr == 1) begin

                    areg[31:16] <= write\_data;

                end

                if (addr == 2) begin

                    breg[15:0] <= write\_data;

                end

                else if (addr == 3) begin

                    breg[31:16] <= write\_data;

                end

            end

        end

    end

endmodule

module avalon\_mm\_master\_wrapper #(

    parameter int SZ = 32

) (

    input \_rst,

    input clk,

    input [31:0] A,

    input [31:0] B,

    input [15:0] read\_data,

    output out\_clk,

    output reg [63:0] res,

    output reg [3:0] addr,  // A1 A0 B1 B0 C3 C2 C1 C0

    output reg read,

    output reg write,

    output reg [15:0] write\_data

);

    assign out\_clk = clk;

    reg [7:0] state;

    always @(posedge clk, negedge \_rst) begin

        if (!\_rst) begin

            state <= 0;

            addr <= 0;

            read <= 0;

            write <= 0;

            write\_data <= 0;

            res <= 0;

        end

        else begin

            if (state == 0) begin

                state <= 1;

                addr <= 0;

                read <= 0;

                write <= 1;

                write\_data <= A[15:0];

            end

            else if (state == 1) begin

                state <= 2;

                addr <= 1;

                read <= 0;

                write <= 1;

                write\_data <= A[31:16];

            end

            else if (state == 2) begin

                state <= 3;

                addr <= 2;

                read <= 0;

                write <= 1;

                write\_data <= B[15:0];

            end

            else if (state == 3) begin

                state <= 4;

                addr <= 3;

                read <= 0;

                write <= 1;

                write\_data <= B[31:16];

            end

            else if (state == 4) begin

                state <= 5;

                addr  <= 4;

                read  <= 1;

                write <= 0;

            end

            else if (state == 5) begin

                state <= 6;

                addr  <= 5;

                read  <= 1;

                write <= 0;

            end

            else if (state == 6) begin

                state <= 7;

                addr <= 6;

                read <= 1;

                write <= 0;

                res[15:0] <= read\_data;

`ifdef VERBOSE

                $display("%b", res);

`endif

            end

            else if (state == 7) begin

                state <= 8;

                addr <= 7;

                read <= 1;

                write <= 0;

                res[31:16] <= read\_data;

`ifdef VERBOSE

                $display("%b", res);

`endif

            end

            else if (state == 8) begin

                state <= 9;

                addr <= 0;

                read <= 0;

                write <= 0;

                res[47:32] <= read\_data;

`ifdef VERBOSE

                $display("%b, %b", res, read\_data);

`endif

            end

            else if (state == 9) begin

                state <= 0;

                addr <= 0;

                read <= 0;

                write <= 0;

                res[63:48] <= read\_data;

`ifdef VERBOSE

                $display("%b", res);

`endif

            end

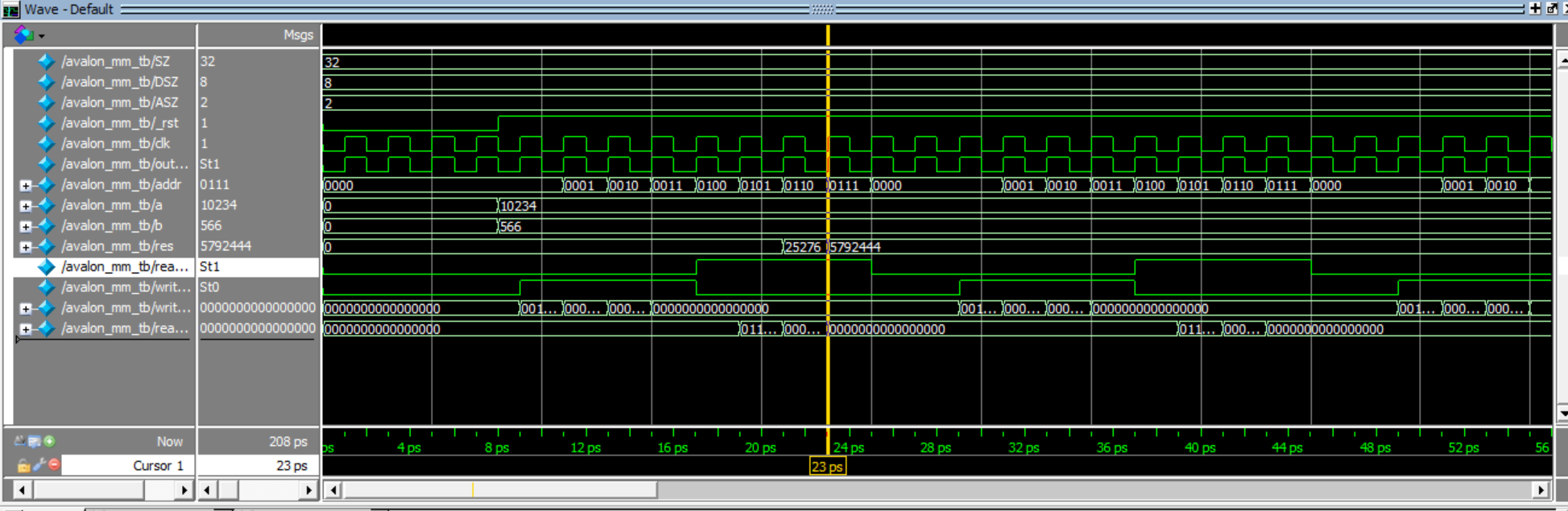
        end

    end

endmodule

همانطور که ملاحظه می‌شود، طرز برخورد مانند یک حافظه word-addressable با طول کلمه 16 بیت است.

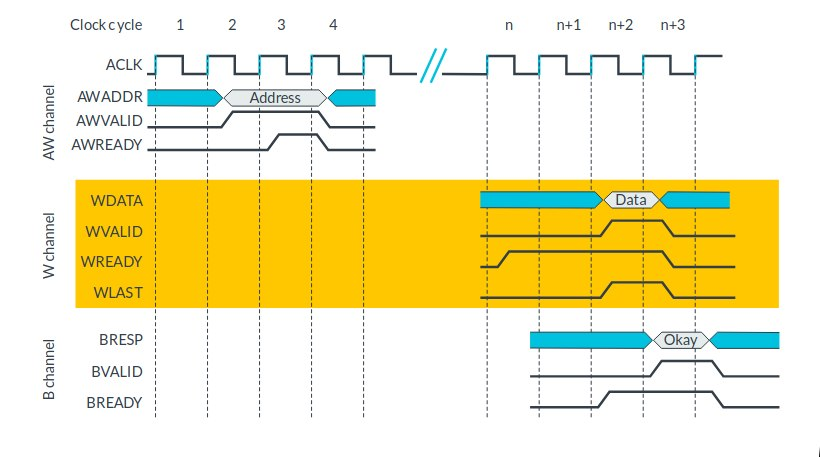
خروجی ویوفرم مربوط به تست بنچ این اینترکانشکن را نیز ببینیم:

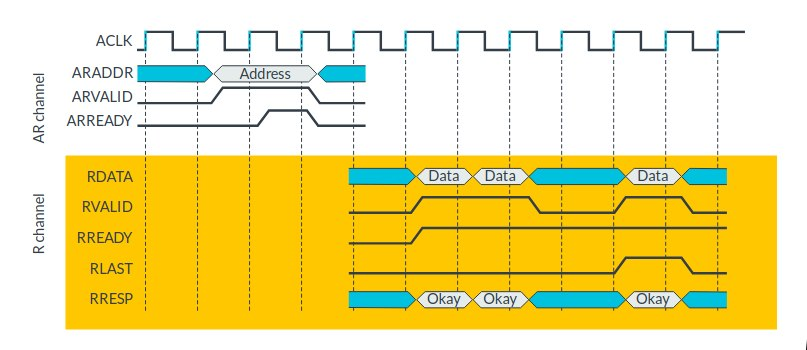


همانطور که ملاحظه $display($time, " | master | ", "res : ", res);می‌شود، این پروتکل در زمان 23 اولین خروجی خود را می‌دهد، که با توجه به شروع از 8، یعنی عملا 15 واحد زمانی تا اولین خروجی خود زمان نیاز داشت.

1. **AXI4**

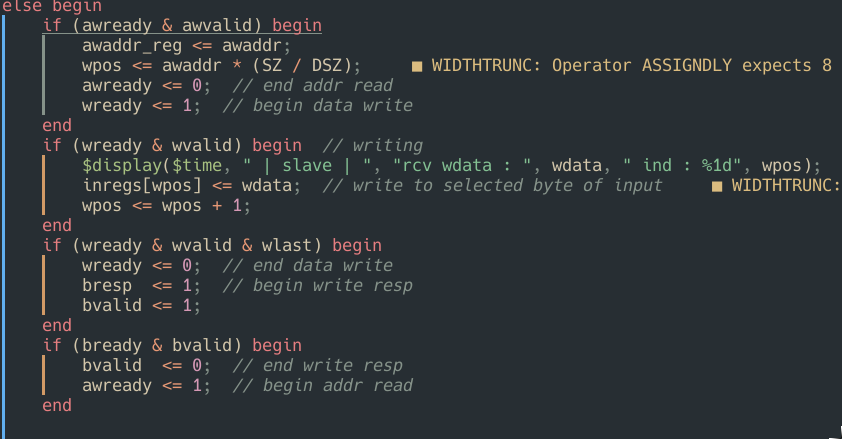
به طور کلی پروتوکل های خانواده AXI از تجمیع تعدادی فرایند HandShaking تشکیل شده است. AXI4 به طور خاص از ۵ کانال استفاده میکند که آنهارا تحت عناوین AW (آدرس نوشتن) W(نوشتن) B (پاسخ نوشتن) AR(آدرس خواندن) R(خواندن) یاد میکند. برای پیاده‌سازی این پروتوکل نیز باید تحت ۲ ماژول، یکی برای master و دیگری برای slave استفاده کنیم.



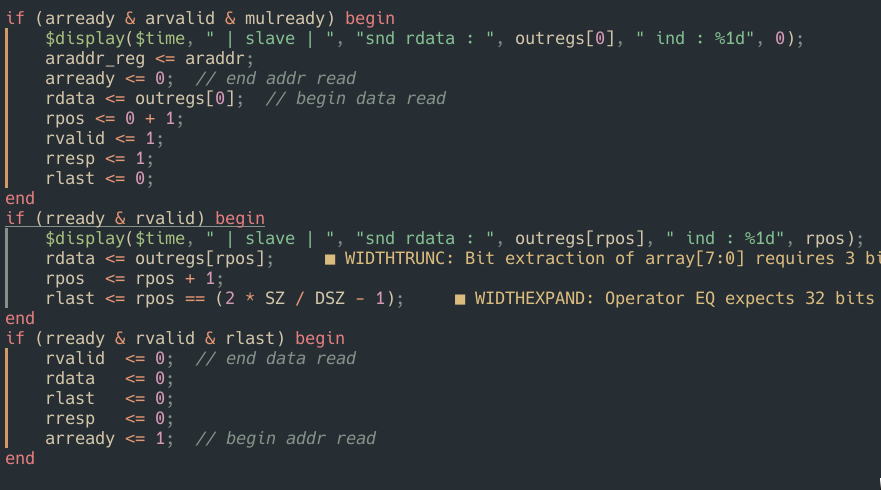


نمونه سیگنال های درخواست های نوشتن و خواندن در AXI4.

پیاده‌سازی این پروتوکل ها را در زیر مشاهده میکنید :



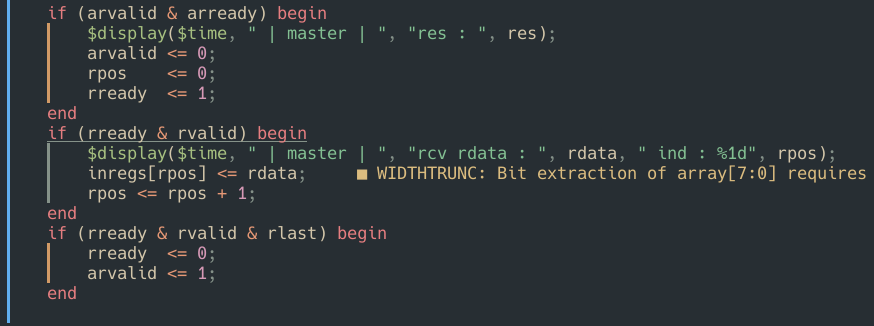
پیاده‌سازی مکانیزم HandShaking برای دریافت داده از ارباب.



پیاده‌سازی مکانیزم HandShaking برای فرستادن داده به ارباب.

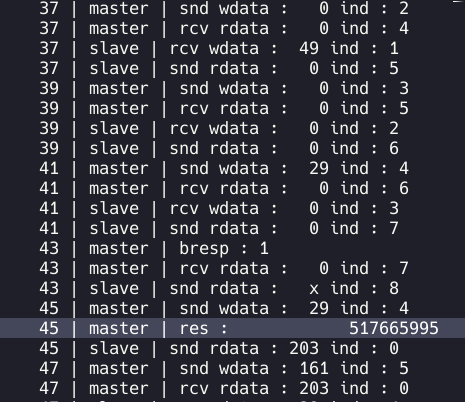


مکانیزم HandShaking برای فرستادن ورودی ها به برده.



مکانیزم HandShaking برای خواندن نتیجه از برده.

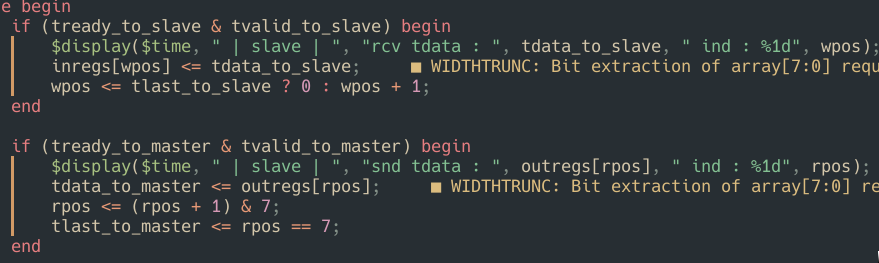
در نهایت نتیجه آزمون این پروتوکل به شکل زیر است :



همانطور که مشاهده میکنید، کل تاخیر رفت و برگشت این پروتوکل به اندازه ۴۵ است که تقریبا نیمی برای فرستادن مقادیر و نیم دیگر برای خواندن نتیجه صرف شده است.

1. **AXI4 stream**

این پروتوکل نیز بسیار شبیه AXI4 عادی است با این تفاوت که این پروتوکل دیگر دوطرفه نیست و تنها یک سویه است. علاوه بر این تنها یک کانال تحت عنوان T(ارسال) داریم که سیگنال های اصلی آن، Tdata, Tvalid, Tready و Tlast هستند. در اینجا با توجه به اینکه میخواهیم به ضرب کننده داده هارا فرستاده و نتیجه را بگیریم، به ۲ تا از این کانال‌ها نیاز داریم که یکی را از پردازنده به ماژول و دیگری را از ماژول به پردازنده در نظر میگیریم. حال باقی مراحل مشابه قبل است.



به طور کلی مکانیزم ارتباطی این پروتوکل به این شکل در ارباب و برده پیاده شده است. به طوری که به طور پیوسته هر طرف برای طرف دیگر مقادیر مورد نظر خود را میفرستد.

در نهایت نتیجه آزمون این پروتوکل نیز به شکل زیر است :

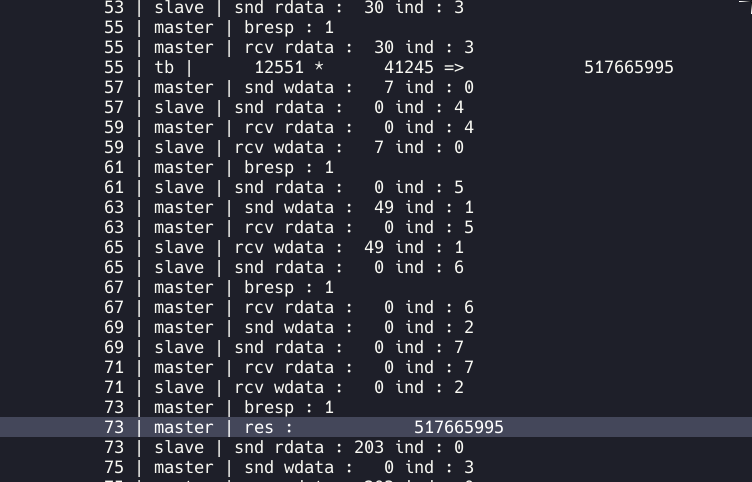


همانطور که مشاهده میکنید، این پروتوکل نیز در زمان ۵۹ به نتیجه رسیده است. البته مقداری زود تر نیز در زمان ۴۳ به نتیجه رسیده بود اما به دلایلی از جمله استفاده از ورودی ناقص برای محاسبه، به نتیجه اشتباه رسیده بود.

1. **AXI4 lite**

این پروتوکل دقیقاً مانند AXI است تنها با این تفاوت که مدارات این پروتوکل بسیار ساده‌تر از AXI هستند و از روش burst پشتیبانی نمیکند. با توجه به این تغییر سیگنال های rlast و wlast را ندارد. با توجه به همین مساله، صرفاً باید تغییر جزئی به سیستم آدرس دهی خود دهیم و همه چیز به درستی کار خواهد کرد.

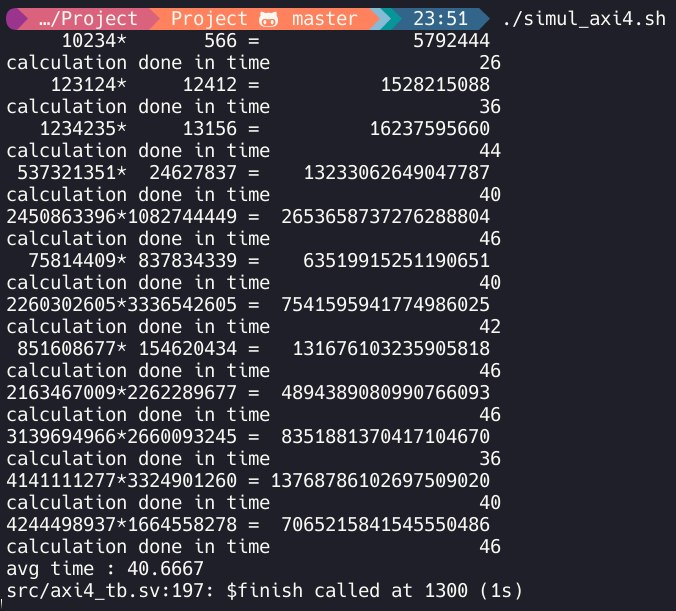
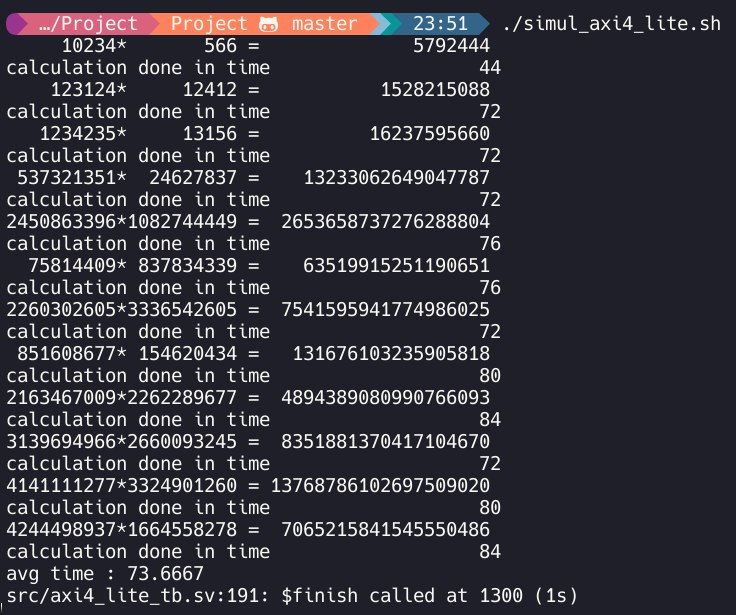
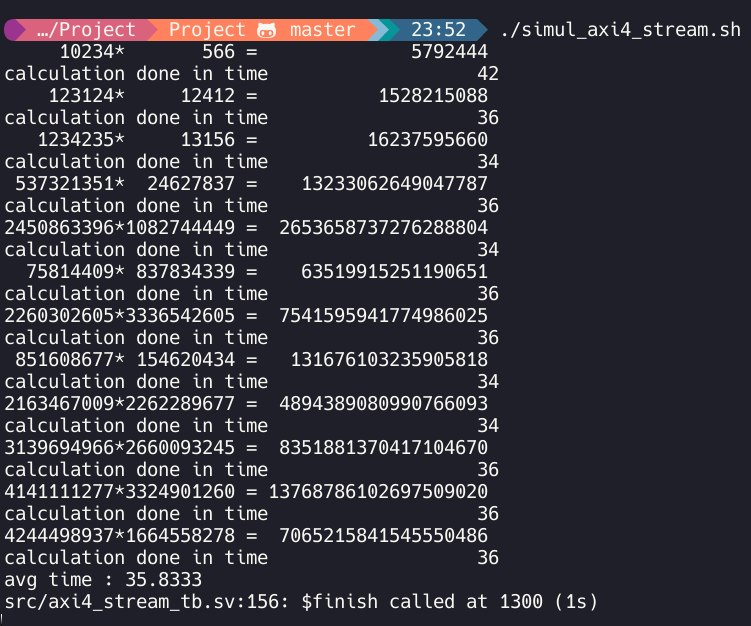
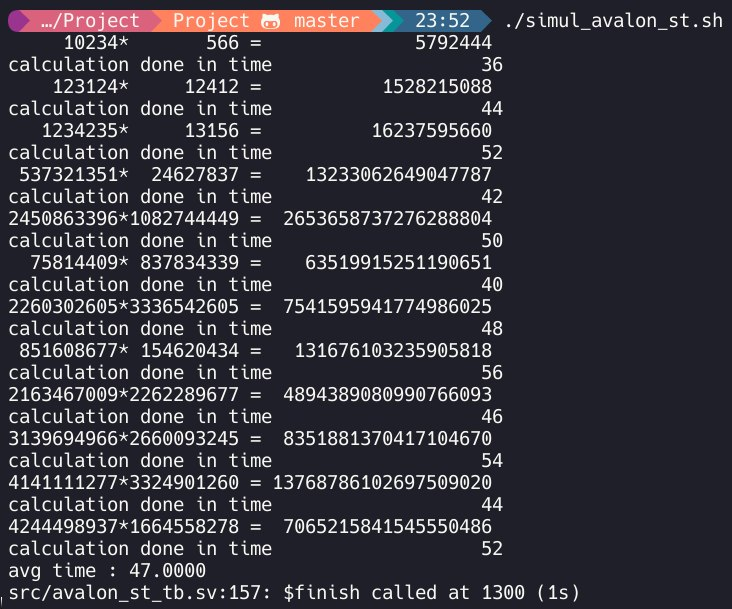
نتیجه آزمون این پروتوکل نیز به شکل زیر است :

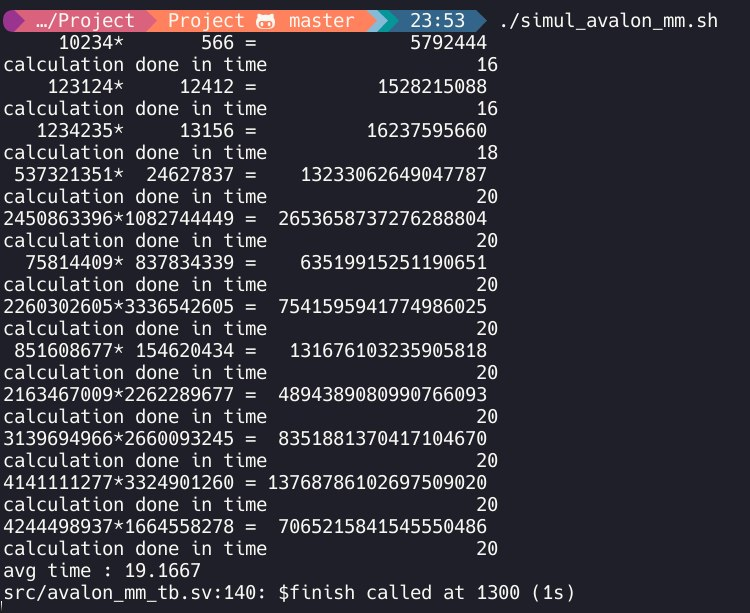


همانطور که در نتیجه میبینید، این آزمون نیز در زمان ۷۳ به نتیجه رسیده که با توجه به نبود قابلیت burst پیشبینی میشد. به این ترتیب سربار و تاخیر فرستادن پیام ها بالا رفته است.

# نتیجه گیری نهایی

برای این کار تست بنچ هارا کمی تغییر میدهیم که اولین زمانی که جواب درست را حساب کردند گزارش کنند. به این شکل، نتیجه آزمایش این پروتوکل ها به شکل زیر خواهد بود :



با توجه به این که زمان برابر با Fmax \* clk است و Fmax وابسته به مدار پیچیده ضرب ما است. در این صورت میتوانیم این پروتوکل هارا در این شرایط محدود شده مطابق تعداد کلاک های مورد نیازشان مقایسه کرد که نتایج به شرح زیر است :

avalon\_mm : 19.16 clk

axi4\_stream : 35.83 clk

axi4 : 40.66 clk

avalon\_st : 47 clk

axi4\_lite : 73.66 clk

با توجه به این آزمون، در شرایط آزمایش ما avalon\_mm بیشترین کارایی را ثبت نمود.