بسمه تعالی

پروژه امتیازی درس طراحی سیستم های دیجیتال

ترم بهار 1402-1403

دانشجو: فرید محمودزاده
شماره دانشجویی: 401106493
استاد: دکتر امین فصحتی

هدف پروژه

در این پروژه قصد داریم با استفاده از زبان برنامه نویسی Verilog یک پردازنده آرایه‌ای 512 بیتی طراحی کنیم که از 3 بخش زیر تشکیل شده است.

- یک رجیستر فایل با قابلیت ذخیره سازی 4 آرایه 512 بیتی با نام های A1 تا A4
- یک واحد ریاضی که قابلیت انجام ضرب و جمع را دارا باشد. ورودي این واحد ریاضی A1 و A2 و خروجی کم ارزش آن در A3 و پر ارزش آن در A4 است.
- دارای یک حافظه با عمق 512 و عرض 32 بیت. این پردازنده امکان بارگزاری/ذخیره‌سازی 16 خانه پشت سر هم از حافظه را دارا است.

همچنین این پردازنده دارای 4 دستور زیر است.

- بارگزاری از حافظه در یکی از ثبات‌ها
- ذخیره‌سازی از یکی از ثبات‌ها به حافظه
- جمع واحد ریاضی
- ضرب واحد ریاضی

وریلاگ

از آن‌جا که این برنامه از 3 بخش تشکیل شده ابتدا ماژول مربوطه به هر بخش را پیاده‌سازی و سپس با استفاده از بخش‌های مختلف آن ماژول کلی را پیاده می‌کنیم.

در آخر برای اطمینان از صحت عملکرد مدار، یک ماژول تحریک برای آن طرح می‌کنیم.

Register File:

شکل کلی ماژول به صورت زیر است، در ادامه آن را بررسی می‌کنیم.

```
module REGISTER_FILE (
        input clk, reset,
        input [511:0] data_in_1, data_in_2,
        input [1:0] r_reg,
        input [1:0] w_reg_1, w_reg_2,
        input w_enable_1, w_enable_2,
        output [511:0] data_out,
        output [511:0] A1, A2, A3, A4
    );

    reg [511:0] regs [0:3];
    assign A1 = regs[0];
    assign A2 = regs[1];
    assign A3 = regs[2];
    assign A4 = regs[3];
    assign data_out = regs[r_reg];

    always @(negedge clk) begin
        if(reset) begin
```

```
            regs[0] <= 0;
            regs[1] <= 0;
            regs[2] <= 0;
            regs[3] <= 0;
        end
        else begin
            if (w_enable_1)
                regs[w_reg_1] <= data_in_1;
            if (w_enable_2)
                regs[w_reg_2] <= data_in_2;
        end
    end
endmodule
```

در ورودی ها و خروجی ها که در ابتدای ماژول مشخص‌اند دقت داشته باشید که چون در پردازنده‌مان 4 ثبات داریم، به 2 بیت برای مشخص‌کردن ثبات مورد نظر نیاز داریم و به همین دلیل r_reg (ثباتی که می‌خواهیم از آن بخوانیم) و w_reg_1 و w_reg_2 (ثبات‌هایی که می‌خواهیم روی آن‌ها بنویسیم) 2 بیتی هستند و به ترتیب این دو بیت از 0 تا 3، ثبات های A1 تا A4 را مشخص می‌کنند.

نوشتن در ثبات‌ها و همچنین ریست کردنشان (در صورت فعال بودن ورودی کنترلی reset) به صورت سنکرون و با لبه‌ی پایین رونده کلاک انجام می‌شود.

خواندن از ثبات‌ها به صورت آسنکرون انجام می‌شود. محتویات همه‌ی ثبات‌ها و همچنین محتوای یک ثبات با آدرس‌دهی از طریق r_reg همواره در خروجی حضور دارند.

از آن‌جا که برای دستورات add و mult به نوشتن در 2 ثبات احتیاج داریم، این قابلیت را با قرار دادن دو داده ورودی پیاده‌سازی کردیم.

ALU:

شکل کلی ماژول به صورت زیر است، در ادامه آن را بررسی می‌کنیم.

```
module ALU (
        input [511:0] data_in_1, data_in_2,
        input operator,
        output [1023:0] data_out
    );

    reg [1023:0] S;
    assign data_out = S;
    integer i;

    always @(*) begin
        if (operator) begin
            for (i = 0; i < 16; i = i + 1) begin
                S[64*i +: 64] <= data_in_1[32*i +: 32] + data_in_2[32*i +:
32];
            end
        end
        else begin
            for (i = 0; i < 16; i = i + 1) begin
                S[64*i +: 64] <= data_in_1[32*i +: 32] * data_in_2[32*i +:
32];
            end
        end
    end
endmodule
```

در این طراحی از آن‌جا که می‌خواهیم در این ALU از فقط دو عمل ضرب و جمع پشتیبانی کنیم، ورودی بیتی operator را قرار می‌دهیم تا با توجه به آن عملیات مورد نظر انتخاب شود. اگر operator برابر با 1 بود جمع انجام می‌شود و اگر 0 بود ضرب انجام می‌شود.

در این ALU هر ورودی 512 بیتی را به عنوان بردارهایی دارای 16 عدد 32 بیتی در نظر گرفته می‌شود و با توجه به operator عملیات مورد نظر میان مولفه‌های متناظر ورودی‌ها انجام می‌شود که حاصلش در مولفه متناظر خروجی نوشته می‌شود.

حاصل هر عملیات روی هر عدد 32 بیتی از بردارها یک عدد 64 بیتی است بنابراین خروجی یک بردار 1024 بیتی می‌شود.

Memory:

شکل کلی ماژول به صورت زیر است، در ادامه آن را بررسی می‌کنیم.

```
module MEMORY(
        input clk, reset,
        input [511:0] data_in,
        input [8:0] mem_addr,
        input w_enable,
        output [511:0] data_out
    );

    reg [31:0] mem [0:511];
    reg [511:0] memory_out;

    assign data_out = memory_out;

    integer i, j;

    always @(*) begin
```

```
    for (i = 0; i < 16; i = i + 1) begin
        if (mem_addr + i < 512)
            memory_out[32*i +: 32] = mem[mem_addr + i];
    end
end

always @(negedge clk) begin
    if(reset) begin
        for (j = 0; j < 12; j = j + 1) begin
            mem[j] <= j;
            mem[j + 16] <= 2000 + j;
        end
        mem[12] <= 32'hFFFFFFFF;
        mem[28] <= 32'hFFFFFFFF;

        mem[13] <= 32'hFF0000FF;
        mem[29] <= 32'h00FFFF00;

        mem[14] <= 32'hFF000000;
        mem[30] <= 32'h10000000;

        mem[15] <= 32'h1FFFFFFF;
        mem[31] <= 32'hEFFFFFFF;
    end
    else begin
        if (w_enable) begin
            for (j = 0; j < 16; j = j + 1) begin
```

```
                    if (mem_addr + j < 512)
                        mem[mem_addr + j] <= data_in[32*j +: 32];
                end
            end
        end
    end
endmodule
```

دقت داشته باشید که چون در تعداد خانه‌های حافظه 512 تا است به 9 بیت برای مشخص‌کردن خانه‌ی مورد نظر نیاز داریم و به همین دلیل mem_addr، دارای 9 بیت است.

نوشتن در خانه‌های حافظه به صورت سنکرون و با لبه‌ی پایین رونده کلاک انجام می‌شود.

همچنین reset هم به صورت سنکرون فعال می‌شود و برای مقداردهی اولیه به صورت بالا آن‌را فعال می‌کنیم که برای تست کردن پردازنده آن ها را قرار دادیم.

مقادیر اولیه در for مقادیر معمولی و مقادیر بعدی در خانه‌های 12 تا 15 و 28 تا 31 مقادیر مرزی هستند.

خواندن و نوشتن در حافظه به این صورت انجام می‌شود که ابتدا خانه‌ی مورد نظر توسط mem_addr انتخاب می‌شود و سپس عملیات مورد نظر برای آن خانه و 15 خانه‌ی جلوتر از آن (در مجموع 16 خانه) انجام می‌شود. دقت کنید که اگر در این بین، به آخرین خانه‌ی حافظه برسیم، عملیات خواندن یا نوشتن را همانجا به اتمام می‌رسانیم.

Vector Processor:

حالا که بخش‌های مختلف مورد نیاز برای طراحی پردازنده آرایه‌ای را پیاده‌سازی کردیم، وقت آن است که با برقراری ارتباط میان این ماژول‌ها، ماژول اصلی پردازنده را طرح کنیم.

شکل کلی ماژول را می‌توانید در صفحات بعد بعد مشاهده کنید، در ادامه به بررسی آن می‌پردازیم.

```
module VECTOR_PROCESSOR (
        input clk, reset,
        input [12:0] instruct,
        output [511:0] A1, A2, A3, A4
    );
    reg [511:0] rf_in_1, rf_in_2;
    reg [1:0] r_reg;
    reg [1:0] w_reg_1, w_reg_2;
    reg rf_w_enable_1, rf_w_enable_2;
    wire [511:0] rf_out;
    wire [511:0] A1_out, A2_out, A3_out, A4_out;

    reg [511:0] ALU_in_1, ALU_in_2;
    reg operation;
    wire [1023:0] ALU_out;

    reg [511:0] data_in;
    reg [8:0] mem_address;
    reg mem_w_enable;
    wire [511:0] data_out;
    integer i;

    REGISTER_FILE register_file (clk, reset,
                                 rf_in_1, rf_in_2,
                                 r_reg,
                                 w_reg_1, w_reg_2,
                                 rf_w_enable_1, rf_w_enable_2,
```

```
                                rf_out,
                                A1_out, A2_out, A3_out, A4_out
    );
ALU alu (ALU_in_1, ALU_in_2,
            operation,
            ALU_out
    );
MEMORY mem (clk, reset,
                data_in,
                mem_address,
                mem_w_enable,
                data_out
    );

assign A1 = A1_out;
assign A2 = A2_out;
assign A3 = A3_out;
assign A4 = A4_out;

always @(posedge clk) begin
    #5;
    if (~instruct[12]) begin
        rf_w_enable_1 <= ~instruct[11];
        rf_w_enable_2 <= 0;
        mem_address <= instruct[8:0];
        mem_w_enable <= instruct[11];
        if(instruct[11]) begin
```

```
                r_reg <= instruct[10:9];
                #1
                data_in <= rf_out;
            end
            else begin
                w_reg_1 <= instruct[10:9];
                #5
                rf_in_1 <= data_out;
            end
        end
        else begin
            operation = instruct[11];
            mem_w_enable <= 0;
            rf_w_enable_1 <= 1;
            rf_w_enable_2 <= 1;
            w_reg_1 <= 2'b10;
            w_reg_2 <= 2'b11;
            ALU_in_1 <= A1_out;
            ALU_in_2 <= A2_out;
            #5
            for(i = 0; i < 16; i = i + 1) begin
                rf_in_1[32*i +: 32] <= ALU_out[64*i +: 32];
                rf_in_2[32*i +: 32] <= ALU_out[64*i+32 +: 32];
            end
        end
    end
endmodule
```

در ورودی‌ها دقت کنید که instruct دستور مورد نظر ما است که 2 بیت پرارزش آن نشان دهنده آن است که از بین 4 دستور مورد نظر کدام یک انجام شود (00 برای load، و 01 برای store، و 10 برای ضرب، و 11 برای جمع)، 2 بیت بعد از آن نشان دهنده ثباتی است که می‌خواهیم دستور را روی آن انجام دهیم و 9 بیت کم ارزش هم نشان دهنده آدرس حافظه‌ای است که می‌خواهیم دستور را روی آن انجام دهیم و در مجموع دستورها 13 بیتی می‌شوند.

همچنین خروجی‌های A1 تا A4 برای نمایش در تست بنچ قرار داده شده‌اند.

در این ماژول دقت کنید که عملیات های کنترلی به صورت سنکرون و در لبه بالارونده کلاک انجام می‌شوند، به طوری که در ابتدا با بیت پرارزش instruct تصمیم می‌گیرد نوع دستور محاسباتی است (جمع و ضرب) یا از نوع load و store است.

اگر محاسباتی باشد، operation ورودی ALU برابر با دومین بیت پرارزش می‌شود؛ در این دستورات چون با حافظه کاری نداریم، mem_w_enable را برای حافظه صفر می‌کنیم و همچنین چون در این دستورات باید بر دو ثبات A3 و A4 بنویسیم، rf_w_enable_1 و rf_w_enable_2 برای رجیستر فایل 1 می‌شوند و در w_reg_1 و w_reg_2 به ترتیب 10 و 11 به نشانه این دو ثبات قرار می‌گیرد و مقادیر ALU_in برابر با مقادیر ثبات‌های 1 و 2 یعنی A1_out و A2_out می‌شوند.

پس از این مقداردهی‌ها کمی تاخیر داریم برای پایدار شدن خروجی ALU و پس از این هم بیت‌های کم ارزش این خروجی در rf_in_1 و بیت‌های پرارزش آن در rf_in_2 برای پر کردن ثبات‌ها ریخته می‌شود. دقت کنید در این دستورات محاسباتی تفاوتی نمی‌کند که 11 بیت کم‌ارزش instruct چه مقداری است.

اگر دستور محاسباتی نباشد، حداکثر در یک ثبات نیاز به نوشتن داریم پس rf_w_enable_2 صفر می‌شود. در هر دو دستور هم به حافظه باید دسترسی داشته باشیم پس mem_address برابر با 9 بیت کم‌ارزش instruct می‌شود. بسته به دومین بیت پرارزش instruct هم می‌توانیم بین load و store تصمیم بگیریم پس rf_w_enable_2 و mem_w_enable را به ترتیب با توجه به load یا store بودن برابر با نقیض دومین بیت پرارزش و خود دومین بیت پرارزش می‌گیریم. در انتها هم اگر دستور load بود w_reg_1 را برابر بیت‌های رجیستر instruct گرفته و پس از کمی تاخیر برای پایدار شدن خروجی حافظه، rf_in_1 برای رجیستر فایل برابر مقدار data_out حافظه می‌شود؛ و اگر دستور store باشد، r_reg را برابر بیت‌های رجیستر instruct گرفته و پس از کمی تاخیر برای پایدار شدن خروجی رجیستر فایل، data_in برای حافظه برابر مقدار rf_out می‌شود.

Testbench:

حالا که کلیت یک پردازنده آرایه‌ای را طراحی کردیم باید برای اطمینان از صحت طراحی‌مان، آن را مورد آزمون قرار دهیم.

شکل کلی یک ماژول به صورت زیر است، در ادامه آن را بررسی می‌کنیم.

```
module TESTBENCH;


    reg clk, reset;
    reg [12:0] instruct;
    wire [511:0] A1, A2, A3, A4;


    VECTOR_PROCESSOR processor (clk, reset, instruct, A1, A2, A3, A4);


    reg [1:0] A1_addr, A2_addr, A3_addr, A4_addr;
    reg [1:0] load, store, add, mult;


    initial begin
        A1_addr <= 2'b00;
        A2_addr <= 2'b01;
        A3_addr <= 2'b10;
        A4_addr <= 2'b11;


        load <= 2'b00;
        store <= 2'b01;
        mult <= 2'b10;
        add <= 2'b11;
    end
```

```
    always #25 clk = ~clk;


    initial begin
        $monitor($time, ":\nA1: %h\nA2: %h\nA3: %h\nA4: %h", A1, A2, A3,
A4);
        clk = 0;
        reset <= 1;
        #50
        reset <= 0;
        instruct <= {load, A1_addr, 9'b0}; // A1 <- M[0]
        #50
        instruct <= {load, A2_addr, 9'b10000}; // A2 <- M[16]
        #50
        instruct <= {load, A3_addr, 9'b110000}; // A3 <- M[48]
        #50
        instruct <= {load, A4_addr, 9'b10000}; // A4 <- M[16]
        #50
        instruct <= {mult, 2'b00, 9'b0};//{A4 A3} <- A1 * A2 (vector mult)
        #50
        instruct <= {add, 2'b00, 9'b0}; // {A4 A3} <- A1 + A2 (vector add)
        #50
        instruct <= {store, A3_addr, 9'b100000}; // M[32] <- A3
        #50
        instruct <= {store, A4_addr, 9'b110000}; // M[48] <- A4
        #50
        instruct <= {load, A1_addr, 9'b100000}; // A1 <- M[32]
        #50
        instruct <= {load, A2_addr, 9'b110000}; // A2 <- M[48]
```

```
        #50
        instruct <= {mult, 2'b11, 9'b101010101}; // {A4 A3} <- A1 * A2
(vector mult)
        #50
        instruct <= {add, 2'b01, 9'b111100000}; // {A4 A3} <- A1 + A2
(vector add)
        #100
        $stop;
    end
endmodule
```

در این ماژول دوره تناوب ساعت 50 واحد زمانی است و عملکرد دستورات جلوی آن کامنت شده.

برای راحتی خوانش دستورات، آدرس ثبات‌ها و همچنین خود دستورات را در رجیسترهای ثابت با نام‌های متناسب ذخیره کردیم.

خروجی این ماژول را می‌توانید در صفحه‌ی بعد مشاهده کنید.

دقت کنید که هنگام دستورات store ثبات‌ها تغییری نمی‌کنند و برای همین در خروجی چیزی نمایش داده نمی‌شود اما بعدا با load از همان خانه‌های حافظه که در آن‌ها store کردیم، صحت این دستور را می‌سنجیم.

```
                0:
A1: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
A2: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
A3: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
A4: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
                 50:
A1: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
A2: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
A3: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
A4: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                100:
A1: 1fffffffff000000ff0000ffffffffff0000000b0000000a0000000900000008000000070000000600000005000000040000000300000002000000010000000
A2: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
A3: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
A4: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                150:
A1: 1fffffffff000000ff0000ffffffffff0000000b0000000a0000000900000008000000070000000600000005000000040000000300000002000000010000000
A2: efffffff1000000000ffff00ffffffff000007db000007da000007d9000007d8000007d7000007d6000007d5000007d4000007d3000007d2000007d1000007d0
A3: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
A4: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                200:
A1: 1fffffffff000000ff0000ffffffffff0000000b0000000a0000000900000008000000070000000600000005000000040000000300000002000000010000000
A2: efffffff1000000000ffff00ffffffff000007db000007da000007d9000007d8000007d7000007d6000007d5000007d4000007d3000007d2000007d1000007d0
A3: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
A4: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                250:
A1: 1fffffffff000000ff0000ffffffffff0000000b0000000a0000000900000008000000070000000600000005000000040000000300000002000000010000000
A2: efffffff1000000000ffff00ffffffff000007db000007da000007d9000007d8000007d7000007d6000007d5000007d4000007d3000007d2000007d1000007d0
A3: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
A4: efffffff1000000000ffff00ffffffff000007db000007da000007d9000007d8000007d7000007d6000007d5000007d4000007d3000007d2000007d1000007d0

                300:
A1: 1fffffffff000000ff0000ffffffffff0000000b0000000a0000000900000008000000070000000600000005000000040000000300000002000000010000000
A2: efffffff1000000000ffff00ffffffff000007db000007da000007d9000007d8000007d7000007d6000007d5000007d4000007d3000007d2000007d1000007d0
A3: f000000100000000feff01000000000100005669000004e84000046a100003ec0000036e100002f040000272900001f500000177900000fa4000007d100000000
A4: 1dffffe0ff0000000feff01fffffffe000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                350:
A1: 1fffffffff000000ff0000ffffffffff0000000b0000000a0000000900000008000000070000000600000005000000040000000300000002000000010000000
A2: efffffff1000000000ffff00ffffffff000007db000007da000007d9000007d8000007d7000007d6000007d5000007d4000007d3000007d2000007d1000007d0
A3: 0fffffe0f000000fffffffffffffffe000007e6000007e4000007e2000007e0000007de000007dc000007da000007d8000007d6000007d4000007d2000007d0
A4: 0000000100000001000000000000000100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                500:
A1: 0fffffe0f000000fffffffffffffffe000007e6000007e4000007e2000007e0000007de000007dc000007da000007d8000007d6000007d4000007d2000007d0
A2: efffffff1000000000ffff00ffffffff000007db000007da000007d9000007d8000007d7000007d6000007d5000007d4000007d3000007d2000007d1000007d0
A3: 0fffffe0f000000fffffffffffffffe000007e6000007e4000007e2000007e0000007de000007dc000007da000007d8000007d6000007d4000007d2000007d0
A4: 0000000100000001000000000000000100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                550:
A1: 0fffffe0f000000fffffffffffffffe000007e6000007e4000007e2000007e0000007de000007dc000007da000007d8000007d6000007d4000007d2000007d0
A2: 0000000100000001000000000000000100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
A3: 0fffffe0f000000fffffffffffffffe000007e6000007e4000007e2000007e0000007de000007dc000007da000007d8000007d6000007d4000007d2000007d0
A4: 0000000100000001000000000000000100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                600:
A1: 0fffffe0f000000fffffffffffffffe000007e6000007e4000007e2000007e0000007de000007dc000007da000007d8000007d6000007d4000007d2000007d0
A2: 0000000100000001000000000000000100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
A3: 0fffffe0f0000000000000ffffffe000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
A4: 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
                650:
A1: 0fffffe0f000000fffffffffffffffe000007e6000007e4000007e2000007e0000007de000007dc000007da000007d8000007d6000007d4000007d2000007d0
A2: 0000000100000001000000000000000100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
A3: 0fffffff0f000001fffffffffffffffe000007e6000007e4000007e2000007e0000007de000007dc000007da000007d8000007d6000007d4000007d2000007d0
A4: 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```