گزارش آزمایش پنجم

دانشگاه صنعتی شریف - **بهار ۱۴۰۰**

آزمایشگاه طراحی سیستم‌های دیجیتال - دکتر اجلالی

نویسندگان:
سروش جهان‌زاد - ۹۸۱۰۰۳۸۹
علی حاتمی تاجیک - ۹۸۱۰۱۳۸۵

۱ مقدمه

در مستند پیش رو گزارشی بر روند طراحی مدار ضرب کننده اعداد با الگوریتم بوث[۱] ارائه شده است. تصاویر مربوط به ای‌اس‌ام چارت مدار همراه با تصاویر مربوط به تست‌ها در کنار فایل این گزارش قرار دارند. همینطور فایل‌های کد این مدار(گرچه کدها درون این مستند نیز موجودند) به همراه کد تست بنچ آن نیز در کنار گزارش قابل دسترسی اند. سعی شده تا در صورت لزوم توضیحات کافی برای بخش‌های مختلف آورده شود. از صبر و بردباری شما در مطالعه این گزارش سپاسگزاریم.

با احترام

سروش جهان‌زاد، علی حاتمی تاجیک – **بهار ۱۴۰۰**

۱ Booth

۲ الگوریتم Booth و ASM Chart آن

الگوریتم بوث، الگوریتمی است که با استفاده از آن ضرب اعداد علامت‌دار مکمل دو انجام می‌شود و برای راحت شدن و تسریع ضرب انجام می‌شود.

چون الگوریتم مورد نظر سؤال با الگوریتم اصلی مقداری تفاوت دارد (به خاطر شیفت‌های چندتایی) کمی روند ماژول تفاوت خواهد داشت. در ادامه ابتدا رجیسترها و چند علامت توضیح داده خواهندشد و پس از آن ای‌اس‌ام چارت رسم می‌شود و الگوریتم بهبود یافته بوث در آن نشان داده خواهد شد. توضیحاتی که در ادامه خواهد آمد مربوط به یک ضرب‌کننده ۸ بیتی با این الگوریتم خواهد بود. برای اینکه مدار حالت دو بخشی کنترل یونیت و دیتا پث پیدا کند، از رجیستر عادی استفاده نشده است و چند ماژول هم برای رجیسترهای عادی ساخته شده است تا دارای سیگنال لود باشند.

ورودی‌های ما به صورت زیر خواهد بود:

- IN1: عدد ۸ بیتی اول مکمل دو
- IN2: عدد ۸ بیتی دوم مکمل دو
- S: سیگنال شروع به کار مدار
- F: سیگنال نشان‌دهنده پایان کار مدار (این سیگنال برابر با صفر بودن رجیستر left خواهد بود و در asm chart نخواهد آمد)

رجیسترهایی که استفاده می‌شوند به صورت زیر خواهند بود:

- A: یک رجیستر ۱۷ بیتی با قابلیت شیفت به راست چندتایی. ۱۶ بیت پرارزش آن پس از پایان کار مدار نشان‌دهنده پاسخ نهاییست. همینطور عملیات‌های جمع و تفریق روی ۸ بیت پرارزش آن انجام می‌شود. همینطور در ابتدای کار عدد ابتدایی در بیت‌های دوم تا نهم آن ریخته خواهد شد. این رجیستر یک ورودی شیفت، یک ورودی لود، یک ورودی عدد ورودی و یک ورودی اندازه شیفت خواهد داشت.
- M: رجیستر ۸ بیتی که اندازه عدد دوم را نگه می‌دارد.
- Left: رجیستر ۴ بیتی که تعداد شیفت‌های باقی‌مانده را در خود نگه می‌دارد که در ابتدا مقدار ۸ درون آن است.

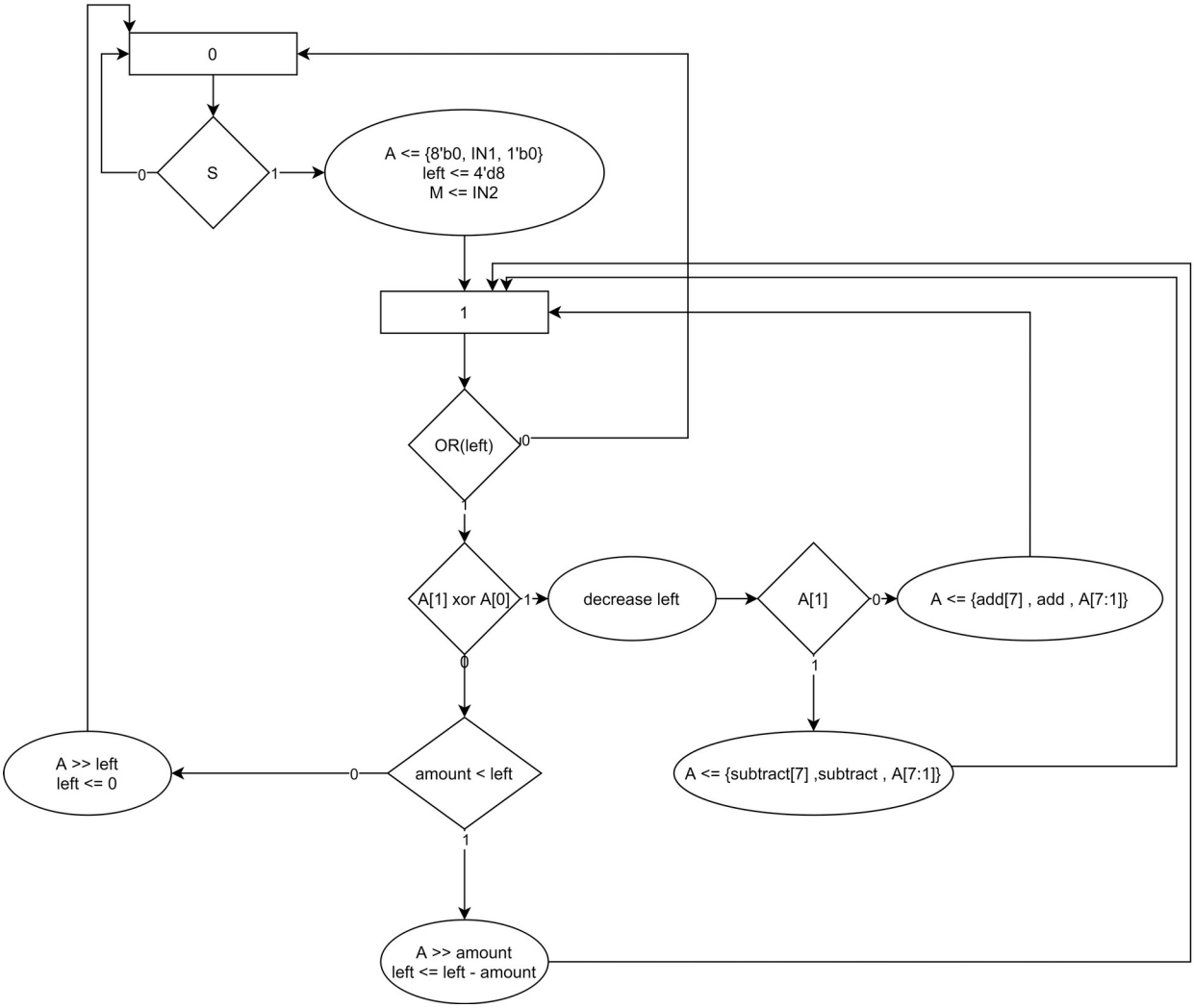
سیگنال‌هایی که به صورت ترکیبی درون دیتا پث وجود دارند:

- amount(): این یک سیگنال است که می‌سنجد که چه تعداد شیفت باید انجام شود تا دو بیت ابتدایی اعداد متفاوتی باشند. این سیگنال به روشی مانند پرایاریتی انکودر کار می‌کند با این تفاوت که صفرها را دنبال نمی‌کند و به دنبال اولین ناصفری می‌گردد، مثل اینکه دو به دو بیت‌های کنارهم را xor کنیم و سپس به دنبال جایگاه اولین یک بگردیم. این سیگنال در دیتا پث پیاده‌سازی می‌شود اما برای جلوگیری از شلوغی ای اس‌ام چارت از نوشتن رابطه دقیق آن در آن خودداری شده و فقط از عبارت amount() استفاده شده است.
- add(): این سیگنال جمع A[16:8] با M است.
- subtract(): این سیگنال تفریق M از A[16:8] است.

ASM Chart

مدار از دو استیت تشکیل شده است. یکی اینکه منتظر است تا شروع به کار کند و مقادیر را لود کند و دیگری کار اصلی را انجام می‌دهد. زمانی که کار شروع می‌شود (پس از مقدار دهی‌ها در مرحله قبل) اول چک می‌کند که تعداد شیفت‌های باقی‌مانده صفر شده است یا نه. اگر شده باشد کار به پایان رسیده است و به استیت ابتدایی می‌رود.

اگر تمام نشده باشد ابتدا چک می‌کند که دو بیت اتدایی A مانند هم هستند یا نه. اگر این حالت باشد یعنی باید جمع یا تفریق و یک شیفت انجام شود که این دو کار به یکبار با استفاده از concatenation و سیگنال‌های subtract , add انجام می‌شود و یکی از لفت‌ها کم می‌شود و به ابتدای بلاک می‌رود تا کارها دوباره تکرار شود. اگر بیت‌های اولی مانند هم بودند یعنی شیفت بدون انجام عملیات دیگر نیاز است. حال با استفاده از سیگنال amount باید شیفت را انجام دهیم تا حداکثر شیفت ممکن انجام شود. اگر این تعداد شیفت از تعداد شیفت باقی‌مانده کمتر باشد به همان تعداد شیفت انجام می‌شود و این تعداد از شیفت باقی‌مانده کم می‌شود و به ابتدای بلاک می‌رود تا روند ادامه پیدا کند. اما اگر این تعداد از شیف مجاز بیشتر باشد، عدد را به اندازه شیفت باقی‌مانده شیفت می‌دهد، تعداد شیفت باقی‌مانده را صفر میکند و به بلاک ابتدایی می‌رود.



شکل ۱: *ASM Chart of Booth's Algorithm with multiple shifts*

۳ ساخت مدار ضرب کننده

پیشنیازها

17bit multiple shift register

این یک شیفت رجیستر است که برای A از آن استفاده خواهیم کرد. لود آن بر شیفتش اولویت دارد. تنها به راست شیفت می‌دهد. از یک تا ۱۷ بار قابلیت شیفت دارد اما در مدارمان حداکثر از ۸بیت شیفت آن استفاده خواهیم کرد. شیفت به صورت ریاضی صورت می گیرد که به این معنی است که اعداد هنگام شیفت، بیت علامت خود را حفظ خواهند کرد.

```
module shift_register(
    clk,
    data,
    load,
    shift,
    amount,
    out_data
    );

    // Terminal Deceleration
    // clk : the universal clock
    // load : load command
    // shift: shift command (the load is prior to shift)
    // reset: asynchronous reset for register
    input clk, load, shift;
    // data : the input data for load command
    input [16:0] data;
    // amount : the amount of shift for variable shift
    input [4:0] amount;
    // out_data : the output data of the register (assigned memory register)
    output [16:0] out_data;
    // Base register that we use for keeping data
    reg [16:0] memory=0;
    // register keep amount of shift at negative edge
    reg [4:0] amountkeeper;

    // keep amount at last clock
    always @ (negedge clk) amountkeeper <= amount;
    // sending the register value to the output
    assign out_data = memory;

    always @ (posedge clk) begin
    if (load) begin // if load commmand requested
            memory <= data;
    end else if (shift && (|amountkeeper)) begin  // else if shift command was requested
    if (amountkeeper == 1) memory <= {memory[16],memory[16:1]}; // concatenation shift
    else if (amountkeeper == 2) memory <= {memory[16],memory[16],memory[16:2]};
    else if (amountkeeper == 3) memory <= {memory[16],memory[16],memory[16],memory[16:3]};
    else if (amountkeeper == 4) memory <= {memory[16],memory[16],memory[16],memory[16],memory[16:4]};
    else if (amountkeeper == 5) memory <=
       {memory[16],memory[16],memory[16],memory[16],memory[16],memory[16:5]};
    else if (amountkeeper == 6) memory <=
       {memory[16],memory[16],memory[16],memory[16],memory[16],memory[16],memory[16:6]};
    else if (amountkeeper == 7) memory <=
       {memory[16],memory[16],memory[16],memory[16],memory[16],memory[16],memory[16],memory[16:7]};
    else if (amountkeeper == 8) memory <=
    {memory[16],memory[16],memory[16],memory[16],memory[16],memory[16],memory[16],memory[16],memory[16:8]};
        end
    end

endmodule
```

8bit simple register

این ماژول یک رجیستر ۸ بیتی ساده تنها با قابلیت لود است.

```
module register8bit(
    input [7:0] data, input load, input clk,
    output [7:0] out);
    reg [7:0] memory;
    assign out = memory;
    always @ (posedge clk) if (load) memory <= data;
endmodule
```

4bit register with decrease and reset

این ماژول یک رجیستر ۴ بیتی با قابلیت کاهش و ریست سنکرون و بارگذاری خواهد بود. ریست بر کاهش و کاهش بر بارگذاری اولویت دارد.

```
module register4bit(
    input clk, input dec, input load, input resetN, input [3:0] data,
    output [3:0] out
    );
    reg [3:0]memory = 0;
    assign out = memory;
    always @ (posedge clk)
        if (~resetN) memory <= 4'b0;
        else if (dec) memory <= memory - 1;
        else if (load) memory <= data;
endmodule
```

Shift amount finder

این ماژول یک ماژول ترکیبی است که جایگاه اولین بیتی که با بیت‌های ابتدای ورودی تفاوت دارد را پیدا کرده و مقداری که باید شیفت بخورد تا این تفاوت به ابتدای عدد برسد را برمی‌گرداند. این کار به وسیله assign و شرط گذاری روی بیت‌های عدد ورودی انجام می‌شود. روش کار این ماژول به این صورت است که اگر دوبیت ابتدایی متفاوت بودند (xor آنها یک بود) مقدار صفر را بگردارند در غیر اینصورت اگر بیت دوم و سوم متفاوت بود مقدار یک را بگرداند در غیر اینصورت اگر بیت سوم وچهارم متفاوت بود مقدار دو را بگرداند در غیر اینصورت اگر ... .

```
module shift_amount(
    input [16:0] number,
    output [5:0] amount
    );
    assign amount = number[ 0] ^ number[ 1] ? 5'b00000
                  :number[ 1] ^ number[ 2] ? 5'b00001
                  :number[ 2] ^ number[ 3] ? 5'b00010
                  :number[ 3] ^ number[ 4] ? 5'b00011
                  :number[ 4] ^ number[ 5] ? 5'b00100
                  :number[ 5] ^ number[ 6] ? 5'b00101
                  :number[ 6] ^ number[ 7] ? 5'b00110
                  :number[ 7] ^ number[ 8] ? 5'b00111
                  :number[ 8] ^ number[ 9] ? 5'b01000
                  :number[ 9] ^ number[10] ? 5'b01001
                  :number[10] ^ number[11] ? 5'b01010
                  :number[11] ^ number[12] ? 5'b01011
                  :number[12] ^ number[13] ? 5'b01100
                  :number[13] ^ number[14] ? 5'b01101
                  :number[14] ^ number[15] ? 5'b01110:5'b01111;

endmodule
```

DFF

یک دی فلیپ‌فلاپ ساده با سیگنال Q و Q’ و قابلیت ریست می‌سازیم تا از آن برای پیاده‌سازی کنترل یونیت با روش One Hot استفاده کنیم.

```
module dff(
    input clk, input d, input resetN, output q, output qn);
    reg bit = 0;
    assign q = bit;
    assign qn = ~bit;
    always @ (posedge clk or negedge resetN)
        if (~resetN) bit <= 1'b0;
        else bit <= d;
endmodule
```

مدار اصلی

Data Path

در این ماژول تمام پیاده‌سازی‌هایی که مربوط به RTL های درون ASMچارت می‌شود را انجام می‌دهیم. سیگنال‌های کنترلی هر ماژول داخلی استفاده شده به صورت ورودی و کامند و سیگنال‌های استاتوس هم خروجی‌های این ماژول خواهد بود. برای خط سلکت A از عملی مشابه وان‌هات استفاده شده است. هر بیت نمایانگر یکی از حالت‌های داده‌های ورودی رجیستر است.

```
module dp(
    clk, loadA, shiftA, selectA, IN1, IN2, loadM, loadleft, selectleft, resetleft, decleft,
    Sin, Sout, orleft, xorfirsttwo, amountlowerthanleft, secondbit, F, out
    );
    // list of inputs - else than clk, IN1 and IN2 other inputs are commands from CU
    input clk, loadA, shiftA, loadM, loadleft, selectleft, resetleft, decleft, Sin;
    input [2:0] selectA;
    input [7:0] IN1,IN2;
    output Sout, orleft, xorfirsttwo, amountlowerthanleft, secondbit, F;
    output [15:0] out;
    // Assigning M register
     wire signed [7:0]M;
     register8bit mregister(.data(IN2), .load(loadM), .clk(clk),
        .out(M));
    // send S directly to the output for CU
    assign Sout = Sin;
    // 17bit shift register coresponding to A
    wire [16:0]dataA;
    wire [4:0]amount;
    wire [16:0]A;
    wire signed [7:0]a8msb;
    assign a8msb = A[16:9];
    // Subtract and Add signals that always are calculated in combinational way
    wire signed [7:0]add, subtract;
    assign add = a8msb + M;
    assign subtract = a8msb + (~M + 1);
    //Assignment for dataA based on selectA (init, add, subtract)
    assign dataA = selectA[2] ? {8'b00000000, IN1, 1'b0} // init
                 :selectA[0] ? {add[7], add, A[8:1]}    // add
                 :selectA[1] ? {subtract[7], subtract, A[8:1]} // subtract
                 : 17'b00000000000000000; // otherwie zero but this value is never used
    wire [4:0] selected_amount;
    shift_register aregister(
        .clk(clk),
        .data(dataA),
        .load(loadA),
        .shift(shiftA),
        .amount(selected_amount),
        .out_data(A)
     );
     // Assigning the amount
     shift_amount amount_finder( .number(A),.amount(amount));
     // Assigning left register
     wire [3:0]left;
     wire [3:0]dataleft;
     assign selected_amount = (amountlowerthanleft) ? amount : left;
     // assigning left data to select
     assign dataleft = selectleft? left - amount[3:0]: 4'd8;
     register4bit leftregister(.clk(clk),.dec(decleft),.load(loadleft),.resetN(resetleft),.data(dataleft),
        .out(left));
     // Assigning outputs
     assign orleft = |left;
     assign xorfirsttwo = A[0] ^ A[1];
     assign amountlowerthanleft = ~(amount >= left);
     assign secondbit = A[1];
     assign F = ~|left;
     assign out = A[16:1];
endmodule
```

Control Unit

```
module cu(
    clk, S, orleft, xorfirsttwo, amountlowerthanleft, secondbit, resetNinit, resetNmul,
    loadA, shiftA, selectA, loadM, loadleft, selectleft, resetleft, decleft
    );
    input clk, S, orleft, xorfirsttwo, amountlowerthanleft, secondbit,resetNinit, resetNmul;
    output loadA, shiftA, loadM, loadleft, selectleft, resetleft, decleft;
    output [2:0] selectA;

    // creating d filp flops for one hot
    wire d_init, init, d_mul, mul;
    dff initdff(.clk(clk),.d(d_init),.resetN(resetNinit),.q(init));
    dff muldff(.clk(clk),.d(d_mul),.resetN(resetNmul),.q(mul));


    // assigning d inputs
    assign d_init = (init && ~S)
                || (mul && ~orleft)
                || (mul && orleft && ~xorfirsttwo && ~amountlowerthanleft);
    assign d_mul = (mul && orleft && ~xorfirsttwo && amountlowerthanleft)
                || (mul && orleft && xorfirsttwo)
                || (init && S);
    // assigning outputs based on statuses and flip-flops
    // assignments for register A
    assign selectA = {init && S, mul && secondbit, mul && ~secondbit};
    assign loadA = (init && S)
                || (mul && orleft && xorfirsttwo);
    assign shiftA = mul && orleft && ~xorfirsttwo;

    // assignments for register M
    assign loadM = init && S;

    // assignments for register left
    assign loadleft = (init && S)
                    ||(mul && orleft && ~xorfirsttwo && amountlowerthanleft);
    assign decleft = mul && orleft && xorfirsttwo;
    assign resetleft = ~(mul && orleft && ~xorfirsttwo && ~amountlowerthanleft);
    assign selectleft = (~(init && S) && (mul && orleft && ~xorfirsttwo && amountlowerthanleft));

endmodule
```

Booth Multiplier

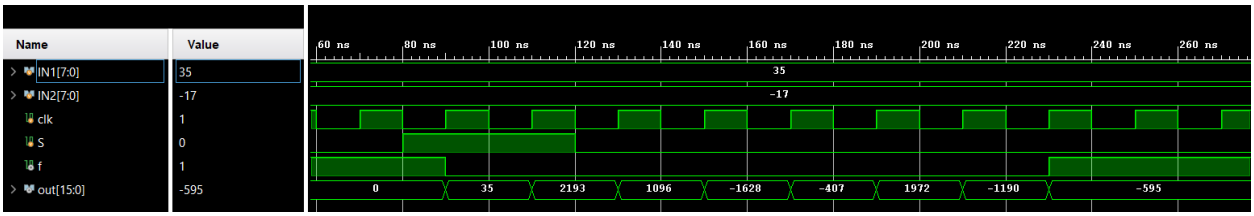
```
module booth(
    input [7:0] IN1,
    input [7:0] IN2,
    input clk,
    input S,
    output f,
    output [15:0] out
    );
    wire loadA, shiftA, loadM, loadleft, selectleft, resetleft,
    decleft, Snet, orleft, xorfirsttwo, amountlowerthanleft, secondbit;
    wire [2:0] selectA;
    reg resetNinit=1, resetNmul = 0;

    dp datapath(
    .clk(clk), .loadA(loadA), .shiftA(shiftA), .selectA(selectA), .IN1(IN1), .IN2(IN2),
    .loadM(loadM), .loadleft(loadleft), .selectleft(selectleft), .resetleft(resetleft), .decleft(decleft),
    .Sin(S), .Sout(Snet), .orleft(orleft), .xorfirsttwo(xorfirsttwo),
    .amountlowerthanleft(amountlowerthanleft), .secondbit(secondbit), .F(f), .out(out)
    );

    cu controlunit(
    .clk(clk), .S(Snet), .orleft(orleft), .xorfirsttwo(xorfirsttwo),
    .amountlowerthanleft(amountlowerthanleft), .secondbit(secondbit),
    .resetNinit(resetNinit), .resetNmul(resetNmul),
    .loadA(loadA), .shiftA(shiftA), .selectA(selectA), .loadM(loadM),
    .loadleft(loadleft), .selectleft(selectleft), .resetleft(resetleft), .decleft(decleft)
    );

    initial begin
        #10 resetNmul <= 1;
    end
endmodule
```

۴ تست مدار

مدار را برای دو سری عدد تست می‌کنیم. برای بررسی بیشتر عددهای دودویی مربوط به هر تغییر در فرایند آورده شده است تا روند ضرب و شیفت‌های چندگانه درون آن مشخص باشند (بیتی که درون پرانتز است آخرین بیتی است که شیفت داده شده است. جزو عدد نیست اما در منطق ضرب تأثیر دارد).

```
module booth_tb;
    reg [7:0] IN1 = 35;
    reg [7:0] IN2 = -17;
    reg clk;
    reg S = 0;
    wire f;
    wire [15:0] out;

    initial clk = 1'b0;
    always #10 clk = ~clk;

    booth uut(
    .IN1(IN1),.IN2(IN2),.clk(clk),.S(S),
    .f(f), .out (out));

    initial begin
        #80 S = 1;
        #40 S = 0;

        #500 $finish;
    end

endmodule
```

۳۵ × −۱۷

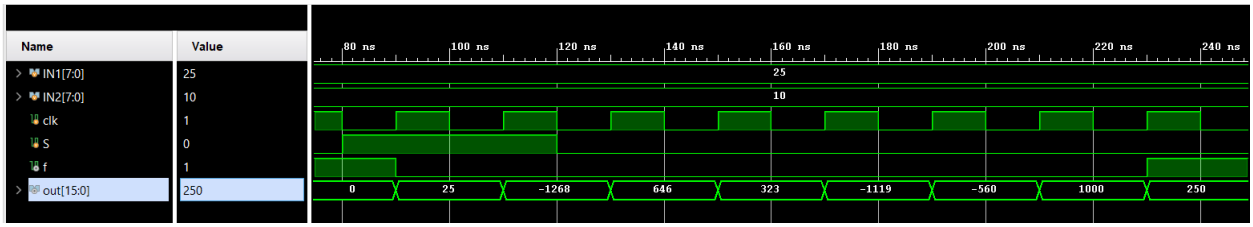


شکل ۲:تست ۱۷−×۳۵

binary sequence:

```
0000000000100011(0): 35          => Subtract + 1 SAR
0000100010010001(1): 2193        => 1 SAR
0000100001001000(1): 1096        => Add + 1 SAR
1111100110100100(0): -1628       => 2 SAR
1111111001101001(0): -407        => Subtract + 1 SAR
0000011110110100(1): 1972        => Add + 1 SAR
1111101101011010(0): -1190       => 1 SAR
1111110110101101(0): -595
```

۲۵ × ۱۰

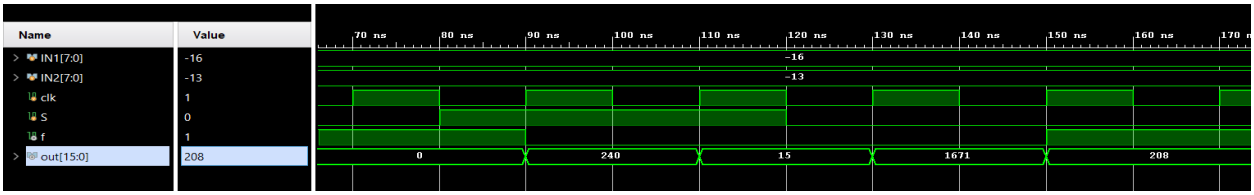


شکل ۳: ۲۵ × ۱۰

binary sequence:

```
0000000000011001(0): 25          => Subtract + 1 SAR
1111101100001100(1): -1268       => Add + 1 SAR
0000001010000110(0): 646         => 1 SAR
0000000101000011(0): 323         => Subtract + 1 SAR
1111101110100001(1): -1119       => 1 SAR
1111110111010000(1): -560        => Add + 1 SAR
0000001111101000(0): 1000        => 2 SAR
0000000011111010(0): 250
```

-۱۶×-۱۳



شکل ۴: -۱۳×-۱۶

binary sequence:

```
0000000011110000(0): 240         => 4 SAR
0000000000001111(0): 15          => Subtract + 1 SAR
0000011010000111(1): 1671        => 3 SAR
0000000011010000(1): 208
```