به نام خدا

آزمایش هشتم

آزمایشگاه طراحی سیستم‌های دیجیتال

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

نویسندگان:

رادین چراغی ۴۰۱۱۰۵۸۱۵

امیرمحمد محفوظی ۴۰۱۱۰۶۴۶۹

سیدعلی جعفری ۴۰۰۱۰۴۸۸۹

تاریخ ارائه تکلیف:

۱۴۰۳/۰۵/۰۸

مقدمه

هدف از انجام این آزمایش طراحی و پیاده‌سازی یک کامپیوتر پایه می‌باشد. برای طراحی این کامپیوتر نیاز به طراحی تعدادی ماژول داریم. این ماژول‌ها عبارتند از:

- ماژول جمع و تفریق اعداد مختلط
- ماژول ضرب اعداد مختلط
- واحد محاسبات و منطق (ALU)
- ماژول حافظه داده (۳۲ کلمه‌ای هر کلمه ۱۶ بیت)
- ماژول حافظه دستورات با قابلیت ذخیره ۳۲ دستور ۱۷ بیتی
- واحد پایپ‌لاین که دستورات را از حافظه دستورات خوانده و عملیات مختلط ورودی را به صورت پایپ‌لاین انجام دهد.

شرح آزمایش

ابتدا به نحوه ذخیره‌سازی اعداد مختلط در حافظه و پیاده‌سازی آن‌ها در زبان وریلاگ می‌پردازیم. همانطور که بالاتر گفته شد حافظه داده را ۳۲ کلمه ۱۶ بیتی درنظر گرفته‌ایم که هر کلمه یک عدد مختلط را نشان می‌دهد. در هر کلمه ۸ بیت پرارزش قسمت حقیقی و ۸ بیت کم‌ارزش قسمت موهومی عدد مختلط می‌باشد. با توجه به این توضیحات فایل macros.v را به صورت زیر پیاده‌سازی کرده و در فایل پروژه قرار می‌دهیم تا بتوانیم در ادامه از ماکروهای موجود در آن در سایر ماژول‌ها بهره ببریم.

```
`define size_spec    [15: 0]
`define Re(x)        x[15:8]
`define Im(x)        x[7:0]
`define signedRe(x)      $signed(`Re(x))
`define signedIm(x)      $signed(`Im(x))
```

به طراحی ماژول جمع و تفریق اعداد مختلط می‌پردازیم. همانطور که از قبل می‌دانیم در هنگام جمع/تفریق دو عدد مختلط، قسمت حقیقی عدد حاصل از جمع/تفریق قسمت‌های حقیقی دو عدد مختلط و قسمت موهومی عدد حاصل نیز از جمع/تفریق قسمت‌های موهومی دو عدد مختلطی که روی آن‌ها عملیات انجام می‌شود بدست می‌آیند. تصویر صفحه بعد این عملیات را نشان می‌دهد. در این ماژول ورودی mode مشخص‌کننده نوع عملیات (جمع/تفریق) می‌باشد که در صورتی که صفر باشد جمع و در غیر این‌صورت تفریق را مشخص می‌کند.

Adding and Subtracting Complex Numbers

cuemath
THE MATH EXPERT

$$(a + ib) + (c + id) = (a + c) + i(b + d)$$

$$(a + ib) - (c + id) = (a - c) + i(b - d)$$

حال با استفاده از زبان وریلاگ ماژول add_sub را به صورت زیر پیاده‌سازی می‌کنیم.

```
`include "macros.v"

module add_sub
(
    input `size_spec a, b,
    input mode, // mode ? sub : add
    output `size_spec out
);

    assign `Re(out) = mode ? `signedRe(a) - `signedRe(b) : `signedRe(a) + `signedRe(b);
    assign `Im(out) = mode ? `signedIm(a) - `signedIm(b) : `signedIm(a) + `signedIm(b);

endmodule
```

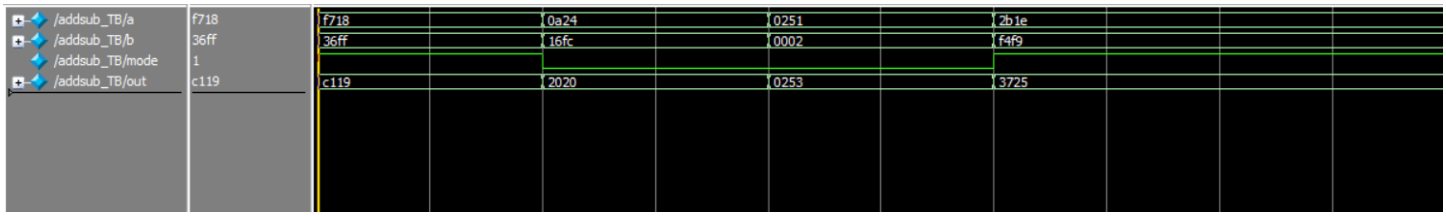
برای این ماژول یک تست‌بنچ طراحی می‌کنیم که در آن با چهار تست مختلف عملکرد ماژول را آزمایش می‌کنیم. تصاویر زیر طراحی این ماژول را نشان می‌دهند.

```
`include "macros.v"

module addsub_TB;

    reg `size_spec a, b;
    reg mode; // mode ? sub : add
    wire `size_spec out;

    add_sub calculator(a, b, mode, out);
```

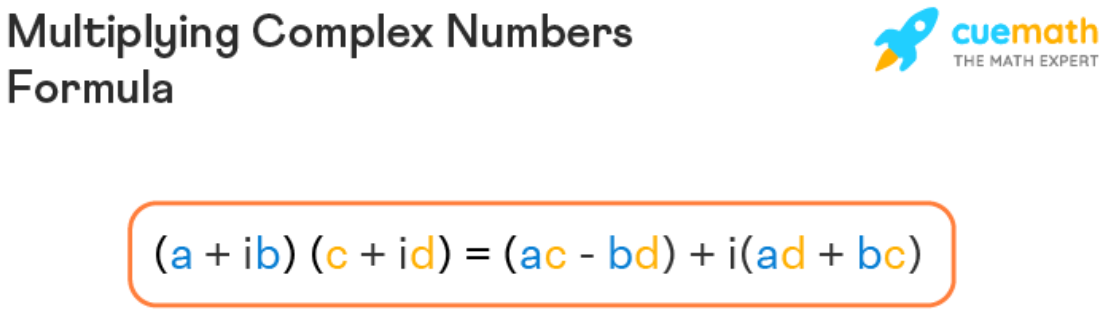
```
initial begin
    `Re(a) = -9;
    `Im(a) = 24;
    `Re(b) = 54;
    `Im(b) = -1;
    mode = 1;
    #10;
    $display("(%d, %d) - (%d, %d) = (%d, %d)",
            `signedRe(a), `signedIm(a), `signedRe(b), `signedIm(b), `signedRe(out), `signedIm(out));

    `Re(a) = 10;
    `Im(a) = 36;
    `Re(b) = 22;
    `Im(b) = -4;
    mode = 0;
    #10;
    $display("(%d, %d) + (%d, %d) = (%d, %d)",
            `signedRe(a), `signedIm(a), `signedRe(b), `signedIm(b), `signedRe(out), `signedIm(out));

    `Re(a) = 2;
    `Im(a) = 81;
    `Re(b) = 0;
    `Im(b) = 2;
    mode = 0;
    #10;
    $display("(%d, %d) + (%d, %d) = (%d, %d)",
            `signedRe(a), `signedIm(a), `signedRe(b), `signedIm(b), `signedRe(out), `signedIm(out));

    `Re(a) = 43;
    `Im(a) = 30;
    `Re(b) = -12;
    `Im(b) = -7;
    mode = 1;
    #10;
    $display("(%d, %d) - (%d, %d) = (%d, %d)",
            `signedRe(a), `signedIm(a), `signedRe(b), `signedIm(b), `signedRe(out), `signedIm(out));
```

این ماژول را در ModelSim شبیه‌سازی می‌کنیم. تصاویر زیر به ترتیب خروجی transcript و waveform نشان می‌دهند.

```
VSIM 5> run -all
# (  -9,   24) - (  54,   -1) = ( -63,   25)
# (  10,   36) + (  22,   -4) = (  32,   32)
# (   2,   81) + (   0,    2) = (   2,   83)
# (  43,   30) - ( -12,   -7) = (  55,   37)
# ** Note: $stop    : C:/Users/ideapad 5/Desktop/E8/add_subTB.v(48)
#    Time: 50 ps  Iteration: 0  Instance: /addsub_TB
```



به طراحی ماژول ضرب اعداد مختلط می‌پردازیم. تصویر زیر چگونگی محاسبه عمل ضرب را در اعداد مختلط نشان می‌دهد.



حال با استفاده از زبان وریلاگ ماژول multiply را به صورت زیر پیاده‌سازی می‌کنیم.

```
`include "macros.v"

module multiply
(
    input `size_spec a, b,
    output `size_spec out
);
    assign `Re(out) = `signedRe(a) * `signedRe(b) - `signedIm(a) * `signedIm(b);
    assign `Im(out) = `signedRe(a) * `signedIm(b) + `signedIm(a) * `signedRe(b);

endmodule
```

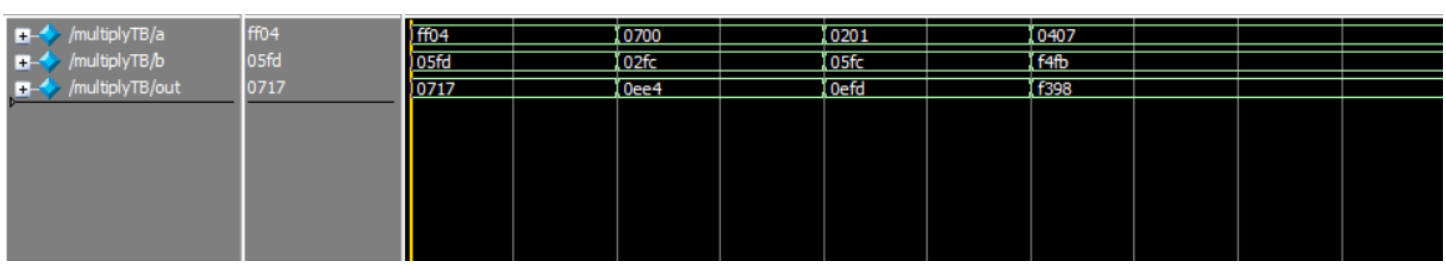
برای این ماژول یک تست‌بنچ طراحی می‌کنیم که در آن با چهار تست مختلف عملکرد ماژول را آزمایش می‌کنیم. تصاویر زیر طراحی این ماژول را نشان می‌دهند.

```
`include "macros.v"

module multiplyTB;
    reg `size_spec a, b;
    wire `size_spec out;

    multiply calculator(a, b, out);
```

```
initial begin
    `Re(a) = -1;
    `Im(a) = 4;
    `Re(b) = 5;
    `Im(b) = -3;
    #10;
    $display("(%d, %d) * (%d, %d) = (%d, %d)",
            `signedRe(a), `signedIm(a), `signedRe(b), `signedIm(b), `signedRe(out), `signedIm(out));

    `Re(a) = 7;
    `Im(a) = 0;
    `Re(b) = 2;
    `Im(b) = -4;
    #10;
    $display("(%d, %d) * (%d, %d) = (%d, %d)",
            `signedRe(a), `signedIm(a), `signedRe(b), `signedIm(b), `signedRe(out), `signedIm(out));

    `Re(a) = 2;
    `Im(a) = 1;
    `Re(b) = 5;
    `Im(b) = -4;
    #10;
    $display("(%d, %d) * (%d, %d) = (%d, %d)",
            `signedRe(a), `signedIm(a), `signedRe(b), `signedIm(b), `signedRe(out), `signedIm(out));

    `Re(a) = 4;
    `Im(a) = 7;
    `Re(b) = -12;
    `Im(b) = -5;
    #10;
    $display("(%d, %d) * (%d, %d) = (%d, %d)",
            `signedRe(a), `signedIm(a), `signedRe(b), `signedIm(b), `signedRe(out), `signedIm(out));

    #10 $stop;
end
```

این ماژول را در ModelSim شبیه‌سازی می‌کنیم. تصاویر زیر به ترتیب خروجی transcript و waveform نشان می‌دهند.

```
# (  -1,    4) * (   5,   -3) = (   7,   23)
# (   7,    0) * (   2,   -4) = (  14,  -28)
# (   2,    1) * (   5,   -4) = (  14,   -3)
# (   4,    7) * ( -12,   -5) = ( -13, -104)
# ** Note: $stop    : C:/Users/ideapad 5/Desktop/E8/multiplyTB.v(42)
#    Time: 50 ps  Iteration: 0  Instance: /multiplyTB
```



اکنون نوبت به طراحی ماژول ALU می‌رسد. این ماژول دو عدد مختلط a و b را ورودی گرفته و با توجه به ورودی دوبیتی alu_op عملیات مورد نظر را روی این دو عدد انجام داده و حاصل را در خروجی out قرار می‌دهد. این ماژول که ابتدا از دو ماژول add_sub و multiply دو نمونه می‌گیرد. با توجه به ورودی alu_op اگر بیت پرارزش آن صفر باشد عملیات جمع یا تفریق و در غیر این‌صورت عملیات ضرب انجام می‌شود. همچنین بیت کم‌ارزش آن مشخص‌کننده نوع عملیات جمع/تفریق می‌باشد که در صورتی که صفر باشد جمع و در غیر این‌صورت تفریق را مشخص می‌کند. تصویر زیر این طراحی را نشان می‌دهد.

```
`include "macros.v"

module ALU
(
    input `size_spec a, b,
    input [1:0] alu_op,
    output `size_spec out
);
    wire `size_spec addsub_out, multiply_out;
    add_sub addsub(a,b, alu_op[0], addsub_out);
    multiply mul(a, b, multiply_out);

    assign out = alu_op[1] ? multiply_out : addsub_out;

endmodule
```

همانطور که قبل‌تر توضیح داده شد، در این کامپیوتر دو حافظه جداگانه داریم که یکی برای دستورات و دیگری برای ذخیره و نگهداری اعداد مختلط می‌باشد.

ابتدا حافظه مربوط به نگهداری دستورات را طراحی می‌کنیم. هر دستور در کامپیوتر پایه ما ۱۷ بیتی می‌باشد. دو بیت اول آن نشان دهنده opcode، بیت سوم تا هفتم مشخص‌کننده write_addr، بیت هشتم تا یازدهم مشخص کننده read_addr1 و بیت دوازدهم تا هفدهم مشخص‌کننده read_addr2 می‌باشد.

ماژول IF_mem یک ورودی clk دارد که کلاک آن می‌باشد. همچنین rstN ورودی دیگر این ماژول می‌باشد که ریست این حافظه بوده و فعال پایین است.

خروجی‌های این ماژول همان بخش‌های مختف دستور پس از دیکود می‌باشند که عبارتند از: خروجی دو بیتی opcode، خروجی پنج بیتی write_addr، خروجی پنج بیتی read_addr1 و خروجی پنج بیتی read_addr2.

در این ماژول یک آرایه mem با عمق ۳۲ و پهنای ۱۷ تعریف می‌کنیم که همان حافظه دستورات می‌باشد. همچنین یک رجیستر program counter تعریف می‌کنیم که حاوی آدرس دستور بعدی می‌باشد و در هر کلاک در صورتی که rstN یک باشد، یکی

به آن اضافه می‌شود. همچنین در صورتی که rstN صفر باشد، صفر خواهد شد. تصویر زیر طراحی این ماژول را با وریلاگ نشان می‌دهد.

```
module IF_mem
(
    input           clk,
    input           rstN,
    output  [1:0]   opcode,
    output  [4:0]   write_addr,
    output  [4:0]   read_addr1,
    output  [4:0]   read_addr2
);
    reg [0:16] mem [31:0]; // opcode: 2 bit, write_addr: 5 bit, read_addr1: 5 bit, read_addr2: 5 bit
    reg [4:0] pc;

    assign opcode = mem[pc][0:1];
    assign write_addr = mem[pc][2:6];
    assign read_addr1 = mem[pc][7:11];
    assign read_addr2 = mem[pc][12:16];

    always @(posedge clk or negedge rstN)
        if(!rstN) pc <= 0;
        else pc <= pc + 1;


endmodule
```

حال حافظه مربوط به نگهداری اعداد مختلط را طراحی می‌کنیم.

ماژول Data_mem دو ورودی پنج بیتی read_addr1 و read_addr2 را دارد که آدرس دو عدد خواسته شده را در حافظه مشخص می‌کنند. همچنین دارای دو write_data و write_addr را می‌باشد که به ترتیب مشخص کننده داده‌ای که قرار است در حافظه نوشته شود و آدرس محل نوشتن می‌باشند. خروجی‌های این ماژول read_data1 و read_data2 می‌باشند که همان اعداد خوانده شده از حافظه پس از دسترس به آن می‌باشند.

در این ماژول یک آرایه mem با عمق ۳۲ و پهنای ۱۶ تعریف می‌کنیم که همان حافظه داده‌ها می‌باشد. تصویر صفحه بعد طراحی این ماژول را با وریلاگ نشان می‌دهد.

```
`include "macros.v"

module Data_mem
(
    input   [4:0]       read_addr1,
    input   [4:0]       read_addr2,
    input   `size_spec  write_data,
    input   [4:0]       write_addr,
    output  `size_spec    read_data1,
    output  `size_spec    read_data2
);

reg `size_spec mem [31:0];

assign read_data1 = mem[read_addr1];
assign read_data2 = mem[read_addr2];
always @(*) begin
   mem[write_addr] <= write_data;
end

endmodule
```

نوبت به طراحی ماژول pipeline می‌رسد. ورودی‌های این ماژول clk و rstN می‌باشند که کلاک و ریست فعال پایین مدار هستند.

پایپ‌لاین این کامپیوتر از چهار مرحله Instruction fetch، Data_mem و Execute(ALU) و Write Back تشکیل شده است. این ماژول به لبه بالارونده کلاک حساس می‌باشد. روش کار اجرای یک دستور به این صورت می‌باشد که ابتدا در مرحله اول دستور از حافظه مربوط به دستورات واکشی شده، دیکود می‌شود و ورودی‌های لازم به حافظه اعداد داده می‌شود. سپس در مرحله بعدی دو عدد از حافظه با توجه به read_addr1 و read_addr2 که از دستور به دست آمده‌اند، خوانده شده و به واحد منطق و محاسبات داده می‌شوند. در مرحله Execute واحد منطق و محاسبه عملیات را بر اساس opcode دستور انجام داده و خروجی بدست آمده از ALU به عنوان ورودی write_data به حافظه، داده می‌شود. در نهایت در مرحله Write Back، عدد حاصل در حافظه در آدرس write_addr که از دستور خوانده شد، نوشته می‌شود. نکته قابل توجه این می‌باشد که برای نگهداری از اجزای مختلف دستور پس از دیکود در کلاک‌های متوالی، نیاز به رجیسترهای موقتی tmp_op، tmp1_write_addr و tmp2_write_addr داریم.

همچنین به این نکته باید توجه کرد که در هر مرحله و کلاک (به جز کلاک‌های اولیه)، سه دستور در پایپ‌لاین وجود دارند. طراحی این ماژول در تصاویر صفحه بعد آورده شده است.

```
`include "macros.v"

module pipeline
(
    input clk,
    input rstN
);
    // IF_mem signals
    wire [1:0] ins_op;
    wire [4:0] ins_write_addr, ins_read_addr1, ins_read_addr2;

    // Data_mem signals
    wire `size_spec mem_read_data1, mem_read_data2;
    reg `size_spec mem_write_data;
    reg  [4:0]  mem_read_addr1, mem_read_addr2, mem_write_addr;

    // ALU signals
    reg [1:0] alu_op;
    wire `size_spec alu_out;
    reg  `size_spec alu_read_data1, alu_read_data2;

    IF_mem IF(clk, rstN, ins_op, ins_write_addr, ins_read_addr1, ins_read_addr2);
    Data_mem MEM(mem_read_addr1, mem_read_addr2, mem_write_data, mem_write_addr, mem_read_data1, mem_read_data2);
    ALU alu(alu_read_data1, alu_read_data2, alu_op, alu_out);

    // tmp registers
    reg [1:0] tmp_op;
    reg [4:0] tmp1_write_addr, tmp2_write_addr;
```

```
    always @(posedge clk or negedge rstN) begin
        if (rstN) begin
            // IF
            tmp_op <= ins_op;
            tmp1_write_addr <= ins_write_addr;
            mem_read_addr1 <= ins_read_addr1;
            mem_read_addr2 <= ins_read_addr2;

            // MEM
            alu_read_data1 <= mem_read_data1;
            alu_read_data2 <= mem_read_data2;
            alu_op <= tmp_op;
            tmp2_write_addr <= tmp1_write_addr;

            // ALU
            mem_write_addr <= tmp2_write_addr;
            mem_write_data <= alu_out;
        end
end

endmodule
```

برای این ماژول یک تست‌بنچ طراحی می‌کنیم. جهت آزمایش بهتر مدار بایستی حافظه‌های دستورات و اعداد را مقدار دهی اولیه کنیم. برای این کار دو فایل IF_INIT و DATA_INIT را تولید می‌کنیم که یکی از آن‌ها شامل ۱۵ دستور ۱۷ بیتی می‌باشد و دیگری دارای ۳۲ عدد ۱۶ بیتی می‌باشد که هر کدام مشخص‌کننده یک عدد مختلط می‌باشند. در هر یک از فایل‌ها با استفاده از کامنت، محتویات هر خط (دستور/عدد) توضیح داده شده است و این فایل‌ها در پیوست آورده شده‌اند.در تست‌بنچ ابتدا حافظه‌های

دستورات و اعداد را مقدار دهی اولیه کرده و سپس تمامی ۱۵ دستور را اجرا می‌کنیم. در نهایت محتویات حافظه پس از انجام دستورات را در یک فایل جدید به نام new_mem قرار می‌دهیم. تصویر زیر ماژول تست‌بنچ را نشان می‌دهد.

```
module pipelineTB;

reg rstN = 0;
reg clk = 0;
pipeline pl(clk, rstN);

always #5 clk = ~clk;

initial begin
    $readmemb("IF_INIT.txt", pl.IF.mem, 0, 14);
    $readmemb("DATA_INIT.txt", pl.MEM.mem, 0, 31);

    #40 rstN = 1;
    wait(pl.IF.pc == 15);
    $writememb("new_mem.txt", pl.MEM.mem);
    $stop;
end

endmodule
```

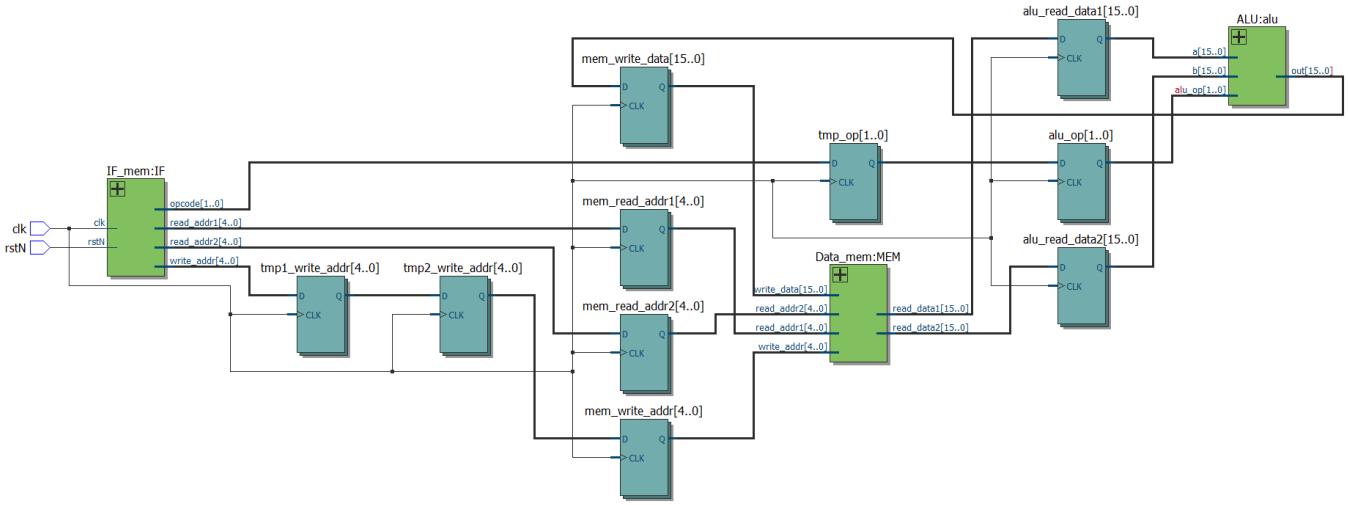
جهت بررسی درست خروجی‌ها به فایل new_mem مراجعه شود.

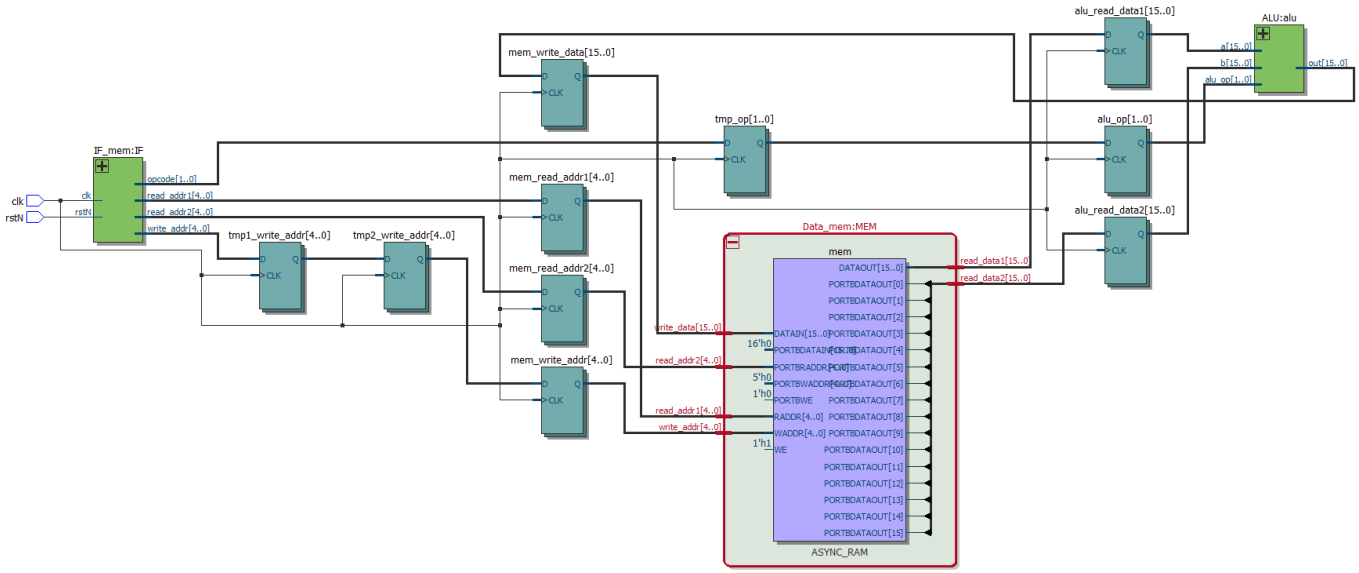
خروجی **flow summary**

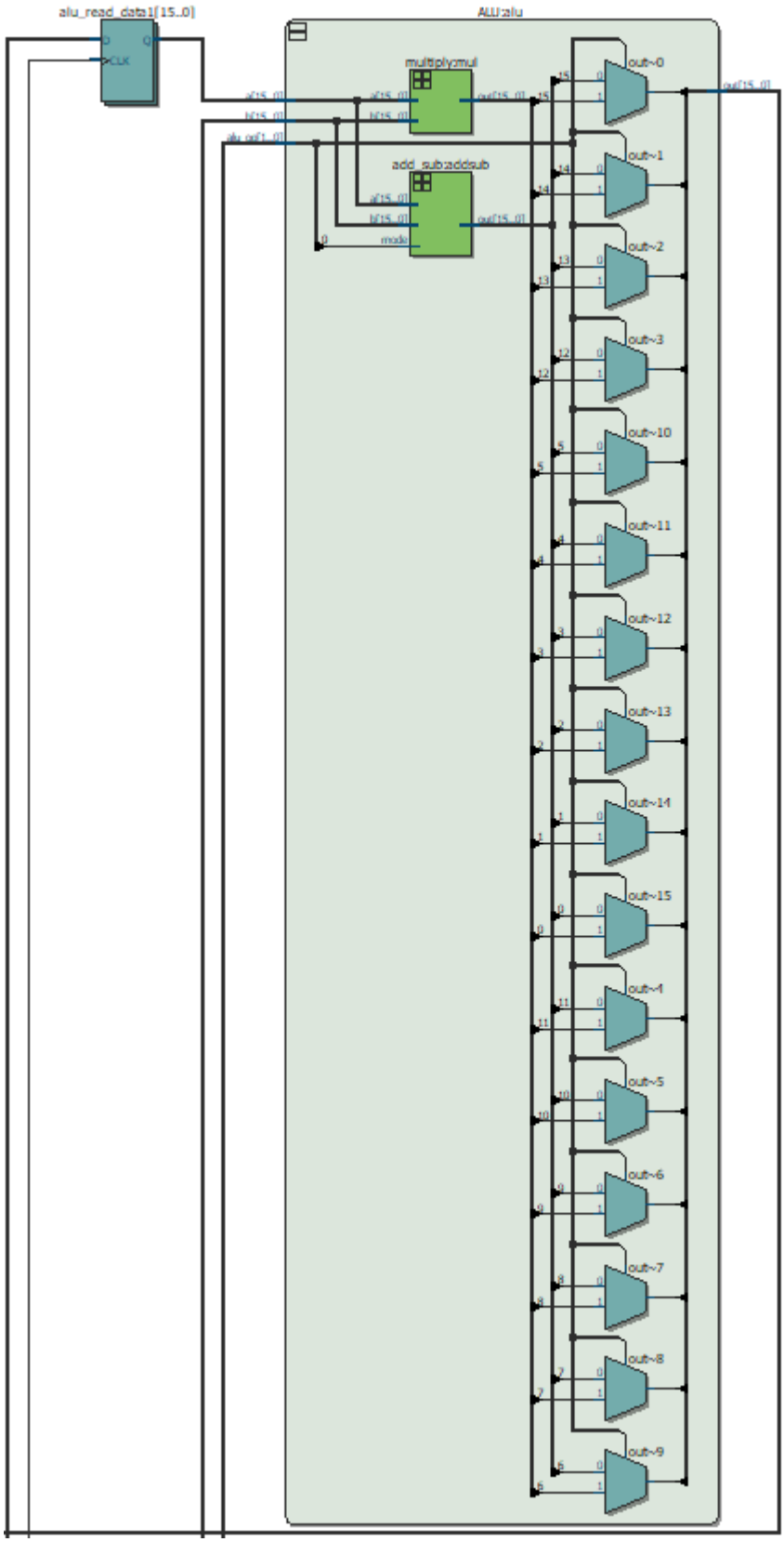
Flow Summary	
Flow Status	Successful - Mon Jul 29 14:20:12 2024
Quartus II 64-Bit Version	13.1.0 Build 162 10/23/2013 SJ Web Edition
Revision Name	basic_comp
Top-level Entity Name	pipeline
Family	Cyclone IV GX
Total logic elements	0 / 14,400 (0 %)
Total combinational functions	0 / 14,400 (0 %)
Dedicated logic registers	0 / 14,400 (0 %)
Total registers	0
Total pins	2 / 81 (2 %)
Total virtual pins	0
Total memory bits	0 / 552,960 (0 %)
Embedded Multiplier 9-bit elements	0
Total GXB Receiver Channel PCS	0 / 2 (0 %)
Total GXB Receiver Channel PMA	0 / 2 (0 %)
Total GXB Transmitter Channel PCS	0 / 2 (0 %)
Total GXB Transmitter Channel PMA	0 / 2 (0 %)
Total PLLs	0 / 3 (0 %)
Device	EP4CGX15BF14C6
Timing Models	Final

خروجی RTL Viewer

در تصاویر زیر خروجی rtl viewer مدار نشان داده شده است. فایل pdf در پیوست آورده شده است.






alu_read_data1[15..0]
D
Q
CLK
ALU:alu
multiplymul
a[15..0]
b[15..0]
out[15..0]
add_subaddsub
a[15..0]
b[15..0]
mode
out[15..0]
alu_op[1..0]
out~0
out~1
out~2
out~3
out~10
out~11
out~12
out~13
out~14
out~15
out~4
out~5
out~6
out~7
out~8
out~9
out[15..0]