دانشگاه صنعتی شریف

دانشکده‌ی مهندسی کامپیوتر

طراحی سیستم‌های دیجیتال

پاسخ سوال امتیازی

انتخاب: سوال ۷ میان‌ترم

علیرضا میرشفیعیان

۴۰۱۱۰۶۶۲۸

تابستان ۱۴۰۳

سوال ۷-

تمامی کدها در کنار این گزارش ضمیمه شده‌اند و برای خوانایی بیشتر، کامت‌گذاری شده‌اند.

ابتدا رجیستر فایل را طراحی میکنیم:

```
module RegisterFile (
    input clk,
    input write_enable,
    input write_enable_alu,
    input [1:0] write_select,
    input [511:0] data_in,
    input [511:0] alu_data_A3,
    input [511:0] alu_data_A4,
    output reg [511:0] A1,
    output reg [511:0] A2,
    output reg [511:0] A3,
    output reg [511:0] A4
);

    always @(posedge clk) begin
        if (write_enable) begin
            case (write_select)
                2'b00: A1 <= data_in;
                2'b01: A2 <= data_in;
                2'b10: A3 <= data_in;
                2'b11: A4 <= data_in;
            endcase
        end

        if (write_enable_alu) begin
            A3 <= alu_data_A3;
            A4 <= alu_data_A4;
        end
    end
endmodule
```

برای سادگی طراحی، هر چهار رجیستر همواره قابل خواندن از بیرون رجیستر فایل هستند و نیازی به انتخابشان نیست. برای نوشتن اما، دو جا نیاز است:

1. در دستور store نیاز است، که با write_enable و write_select هندل میشود.
2. برای ذخیره‌سازی نتیجه‌ی عملیات ریاضی در A3 و A4 نیاز است، که با write_enable_alu هندل میشود.

میبینید که طراحی ما سه ورودی 512 بیتی دارد. برای بهبود، میتوانیم آن را به دو ورودی 512 بیتی کاهش دهیم بدون اینکه طراحی را تغییر دهیم. همچنین میتوانیم 4 خروجی 512 بیتی را به دو خروجی کاهش دهیم؛ اما فعلا برای سادگی از آن صرف نظر کردیم.

در ادامه، واحد ریاضی را طراحی میکنیم:

```
module ALU (
    input [511:0] A1,
    input [511:0] A2,
    output reg [511:0] A3,
    output reg [511:0] A4,
    input operation // 0: add, 1: multiply
);
    integer i;
    reg signed [32:0] add_temp [15:0];
    reg signed [63:0] mul_temp [15:0];

    always @(*) begin
        if (operation == 1'b0) begin // Addition
            for (i = 0; i < 16; i = i + 1) begin
                add_temp[i] = $signed(A1[32*i +: 32]) + $signed(A2[32*i +:
32]);
                A3[32*i +: 32] = add_temp[i][31:0];
                A4[32*i +: 32] = {32{add_temp[i][32]}}; // Correctly sign-
extend
            end
        end else begin // Multiplication
            for (i = 0; i < 16; i = i + 1) begin
                mul_temp[i] = $signed(A1[32*i +: 32]) * $signed(A2[32*i +:
32]);
                A3[32*i +: 32] = mul_temp[i][31:0];
                A4[32*i +: 32] = mul_temp[i][63:32];
            end
        end
    end
endmodule
```

در این طراحی از یک حلقه‌ی for با حد ثابت (یعنی تعداد اجرای حلقه همواره ثابت است)، از $signed، و از سینتکس [from +: width] بهره بردیم که همگی قابل سنتزند (زیرا حلقه میتواند unroll شود و بازه‌ی [+:] همواره مشخص و ثابت است). مطابق با خواسته‌ی سوال، هر رجیستر حاوی16 کلمه‌ی 32بیتی است و هر عملیات ریاضی به صورت جداگانه و موازی روی این 16 کلمه اجرا می‌شود و 16 خروجی به دست می‌آید که در ادامه باید در A3 و A4 ذخیره شود. برای مثال اگر کلمه‌ی اول A1 برابر با 2 و کلمه‌ی اول A2 برابر با 4 بود، پس از عملیات جمع، کلمه‌ی اول A3 برابر با 6 و کلمه‌ی اول A4 برابر با 0 (زیرا overflow ندارد) خواهد بود.

به دو نکته در این طراحی باید توجه کرد:

1. عملیات ریاضی به کمک $signed به صورت علامت‌دار اجرا می‌شوند.
2. حاصل ضرب دو عدد 32 بیتی در 64 بیت جا می‌شود. حاصل جمع دو عدد 32 بیتی نیز در 33 بیت جا می‌شود، و ما برای پر کردن A4 باید حاصل جمع را sign_extend (و نه zero_extend) کنیم.

دقت کنید که alu را به شکل combinational طراحی کردیم.

در ادامه حافظه را طراحی میکنیم:

```
module Memory (
    input clk,
    input [8:0] mem_addr,
    input [511:0] data_in,
    input write_enable,
    output reg [511:0] data_out
); //#(parameter addr_width = 9, width = 32, count = 16), localparam depth =
2**addr_width;

    // Memory array: 512 x 32 bits
    reg [31:0] memory [0:511];
    integer i;

    always @(posedge clk) begin
        if (write_enable) begin
            // Write 16 consecutive 32-bit words
            for (i = 0; i < 16; i = i + 1) begin
                memory[mem_addr + i] <= data_in[32*i +: 32];
            end
        end
    end

    // Asynchronous read
    always @(*) begin
        // Read 16 consecutive 32-bit words
        for (i = 0; i < 16; i = i + 1) begin
            data_out[32*i +: 32] = memory[mem_addr + i];
        end
    end
endmodule
```

حافظه از 512 کلمه‌ی 32 بیتی تشکیل شده است. نوشتن در حافظه به صورت synchronous و روی 16 خانه‌ی متوالی انجام می‌شود و همان توضیحاتی که قبلا دادیم برای این هم صادقند و کد قابل سنتز است. برای خواندن اما تصمیم گرفتیم که

به صورت asynchronous عمل کند. دلیل این انتخاب این است که پیش از این، خواندن را به صورت سنکرون انجام میدادیم، اما در تست متوجه شدیم که دستور load به درستی عمل نمیکند زیرا رجیستر فایل پیش از آنکه خروجی حافظه (یعنی data_out) آماده‌ی خواندن باشد، از آن میخواند. برای حل این مشکل میتوانستیم دستور load را به جای 1 سایکل در 2 سایکل اجرا کنیم، اما برای کم نشدن فرکانس و همچنین حفظ سادگی طراحی تصمیم گرفتیم خواندن از حافظه را آسنکرون کنیم.

یک نکته که باید به آن توجه کنیم این است که از آدرس ورودی حافظه تا انتهای حافظه باید حداقل 16 کلمه باشد، وگرنه مدار عملکردی تعریف نشده خواهد داشت. این مشکل را در ادامه در ماژول VectorProcessor حل خواهیم کرد و فعلا به آن نمیپردازیم، هرچند که با نوشتن یک شرط ساده در این ماژول هم میتوانستیم به آن توجه کنیم.

یک بهبود که میتوانیم به این طراحی بدهیم، پارامتری کردن آن است. فعلا برای سادگی از آن صرف نظر کردیم، اما نحوه‌ی پیاده‌سازی آن را کامنت کردیم که میتوانید در بالا ببینید.

در ادامه، ماژول اصلی یعنی VectorProcessor که ماژول‌های قبلی را به وصل میکند را توصیف میکنیم:

```
module VectorProcessor (
    input clk,
    input [1:0] instruction, // 00: load, 01: store, 10: add, 11: multiply
    input [8:0] mem_addr,
    input [1:0] reg_select,
    output out_of_bound
);

    wire [511:0] mem_data_out;
    reg [511:0] mem_data_in;
    reg mem_write_enable;
    reg reg_write_enable;
    reg reg_write_enable_alu;

    wire [511:0] alu_A3;
    wire [511:0] alu_A4;

    wire [511:0] A1;
    wire [511:0] A2;
    wire [511:0] A3;
    wire [511:0] A4;

    // Instantiate Memory
    Memory mem (
        .clk(clk),
        .mem_addr(mem_addr),
        .data_in(mem_data_in),
        .write_enable(mem_write_enable),
        .data_out(mem_data_out)
    );
```

```
// Instantiate Register File
RegisterFile rf (
    .clk(clk),
    .write_enable(reg_write_enable),
    .write_enable_alu(reg_write_enable_alu),
    .write_select(reg_select),
    .data_in(mem_data_out),
    .alu_data_A3(alu_A3),
    .alu_data_A4(alu_A4),
    .A1(A1),
    .A2(A2),
    .A3(A3),
    .A4(A4)
);

// Instantiate ALU
ALU alu (
    .A1(A1),
    .A2(A2),
    .operation(instruction[0]), // 0: add, 1: multiply
    .A3(alu_A3),
    .A4(alu_A4)
);

// Address bounds checking
assign out_of_bound = (mem_addr + 16 > 512);

always @(*) begin
    mem_write_enable = 0;
    reg_write_enable = 0;
    reg_write_enable_alu = 0;
    mem_data_in = 0;

    if (!out_of_bound) begin
        case (instruction)
            2'b00: begin // Load
                reg_write_enable = 1;
            end
            2'b01: begin // Store
                mem_write_enable = 1;
                case (reg_select)
                    2'b00: mem_data_in = A1;
                    2'b01: mem_data_in = A2;
                    2'b10: mem_data_in = A3;
                    2'b11: mem_data_in = A4;
                endcase
            end
```

```
                2'b10: begin // Add
                    reg_write_enable_alu = 1;
                end
                2'b11: begin // Multiply
                    reg_write_enable_alu = 1;
                end
            endcase
        end
    end
endmodule
```

این پردازنده ورودی‌های زیر را دارد:

- کلاک
- دو بیت برای انتخاب دستور (load, store, add, multiply)
- آدرس حافظه (برای load/store)
- دو بیت برای انتخاب رجیستر (برای load/store)

برای تعامل با این پردازنده فعلا باید به حافظه‌ی آن نگاه کنیم. میتوانیم مقدار رجیستر انتخاب شده را نیز بعنوان یکی از خروجی‌های پردازنده قرار دهیم، اما چون نیاز نداشتیم فعلا آن کار را نکردیم. یک خروجی پرچم وضعیت اما قرار دادیم:

- Out_of_bound

به دلیل اینکه تمام load/storeها روی 16 کلمه کار میکنند، اگر از آدرسی که به پردازنده ورودی داده میشود تا انتهای حافظه کمتر از 16 کلمه وجود داشته باشد، پردازنده نمیتواند به روال عادی پیش رود. برای هندل کردن این حالت، ما میتوانستیم کارهای مختلفی بکنیم. مثلا بگوییم که اگر فقط 7 کلمه تا انتهای حافظه موجود بود، عملیات را فقط روی 7 کلمه بجای 16 کلمه انجام بده. اما تصمیم گرفتیم که یک پرچم وضعیت قرار دهیم تا درصورت نامعتبر بودن ادرس ورودی، فعال شود. بدیهی است که از کارکرد پردازنده کم نمیشود و همچنان به تمام خانه‌های حافظه میتوان دسترسی داشت.

نیازی به قرار دادن پرچم وضعیت دیگری ندیدیم، زیرا دستورات این پردازنده محدودند. اما در آینده برای گسترش پروژه میتوان پرچم zero را به سادگی اضافه کرد. همچنین چون در پردازنده‌ی ما نتیجه‌ها به هر حال در دو رجیستر A3 و A4 (و نه در یک رجیستر) ذخیره میشوند و قطعا در آن دو جا میشوند، پرچم overflow قرار ندادیم.

تمام اتصالات در instantiationها را نامگذاری کردیم تا کد خواناتر باشد.

کارکرد پردازنده در بلوک اصلیش به این صورت است که اگر آدرس معتبر بود (و out_of_bound غیرفعال بود)، چک میکند که ببیند چه دستوری باید اجرا شود و طبق آن، سیگنال‌های کنترلی‌ای مثل mem_write_enable، reg_write_enable، و ... را تنظیم میکند. مشاهده میکنید که بلوک اصلی توصیف این طراحی، کوتاه شد؛ دلیلش این است که هنگام طراحی‌های ماژول‌های قبلی دقت کردیم و ورودی و خروجی‌هایی برای انها قرار دادیم تا در اینجا کارمان ساده‌تر شود. برای مثال خروجی

حافظه مستقیما به ورودی رجیستر فایل وصل است و کافیست پردازنده reg_write_enable را فعال کند تا دستور load به درستی اجرا شود (به یاد بیاورید که برای همینجا، خواندن از حافظه را آسنکرون کردیم).

طراحی ما به پایان رسید. در ادامه، تست‌هایی که برای سنجش صحت طراحی خود نوشتیم (که به ما در رفع خطاهایی که طراحی‌های متعدد قبلیمان داشتند کمک کردند) را می‌آوریم.

نوشتن Test Bench:

فایل کامل تست‌بنچ در کنار گزارش ضمیمه شده است. به علت حجم زیاد آن، در اینجا فقط تکه‌هایی از آن را نشان میدهیم.

ابتدا از پردازنده یک instance میگیریم:

```
// Instantiate the VectorProcessor
    VectorProcessor vp (
        .clk(clk),
        .instruction(instruction),
        .mem_addr(mem_addr),
        .reg_select(reg_select),
        .out_of_bound(out_of_bound)
    );
```

سپس کلاک را تنظیم میکنیم:

```
    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
```

سپس حافظه‌ی پردازنده را مقداردهی اولیه میکنیم تا به کمک آن بتوانیم طراحی خود را به ازای ورودی‌های مختلف تست کنیم. این یک روش استاندارد در طراحی سخت‌افزار است و از unique identifier استفاده می‌کند.

برای آزمایش منطق پردازنده‌ی خود، آن را با سه مدل داده تست میکنیم:

1. داده‌هایی ساده و نرمال
2. داده‌های مرزی یا edgecaseها
3. داده‌های تصادفی (Randomly Generated)

کد این را مشاهده میکنید:

```
// Initialize memory with specific values
    initial begin
        // Normal values initialization
        for (i = 0; i < 16; i = i + 1) begin
            vp.mem.memory[i] = i + 1; // Memory 0-15 with values 1 to 16
        end
        for (i = 16; i < 32; i = i + 1) begin
            vp.mem.memory[i] = (i - 15) * 2; // Memory 16-31 with values 2, 4,
6, ..., 32
        end

        // Edge cases
        vp.mem.memory[32] = 32'h7FFFFFFF; // Edge case: Maximum positive value
        vp.mem.memory[33] = 32'h80000000; // Edge case: Maximum negative value
        vp.mem.memory[34] = 32'h00000001; // Small positive value
        vp.mem.memory[35] = 32'hFFFFFFFF; // Edge case: -1 in two's complement
        vp.mem.memory[36] = 32'h80000001;
        vp.mem.memory[37] = 32'h80000000;
        vp.mem.memory[38] = 32'h7FFFFFFF;
        vp.mem.memory[39] = 32'h7FFFFFFF;

        // Random values
        for (i = 0; i < 32; i = i + 1) begin
            vp.mem.memory[i + 48] = $random;
        end
    end
```

برای حالات مرزی، ترکیب‌های مختلفی از ضرب و جمع روی بزرگترین و کوچکترین اعداد مثبت و منفی (32 بیتی) را آزمایش خواهیم کرد. همچنین آزمایش با داده‌های رندوم به ما این اطمینان را میدهد که پردازنده برای داده‌های مختلف، مثبت و منفی، در حالات مختلف و پیشبینی نشده، درست عمل میکند.

برای خوانایی بیشتر خروجی تست‌های خود، تسک زیر را مینویسیم که یک رجیستر (که محتوی 16 کلمه است) را به صورت کلمه کلمه نشان میدهد:

```
    // Helper function to display 512-bit registers
    task display_512bit;
        input [511:0] reg_val;
        integer j;
        begin
            for (j = 15; j >= 0; j = j - 1) begin
                $write("%h", reg_val[32*j +: 32]);
                if (j != 0) $write("-");
            end
```

```
            $write("\n");
        end
    endtask
```

به کمک این تسک میتوانیم خروجی خود را در آینده به سادگی به هر فرمی که بخواهیم درآوریم. فعلا با فرم خوانایی که نوشتیم پیش میرویم.

حال آزمایش خود را ابتدا روی داده‌های معمولی آغاز میکنیم:

```
// Test normal load/store operations
        instruction = 2'b00; // Load
        mem_addr = 9'd0; // Load from memory address 0
        reg_select = 2'b00; // Load into A1
        #10;

        // Debug: Check values in A1
        $display("A1 after loading from memory address 0:");
        display_512bit(vp.rf.A1);

        instruction = 2'b00; // Load
        mem_addr = 9'd16; // Load from memory address 16
        reg_select = 2'b01; // Load into A2
        #10;

        // Debug: Check values in A2
        $display("A2 after loading from memory address 16:");
        display_512bit(vp.rf.A2);

        // Perform addition
        instruction = 2'b10; // Add
        #10;

        // Check for addition result
        if (out_of_bound) $display("Test Failed: Out of Bound Detected During
Add");
        else begin
            // Display result for verification
            $display("Addition Result A3:");
            display_512bit(vp.rf.A3);
            $display("Addition Overflow A4:");
            display_512bit(vp.rf.A4);
        end

        // Perform multiplication
        instruction = 2'b11; // Multiply
```

```
        #10;

        // Check for multiplication result
        if (out_of_bound) $display("Test Failed: Out of Bound Detected During
Multiply");
        else begin
            // Display result for verification
            $display("Multiplication Result A3:");
            display_512bit(vp.rf.A3);
            $display("Multiplication Upper A4:");
            display_512bit(vp.rf.A4);
        end
```

نحوه‌ی کارش به این صورت است که ابتدا مقادیری که در حافظه قرار داده بودیم را در رجیسترها load میکند، سپس محتوای رجیسترها را نمایش میدهد تا از عملکرد صحیح دستور load مطمئن شویم. سپس چک میکند که پرچم out_of_bound به اشتباه فعال نباشد. سپس عملیات جمع و ضرب را انجام میدهد و نتیجه‌ی آنها (که در A3 و A4اند) را نشان میدهد تا از صحت عملکرد این دو دستور نیز مطمئن شویم.

در ادامه دقیقا با همین فرمت داده‌های مرزی را مورد آزمون قرار میدهیم. میتوانید کد آن را در ضمیمه مشاهده کنید.

سپس چک میکنیم که آیا پرچم out_of_bound هنگام نیاز فعال میشود یا خیر:

```
// Test out-of-bound load
        instruction = 2'b00; // Load
        mem_addr = 9'd500; // Out of bound
        reg_select = 2'b00;
        #10;
        if (out_of_bound) $display("Test Passed: Out of Bound Load Detected");
        else $display("Test Failed: Out of Bound Load Not Detected");

        // Test out-of-bound store
        instruction = 2'b01; // Store
        mem_addr = 9'd500; // Out of bound
        reg_select = 2'b00;
        #10;
        if (out_of_bound) $display("Test Passed: Out of Bound Store
Detected");
        else $display("Test Failed: Out of Bound Store Not Detected");
```

سپس با همان فرمتی که گفتیم، ضرب و جمع را روی داده‌های تصادفی هم چک میکنیم. میتوانید به کد در ضمیمه رجوع کنید.

صحت دستور load را در طی این تست‌ها روی مکان‌های مختلف حافظه تایید کردیم. در آخر صحت دستور store را هم تایید میکنیم:

```
// Store the result of a test back into memory
        instruction = 2'b01; // Store
        mem_addr = 9'd100; // Store result at memory address 100
        reg_select = 2'b10; // Store A3
        #10;

        // Debug: Check stored values in memory
        $display("Memory after storing A3:");
        for (i = 15; i >= 0; i = i - 1) begin
            $write("%h", vp.mem.memory[100 + i]);
            if (i != 0) $write("-");
        end
        $write("\n");
```

بسیار عالی. کار تمام است و پردازنده را شبیه‌سازی می‌کنیم. نتیجه‌ی آن:

داده‌های عادی:

```
VSIM 29> run -all
# A1 after loading from memory address 0:
# 00000010-0000000f-0000000e-0000000d-0000000c-0000000b-0000000a-00000009-00000008-00000007-00000006-00000005-00000004-00000003-00000002-00000001
# A2 after loading from memory address 16:
# 00000020-0000001e-0000001c-0000001a-00000018-00000016-00000014-00000012-00000010-0000000e-0000000c-0000000a-00000008-00000006-00000004-00000002
# Addition Result A3:
# 00000030-0000002d-0000002a-00000027-00000024-00000021-0000001e-0000001b-00000018-00000015-00000012-0000000f-0000000c-00000009-00000006-00000003
# Addition Overflow A4:
# 00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000
# Multiplication Result A3:
# 00000200-000001c2-00000188-00000152-00000120-000000f2-000000c8-000000a2-00000080-00000062-00000048-00000032-00000020-00000012-00000008-00000002
# Multiplication Upper A4:
# 00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000-00000000
# ----------------------------------------------------------------
# ----------------------------------------------------------------
```

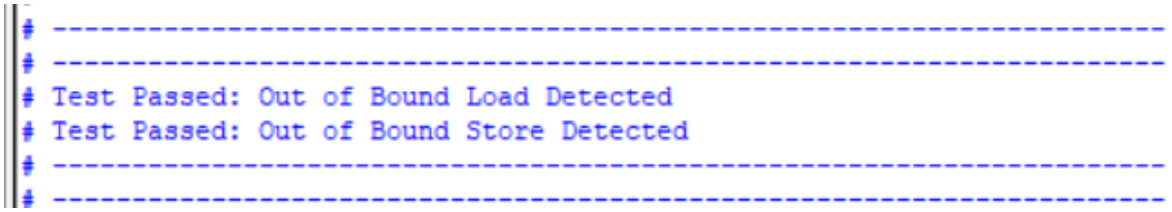
برای خوانایی بیشتر، ستون‌ها با علامت – از هم جدا شده‌اند. هر ستون مربوط به یک جفت داده‌ی ورودی است. برای مثال، در ستون سمت راست، دو عدد 1 و 2 را با هم جمع زدیم که حاصلش شده 3 و overflow نداشته، و سپس دو عدد را در هم ضرب کردیم که حاصلش شده 2 و باز هم در 32 بیت A3 جا شده و بیت‌های مربوط به A4اش همگی 0اند. صحت تمام ستون‌ها را میتوانید با ماشین حساب تایید کنید.

داده‌های مرزی:

```
# ----------------------------------------------------------------
# ----------------------------------------------------------------
# A1 after loading from memory address 32:
# xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-7fffffff-7fffffff-80000000-80000001-ffffffff-00000001-80000000-7fffffff
# A2 after loading from memory address 33:
# 12153524-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-7fffffff-7fffffff-80000000-80000001-ffffffff-00000001-80000000
# Addition Result A3:
# xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-fffffffe-ffffffff-00000001-80000000-00000000-80000001-ffffffff
# Addition Overflow A4:
# xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-00000000-ffffffff-ffffffff-ffffffff-00000000-ffffffff-ffffffff
# Multiplication Result A3:
# xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-00000001-80000000-80000000-7fffffff-ffffffff-80000000-80000000
# Multiplication Upper A4:
# xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-xxxxxxxx-3fffffff-c0000000-3fffffff-00000000-ffffffff-ffffffff-c0000000
# ----------------------------------------------------------------
# ----------------------------------------------------------------
```

ترکیبات مختلفی از داده‌های مرزی مثبت و منفی کوچک و بزرگ را تحت عملیات جمع و ضرب آزمایش کردیم که نتیجه‌ی آنها را میتوانید ببینید و با ماشین حساب تایید کنید. برای مثال ستون سوم از سمت راست حاصل ضرب و جمع 1 با 1- را نشان میدهد. مشاهده میکنید که A4 در مواقع نیاز به درستی با 0 یا با 1 پر میشود (و علامت A3 را نمایش میدهد).

آزمون حفظ مرز حافظه:



داده‌های تصادفی (و همچنین آزمون دستور store، که نتیجه‌اش در دو خط آخر آمده):

میبینید که داده‌های تصادفی مختلفی آزمایش شدند. برای مثال ستون چهارم از سمت راست، عملیات زیر را بیان میکند:

b1f05663 + bbd27277 = ffffffff 6dc2c8da

(A1) (A2) (A4) (A3)

(-1309649309) + (-1143836041) = -2453485350

b1f05663 * bbd27277 = 14ca0cf6 eb663e05

(-1309649309) * (-1143836041) = 1498024080704945669

که صحیح است.

با توجه به سه حالت مختلف دسته‌های داده که تست کردیم، و اینکه کد تست‌بنچمان 250 خط شد، میتوان گفت به خوبی و بیش از حد نیاز به آزمایش پردازنده‌ی خود پرداختیم.

در آخر، بیان می‌کنیم که در آینده می‌توان با گام‌های زیر، این پروژه را بهبود داد:

1. نکاتی که در طی گزارش به آنها اشاره کردیم را اعمال کنیم.
2. توصیف را به صورت پارامتری درآوریم.
3. تست‌بنچی که نوشتیم را طوری تغییر دهیم که پاسخ‌های پردازنده را به‌جای نمایش دادن، به صورت خودکار چک کند و در آخر در صورتی که تمام آزمون‌ها پاس شدند، صرفا در یک خط این را اعلام کند. فعلا خودمان آنها را به صورت دستی چک کردیم.