به نام خدا

آزمایشگاه طراحی سیستم‌های دیجیتال

گزارش کار هشتم

ALU اعداد مختلط

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

بهار ۱۴۰۱

استاد:

علیرضا اجلالی

دستیار آموزشی:

سحر رضاقلی

نویسندگان:

هیربد بهنام

۹۹۱۷۱۳۳۳

عرفان مجیبی

۹۹۱۰۵۷۰۷

علی نظری

۹۹۱۰۲۴۰۱

فهرست

مقدمه

در این آزمایش می‌خواهیم یک ALU مربوط به اعداد مختلط بسازیم و این واحد دارای چندین بخش است که دو بخش محاسباتی آن بخش جمع کننده و ضرب کننده است و سپس در طول آزمایش با تنها یکبار استفاده از این بخش‌ها و به شکل pipeline مدار را طراحی می‌کنیم. در این آزمایش هر بخش از عدد، یعنی هر دو بخش حقیقی و موهومی اعداد مختلط ۴ بیتی در نظر گرفته شده‌اند. این مدار قرار است که به شکل pipeline عملیات را انجام دهد یعنی در یک کلاک، واحدها در صورت مهیا بودن شرایط، به شکل موازی کار کنند. یعنی طبق درس معماری کامپیوتر، این pipeline دارای چندین Stage است و کارها بین آن‌ها پخش می‌شود.

گزارش آزمایش

بخش ۱. توضیحات مربوط به ALU و نحوه طراحی

نحوه محاسبه

می‌دانیم که یک عدد مختلط به شکل a+bi نوشته می‌شود که همانطور که گفته شد، این a و b ۴ بیتی هستند پس در مجموع برای هر عدد مختلط به ۸ بیت نیاز داریم که ۴ بیت اول برای بخش حقیقی و ۴ بیت دوم برای بخش موهومی است. طبق محاسباتی که از ریاضی ۱ می‌دانیم، روابط زیر برقرار است:

```
(a+bi) + (c+di) = (a+c) + (b+d)i
(a+bi) - (c+di) = (a-c) + (b-d)i
(a+bi) * (c+di) = (ac-bd) + (ad+bc)i
```

شکل ۱: روابط مربوط به اعداد مختلط

پس برای عملیات ریاضی، ما نیاز به ۳ بخش اصلی جمع و تفریق و ضرب داریم که برای این بخش‌ها ماژول جداگانه در نظر می‌گیریم. البته ماژول جمع و تفریق می‌تواند یکی باشد و فقط یک فلگ ارسال شود که جمع می‌خواهیم یا تفریق ولی به شکل کلی خیلی تفاوتی ندارد. ورودی هر کدام از این‌ها هم دو عدد ۸ بیتی است. در ALU هم مانند instruction ها که در درس ساختار و زبان کامپیوتر دیدیم، دارای ۳ بخش اصلی در این آزمایش هستند که این بخش‌ها شامل opcode و عملوند اول و عملوند دوم است که با توجه به opcode مشخص می‌شود که چه کاری روی این عملوندها باید انجام شود.
برای بخش pipeline همانطور که از درس معماری کامپیوتر می‌دانیم، ۴ عمل اصلی داریم که عبارت‌اند از fetch و decode و execute و write back که همه‌ی این بخش‌ها در یک ماژول واحد نوشته شده‌اند و انجام می‌شوند. در بخش fetch ما حساس به لبه بالارونده هستیم و با هر کلاک، بر حسب آدرسی که مشخص شده است، دستور را از حافظه می‌گیریم و آن را نگه می‌داریم. در بخش decode همان دستور fetch شده را می‌گیریم و بخش‌های مختلف آن که شامل آپکد و عملوند اول و دوم است را جدا می‌کنیم و این خروجی‌ها به بخش execute داده می‌شود و در این بخش، موارد گفته شده به ALU داده می‌شود و خروجی حاصل می‌شود. در کلاک بعدی هم خروجی به بخش write back می‌رود تا در out نوشته شود. درون خود ALU هم pipeline انجام می‌شود و بخش‌های مختلف آن در صورت امکان به شکل موازی کار خود را انجام می‌دهند. البته به شکل کلی، خروجی هم در جایی از حافظه نوشته می‌شود که آن هم در instruction مشخص می‌شود ولی چون در اینجا یک حافظه بسیار ساده طراحی می‌شود، از این مورد چشم‌پوشی می‌شود و تنها در یک رجیستر گذاشته می‌شود خروجی مربوطه.
در این آزمایش دستورات موجود را به شکل دستی درون خود ماژول memory گذاشته‌ایم و از آنجا دستورات را بر می‌داریم. می‌شود به این شکل هم عمل کرد که در هر کلاک ورودی را چک کنیم و بر حسب دستور ورودی دستوری را اجرا کنیم ولی چون در صورت آزمایش چیزی نیامده است، به همان شکل درون ماژول می‌گذاریم که البته این‌ها قابل تغییر هم می‌باشند به راحتی. در واقع چون کد باید قابل سنتز باشد، به این شکل طراحی کردیم وگرنه میشد داخل بلاک initial هم مقداردهی را انجام داد. دستورات همانطور که گفته شد، دارای دو تا ۸ بیت عملوند هستند و چون کارهای مدار و در واقع opcode های آن ۳ نوع بیشتر نیست، می‌توان بخش opcode را با ۲ بیت نمایش داد. پس در مجموع، instruction ما شامل ۱۸ بیت است که به شکل زیر مشخص می‌شود:

Opcode(2)	First complex number(8)	Second complex number(8)

شکل ۲: دستورالعمل

برای opcode ها هم همانطور که گفته شد، دارای ۳ نوع دستور add و sub و multipy است که به شکل زیر در opcode ها مشخص می‌شود:

Opcode	Translation
00	Sub
01	Add
10	Multiply

شکل ۳: آپکد مربوط به دستورالعمل‌ها

بخش ۲. کد وریلاگ و طراحی بخش‌های موجود

ماژول adder

در این ماژول ما می‌خواهیم عملیات جمع و همچنین تفریق مربوط به ALU را انجام دهیم پس ورودی‌های این ماژول عبارت‌اند از a و b که همان اعداد مختلط ما هستند که عملیات روی آن‌ها باید انجام شود و یک ورودی یک بیتی هم داریم که مشخص می‌کند عملیات جمع باید انجام شود یا تفریق. و این بیت اگر یک باشد یعنی عملیات جمع است و اگر صفر باشد یعنی عملیات تفریق است. خروجی این ماژول هم output است که همان حاصل جمع یا تفریق دو عدد a و b است.
دقت شود که بخش‌های حقیقی و موهومی عددها جداگانه داده می‌شوند و برای همین ورودی‌ها و همچنین خروجی، ۴ بیتی هستند.
کد این ماژول، مانند زیر است:

```
module adder(input add_diff_not, input [3:0] a, input [3:0] b, output
    [3:0] out);
  assign out = add_diff_not ? (a + b) : (a - b);
endmodule
```

ماژول multiplier

این ماژول هم وظیفه انجام عملیان ضرب داخل ALU را دارد و در این بخش هم دو ورودی ۴ بیتی a و b داریم که همان عددهایی هستند که باید ضرب شوند و خروجی هم در out قرار دارد و حاصل ضرب این دو عدد است. کد این بخش به شکل زیر است:

```
module multiplier(input [3:0] a, input [3:0] b, output [3:0] out);
  assign out = a * b;
endmodule
```

ماژول memory

حال به بخش memory می‌رسیم که بخشی است که دستورات از آن خوانده می‌شود و هر دستور هم ۳ بخش اصلی دارد که همان opcode و دو operand مربوط به آن دستور است. در این آزمایش که حافظه ۳۲ آدرس دارد، پس یکی از مهم‌ترین ورودی‌های این ماژول همان address است که ۵ بیتی است و با آن مشخص می‌شود که به کدام خط از حافظه می‌خواهیم دسترسی داشته باشیم و خروجی هم یک رجیستر ۱۸ بیتی است چون هر خط از حافظه شامل ۱۸ بیت است چون هر کدام از operand ها دارای ۸ بیت هستند و خود opcode هم ۲ بیتی است. پس خروجی این ماژول یک رجیستر ۱۸ بیتی است. در مورد opcode ها هم گفته شد که ۰۰ به معنای تفریق و ۰۱ به معنای جمع و در نهایت هم ۱۰ به معنای ضرب است. در ادامه هم ۴ عدد ۴ بیتی می‌آیند که از چپ به راست، آن‌ها را a و b و c و d می‌نامیم. و منظور از هر خط آدرس، عبارت (a+bi) OPCODE (c+di) است. و Opcode هم منظور یکی از عملیات‌های جمع یا تفریق یا ضرب است. عملیات این بخش توسط ALU انجام می‌شود و نتیجه آن در خروجی ALU ظاهر می‌شود و نحوه به دست آمدن از نظر ریاضی هم قبل‌تر گفته شد. درون memory هم توسط ساختار if می‌توانیم بر حسب آدرس ورودی، خط مدنظر را برگردانیم و دستوراتی که در این ماژول نوشته شده‌اند، کاملا تصادفی هستند و قابلیت تغییر هم دارند. کد این بخش به شکل زیر است:

```
module memory(input [4:0] address, output reg [17:0] instruction);
  always @(address)
```

```
  begin
    if (address == 0) instruction = 18'b01_1111_0001_0001_0010;
    if (address == 1) instruction = 18'b00_0111_0001_0001_0011;
    if (address == 2) instruction = 18'b00_0001_0001_0001_0010;
    if (address == 3) instruction = 18'b10_0011_0001_0001_0010;
    if (address == 4) instruction = 18'b01_0011_0011_0011_0000;
    if (address == 5) instruction = 18'b00_0011_0001_0001_0000;
    if (address == 6) instruction = 18'b00_0011_0001_0001_0010;
    if (address == 7) instruction = 18'b10_0011_0001_0001_0010;
    if (address == 8) instruction = 18'b01_0011_0001_0001_0011;
    if (address == 9) instruction = 18'b10_0011_0001_0001_0011;
    if (address == 10) instruction = 18'b10_0011_0001_0011_0010;
    if (address == 11) instruction = 18'b10_0011_0011_0001_0010;
    if (address == 12) instruction = 18'b10_0011_0011_0001_0010;
    if (address == 13) instruction = 18'b10_0001_0001_0001_0010;
    if (address == 14) instruction = 18'b01_0101_0011_0001_0011;
    if (address == 15) instruction = 18'b01_1000_0001_0001_0110;
    if (address == 16) instruction = 18'b10_0011_0011_0001_0010;
    if (address == 17) instruction = 18'b00_0011_0001_0001_0011;
    if (address == 18) instruction = 18'b00_1000_0001_0001_0110;
    if (address == 19) instruction = 18'b01_0011_0001_0001_0011;
    if (address == 20) instruction = 18'b10_0011_0001_0001_0011;
    if (address == 21) instruction = 18'b10_0001_0001_0001_0010;
    if (address == 22) instruction = 18'b10_0011_0010_0010_0011;
    if (address == 23) instruction = 18'b00_0011_0001_0001_0010;
    if (address == 24) instruction = 18'b01_0011_1101_0001_0000;
    if (address == 25) instruction = 18'b00_0011_0010_0001_0111;
    if (address == 26) instruction = 18'b10_0011_0001_0011_0001;
    if (address == 27) instruction = 18'b01_0011_0001_0001_0010;
    if (address == 28) instruction = 18'b01_0011_0111_0001_0001;
    if (address == 29) instruction = 18'b01_0011_0111_0001_0010;
    if (address == 30) instruction = 18'b00_0011_0111_0001_0010;
    if (address == 31) instruction = 18'b01_1111_1111_0001_0010;
  end
endmodule
```

ماژول ALU

حال به بخش اصلی این آزمایش می‌رسیم. این بخش شامل چند Stage است که هر کدام جداگانه و کامل توضیح داده خواهد شد. این ماژول دارای ورودی‌های reset و clock و خروجی بخش حقیقی و خروجی بخش موهومی است که کلاک برای همان فعالیت pipeline لازم است و خروجی ما شامل دو بخش حقیقی و موهومی است که به شکل جداگانه در خروجی داده می‌شوند و همچنین یک خروجی ready هم در نظر می‌گیریم که فقط زمانی یک می‌شود که خروجی هر دو بخش حقیقی و موهومی آماده باشد و زمانی که صفر باشد، یعنی هنوز آماده نیستند و قابلیت گزارش خروجی را نداریم. این بخش از ورودی‌ها به شکل زیر تعریف

می‌شوند و چون این بخش کمی گسترده است، کدها قسمت قسمت گذاشته می‌شوند:

```
(input reset, input clk, output reg [3:0] real_part, output reg [3:0]
    imaginary_part, output ready)
```

همانطور که قبل‌تر گفته شد، pipeline طراحی شده داری سه stage اصلی است که اولی همان fetch است و دومی بخش ضرب و سومی بخش جمع و تفریق است. به شکل کلی می‌دانیم که همه instruction ها باید تعداد stage یکسانی را طی کنند تا pipeline به مشکل بر نخورد. چون اگر یکی از instruction بخواهد تعداد stage کمتری داشته باشد، آنگاه ممکن است دو دستور به شکل همزمان به یک stage برسند و کار این stage را با خلل مواجه کنند. به عنوان مثال، در دستور جمع و تفریق خب ما نیازی نداریم که عمل ضربی انجام دهیم و عملا به Stage مربوط به ضرب نیازی نداریم ولی در هر صورت باید وارد این Stage شویم و در آن انقدر صبر کنیم تا stage بعدی کاملا خالی شود و آمادگی خودش برای رسیدن دستور بعدی را اعلام کند وگرنه به مشکل بر می‌خوریم. پس به شکل کلی باید وارد آن stage حتی اگر کاری در آن نداریم بشویم و منتظر بمانیم و به هیچ عنوان نمی‌توان به شکل کلی آن stage را رد کرد. برای مدیریت کردن این موضوع، برای هر کدام از Stage های ضرب و جمع، یک wire در نظر گرفته‌ایم که صفر بودن آن‌ها به معنای بیکار بودن آن stage است و یک بودن آن‌ها به معنای پر بودن آن stage است و or شده‌ی این دو wire را در wire دیگری می‌گذاریم تا به شکل کامل روی بخش ضرب و جمع، اشراف داشته باشیم. هرگاه این متغیر یک باشد یعنی بخش ضرب یا جمع هنوز کارش را تمام نکرده است و دستور بعدی نمی‌تواند وارد این بخش‌ها شود و هرگاه این متغیر صفر بود یعنی می‌توان کار جدیدی به این بخش‌ها داد. کد این بخش به شکل زیر است:

```
wire wait_all;
wire wait_mult;
wire wait_add;
assign wait_all = wait_mult || wait_add;
```

حال به stage نخست یا همان fetch می‌رسیم که اول کد آن را قرار می‌دهیم و از روی آن به نحوه عملکرد این بخش می‌پردازیم:

```
  reg [4:0] PC;
  wire [17:0] instruction;
  reg [17:0] instruction_in_register;
  memory mem(PC, instruction);
  always @(posedge clk)
  begin
    if (reset) PC <= 0;
    else
      if (!wait_all)
      begin
        instruction_in_register <= instruction;
        PC <= PC + 1'b1;
      end
  end
```

در این بخش، یک متغیر ۵ بیتی داریم که برای دسترسی به محتوای حافظه است و در واقع دارد خط حافظه را مشخص می‌کند. یک wire ۱۸ بیتی هم داریم که دستورالعمل واکشی شده را در آن نگه می‌داریم و به این دلیل به شکل wire قرار داده‌ایم که ورودی ماژول memory است و چون محتوای این دستورالعمل در همه فازها نیاز داریم، باید آن را در یک رجیستر هم ذخیره کنیم

تا برایمان بماند. مقدار این wire هم جلوتر به نمونه‌ی memory داده می‌شود و همانطور که در کد بخش memory دیدیم، دستورالعمل مورد مظر در این wire قرار داده می‌شود. حال به لبه بالارونده کلاک دقت می‌کنیم. اگر reset یک بود، یعنی کل مدار باید reset شود و این موضوع با صفر کردن PC انجام می‌شود چون این متغیر دارد به خط آدرس اشاره می‌کند و با صفر کردن آن یعنی داریم از اول شروع می‌کنیم و مدار را reset کرده‌ایم. اگر reset صفر بود، آنگاه به همان متغیر wait-all توجه می‌کنیم اگر این متغیر یک بود، یعنی کار ضرب و جمع قبلی تمام نشده باید در همین بخش از pipeline منتظر بمانیم تا جلوتر خالی شود و تا لبه بالارونده کلاک بعدی باید این انتظار را انجام دهیم. در غیر این صورت، یعنی اگر wait-all هم صفر بود، یعنی آماده هستیم تا دستور بعدی را هم به بخش ضرب و جمع بدهیم. در واقع با صفر بودن wait-all باید هم مقدار دستورالعملی که الان در wire است را به یک reg منتقل کنیم تا مقدار آن را در مرحله‌های بعد هم داشته باشیم. همچنین چون این دستورالعمل فعلی را واکشی کردیم، PC را یکی زیاد می‌کنیم تا در کلاک بعدی، به سراغ دستور خط بعد برویم.
حال به سراغ stage بعد که ضرب است می‌رویم. در این بخش هم نخست کد آن را قرار می‌دهیم و از روی آن به توضیح مدار می‌پردازیم:

```
wire [3:0] multiplier_first_operand;
wire [3:0] multiplier_second_operand;
wire [3:0] multiplier_output;
reg [1:0] stage2_counter;
reg stage2_ready;
reg [3:0] stage2_first, stage2_second, stage2_third, stage2_forth;
reg [1:0] stage2_opcode;
reg [3:0] first, second, third;
multiplier multiplier_unit(multiplier_first_operand,
 multiplier_second_operand, multiplier_output);
assign multiplier_first_operand = (stage2_counter == 0) ?
 instruction_in_register[15:12]: (stage2_counter == 1) ?
 instruction_in_register[11:8]: (stage2_counter == 2) ?
 instruction_in_register[11:8]: instruction_in_register[15:12];
assign multiplier_second_operand = (stage2_counter == 0) ?
 instruction_in_register[7:4]: (stage2_counter == 1) ?
 instruction_in_register[7:4]: (stage2_counter == 2) ?
 instruction_in_register[3:0]: instruction_in_register[3:0];
assign wait_mult = !stage2_ready;
always @(posedge clk)
begin
  if (reset || !wait_all)
  begin
    stage2_counter <= 0;
    stage2_ready <= 0;
  end
  if (stage2_ready == 0)
  begin
    if (instruction_in_register[17] == 0) // it is add or sub
    begin
      if (!wait_add)
```

```
        begin
          stage2_opcode <= instruction_in_register[17:16];
          stage2_first <= instruction_in_register[15:12];
          stage2_second <= instruction_in_register[11:8];
          stage2_third <= instruction_in_register[7:4];
          stage2_forth <= instruction_in_register[3:0];
          stage2_counter <= 0;
          stage2_ready <= 1;
        end
      end
      else // it is mult
      begin
        if (stage2_counter == 0)
        begin
          first <= multiplier_output;
          stage2_counter <= stage2_counter + 1'b1;
        end
        if (stage2_counter == 1)
        begin
          second <= multiplier_output;
          stage2_counter <= stage2_counter + 1'b1;
        end
        if (stage2_counter == 2)
        begin
          third <= multiplier_output;
          stage2_counter <= stage2_counter + 1'b1;
        end
        if (stage2_counter == 3)
        begin
          stage2_first <= first;
          stage2_second <= second;
          stage2_third <= third;
          stage2_forth <= multiplier_output;
          stage2_opcode <= instruction_in_register[17:16];
          stage2_counter <= stage2_counter + 1'b1;
          stage2_ready <= 1;
        end
      end
    end
  end
```

در این بخش همانطور که قبل‌تر گفته شد، برای محاسبه ضرب، به ۴ مقدار ac و ad و bc و bd نیاز داریم که باید تک به تک آن‌ها را محاسبه کنیم. در نتیجه می‌توان فهمید که خود بخش ضرب، ۴ کلاک طول می‌کشد چون تنها یک ضرب کننده داریم. در این بخش، در هر کلاک باید بدانیم که در حال حاضر، کدام یک از آن ۴ ضرب گفته شده را باید انجام دهیم. پس یک متغیر دو

بیتی در نظر می‌گیریم تا بر حسب آن بتوانیم تشخیص دهیم که کدام ضرب را الان باید انجام دهیم. یک رجیستر هم داریم که بر اساس آن بتوانیم مشخص کنیم که نتیجه ضرب آماده است یا نه که این رجیستر لازم است یک بیتی باشد و به شکل فلگ عمل کند. از این رجیستر برای wait-all هم استفاده می‌شود و همانطور که در بخش اول این ماژول آورده شد، در هر لحظه باید بدانیم که کار قسمت ضرب و جمع، انجام شده است یا خیر که با کمک این رجیستر می‌توانیم بفهمیم که کارش انجام شده است یا نه. همانطور که مشخص است، وقتی عملیات مدنظر، جمع باشد، هیچ نیازی به ضرب نیست ولی اگر عملیات خواسته شده، ضرب باشد، باز هم به جمع و تفریق نیاز داریم چون بعد از محاسبه آن ۴ ضرب که گفتیم، نیاز می‌شود که جمع یا تفریقی انجام دهیم. پس خروجی‌های هر کدام از این ۴ ضرب را باید در جایی نگه داریم و به stage بعدی منتقل کنیم تا عملیات جمع یا تفریق روی آن‌ها انجام شود. رجیسترهای stage۲-first و stage۲-second و stage۲-third و stage۲-forth برای همین کار هستند. در واقع این ۴ رجیستر هستد که در بخش جمع و تفریق، روی آن‌ها عملیات انجام می‌شود. قبل‌تر هم گفته شد که حتی اگر عملیات جمع باشد، از این stage باید بگذریم و ۴ رجیستر stage۲-first تا stage۲-forth هستند که به stage بعدی فرستاده می‌شوند. پس اگر عملیات، جمع یا تفریق باشند، خود a و b و c و d به ترتیب در stage۲-first تا stage۲-forth قرار می‌گیرند ولی اگر عملیات وارد شده، ضرب باشد، به ترتیب ac و bc و bd و ad در این رجیسترها قرار می‌گیرند چون در نهایت باید مقدارهای ac-bd و bc+ad محاسبه شوند پس جوری در این ۴ رجیستر قرار می‌دهیم که در بخش جمع به مشکل بر نخوریم و بدون در نظر گرفتن نوع دستور که جمع یا ضرب است، رجیسترهای واحدی را از هم کم یا به هم اضافه کنیم. البته در هر صورت stage جمع یا تفریق باید بداند opcode چیست چون بر حسب آن باید تصمیم بگیرد که دو جمع یا دو تفریق یا یک جمع و یک تفریق انجام دهد. پس opcode را هم در این بخش در رجیستری ذخیره می‌کنیم و آن را به stage بعدی می‌دهیم تا از آن استفاده کند. در واقع ما نتیجه هر ضرب را در رجیستر جداگانه‌ای می‌گذاریم و در آخر کار که قرار است از این stage خارج شویم، همه را به رجیسترهای نهایی منتقل می‌کنیم تا به stage بعدی بروند. در واقع نیاز است که حاصل ۳ ضرب قبلی را داشته باشیم و به محض پایان یافتن ضرب چهارم، هر ۴ تا را به رجیسترهایی منتقل کنیم که قرار است به Stage بعدی بروند. چون که داریم درون pipeline عملیات‌ها را انجام می‌دهیم، خیلی باید حواسمان باشد که روی رجیستری که stage دیگری دارد با آن کار می‌کند یک دفعه تغییری ایجاد نکنیم و برای همین است که در واحد ضرب، در پایان کار نتایج را منتقل می‌کنیم به بخش دیگر چون اگر بعد پایان هر یک از این ۴ ضرب و به شکل مستقل اینکار را انجام می‌دادیم، ممکن بود که واحد جمع با مقادیر قبلی داخل آن رجیسترها هنوز کار داشته باشد و کلا کار خراب شود. پس یک سری رجیستر داریم که نتایج ضرب را به شکل موقت نگه می‌دارند و واحد جمع با آن‌ها کاری ندارد و یک سری رجیستر هم داریم که رجیسترهای اصلی هستند و واحد جمع از روی آن‌ها مقدارها را بر می‌دارد. آن رجیسترهای موقت، همان first تا third هستند و رجیسترهای اصلی هم قبلا معرفی شدند و stage۲-first تا stage۲-forth هستند.

در ادامه نمونه‌گیری از multiplier انجام شده و ورودی‌ها و خروجی‌ها به آن داده شده است که جلوتر مقدار هر کدام از این ورودی‌ها را مشخص می‌کنیم و می‌گوییم که خروجی کجا گذاشته شود.

نخست ورودی‌ها را مشخص می‌کنیم:

- اگر شمارنده ضرب در مرحله صفر بود، یعنی اولین بار بود که ضرب انجام می‌دادیم، ورودی اول می‌شود بخشی از instruction که a را نشان می‌دهد و ورودی دوم می‌شود بخشی از instruction که c را نشان می‌دهد تا در واقع ac در این مرحله ساخته شود.
- اگر شمارنده ضرب در مرحله اول بود، یعنی دومین بار بود که ضرب انجام می‌دادیم، ورودی اول می‌شود بخشی از instruction که b را نشان می‌دهد و ورودی دوم می‌شود بخشی از instruction که c را نشان می‌دهد تا در واقع bc در این مرحله ساخته شود.
- اگر شمارنده ضرب در مرحله دوم بود، یعنی سومین بار بود که ضرب انجام می‌دادیم، ورودی اول می‌شود بخشی از instruction که b را نشان می‌دهد و ورودی دوم می‌شود بخشی از instruction که d را نشان می‌دهد تا در واقع bd در این مرحله ساخته شود.
- اگر شمارنده ضرب در مرحله سوم بود، یعنی چهارمین بار بود که ضرب انجام می‌دادیم، ورودی اول می‌شود بخشی از instruction که a را نشان می‌دهد و ورودی دوم می‌شود بخشی از instruction که d را نشان می‌دهد تا در واقع ad در

این مرحله ساخته شود.

سپس در خط بعد مشخص کرده‌ایم که کار بخش ضرب شروع شده است و آن سیگنال wait-all یک می‌شود تا دستور جدیدی به این بخش نیاید.
حال به لبه بالارونده کلاک نگاه می‌کنیم. اگر reset یک بود یا آن wait-all صفر بود، یعنی اینکه هیچ ضرب یا جمعی در جریان نیست و می‌توانیم کار خودمان را شروع کنیم. پس هم ready را صفر می‌کنیم تا بقیه بخش‌ها بفهمند که این بخش ضرب مشغول است و شمارنده ضرب این بخش را هم صفر می‌کنیم و این شمارنده همان است که مشخص می‌کرد کدام ضرب را الان باید انجام دهیم و آن را صفر می‌کنیم تا از اول کارمان را شروع کنیم و به محاسبه ضرب‌ها بپردازیم. سپس چک می‌کنیم که اگر نتیجه این بخش آماده نشده بود، بر حسب اینکه opcode چیست، تصمیم می‌گیریم که چه کاری انجام دهیم. اگر دو بیت پر ارزش instruction ۰۰ یا ۰۱ بود، یعنی اینکه با عمل جمع یا تفریق مواجه هستیم. اگر بخش جمع مشغول بود که باید یک کلاک دیگر هم در این بخش باقی بمانیم ولی اگر بخش جمع بیکار می‌توانیم ورودی‌ها را آماده کنیم تا به stage بعدی برویم. پس اگر بخش بعدی کاری نداشت، اول از همه ورودی‌های a و b و c و d را به فرمتی که قبل‌تر گفته شد، به رجیسترهای مربوطه می‌دهیم. در گام بعدی، شمارنده ضرب را صفر می‌کنیم تا دفعه بعد که به stage ضرب آمدیم، به اشتباه نیفتیم و از همان صفر شروع به محاسبه کنیم.
همچنین خروجی ready مربوط به این بخش ضرب را هم یک می‌کنیم که یعنی کار این بخش تمام شده است و بخش fetch اگر بخواهد، می‌تواند ورودی بعدی را بدهد به این بخش.
حال اگر ۲ بیت پرارزش instruction ۱۰ بود، یعنی با عمل ضرب روبه‌رو هستیم. برای انجام اینکار، بر حسب شمارنده ضرب، مشخص می‌کنیم که در کدام مرحله هستیم و طبق توضیحات گذشته و همانطور که در کد مشخص است، تصمیم می‌گیریم که چه کاری انجام دهیم.

- اگر شمارنده صفر باشد، خروجی ضرب‌کننده که قبل‌تر ساختیم را داخل first می‌ریزیم و شمارنده را هم یکی بالا می‌بریم.
- اگر شمارنده یک باشد، خروجی ضرب‌کننده که قبل‌تر ساختیم را داخل second می‌ریزیم و شمارنده را هم یکی بالا می‌بریم.
- اگر شمارنده دو باشد، خروجی ضرب‌کننده که قبل‌تر ساختیم را داخل third می‌ریزیم و شمارنده را هم یکی بالا می‌بریم.
- اگر شمارنده سه باشد، آن مقادیر ۳ رجیستر موقت را به رجستر اصلی منتقل می‌کنیم و خروجی نمونه multiplier را هم در stage۲-forth قرار می‌دهیم تا هر ۴ خروجی آماده باشد. شمارنده را هم مثل ۳ حالت قبل، یکی اضافه می‌کنیم و در این شرایط، شمارنده از ۱۱ تبدیل به ۱۰۰ می‌شود و چون رجیستر دو بیتی است، تبدیل به همان ۰۰ می‌شود. در نهایت هم چون کار بخش ضرب‌کننده تمام شده است، ready مربوط به این بخش را یک می‌کنیم تا به stage بعدی که بخش جمع و تفریق است، برویم.

حال به بخش سوم و نهایی می‌رسیم که بخش جمع و تفریق است. در این بخش هم نخست کد را قرار می‌دهیم و بعد از روی آن شروع به توضیح دادن می‌کنیم:

```
wire [3:0] adder_first_operand;
wire [3:0] adder_second_operand;
wire add_diff_not;
wire [3:0] adder_out;
reg stage3_counter;
reg stage3_ready;
reg [3:0] stage3_first;
adder adder_unit(add_diff_not, adder_first_operand,
 adder_second_operand, adder_out);
```

```
assign adder_first_operand = (stage3_counter == 0) ? stage2_first :
 stage2_second;
assign adder_second_operand = (stage3_counter == 0) ? stage2_third :
 stage2_forth;
assign add_diff_not = stage2_opcode[0] || stage2_opcode[1] &&
 stage3_counter;
assign wait_add = !stage3_ready && (stage3_counter != 1);
assign ready = stage3_ready;
always @(posedge clk)
begin
  if (reset || !wait_all)
  begin
    stage3_counter <= 0;
    stage3_ready <= 0;
  end
  if (stage3_ready == 0)
  begin
    if (stage3_counter == 0)
    begin
      stage3_first <= adder_out;
      stage3_counter <= stage3_counter + 1'b1;
    end
    if (stage3_counter == 1)
    begin
      real_part <= stage3_first;
      imaginary_part <= adder_out;
      stage3_counter <= stage3_counter + 1'b1;
      stage3_ready <= 1;
    end
  end
end
```

در این بخش دو حالت انجام کار داریم، یا باید ۲ تا جمع انجام دهیم یا دو تا تفریق و یا یک جمع و یک تفریق باید انجام دهیم که در همه‌ی این حالت‌ها، دو مورد کلی وجود دارد. پس همانطور که در بخش ضرب یک شمارنده ۲ بیتی داشتیم که مرحله ضرب را مشخص کند، در این بخش هم به متغیری نیاز داریم تا مرحله جمع و یا تفریق را مشخص کند که چون در این stage کلا دو حالت داریم، به یک شمارنده ۱ بیتی نیاز داریم. در این بخش هم رجیستر ready داریم که مشخص می‌کند کار این بخش انجام شده است یا نه هنوز و از این رجیستر در آینده برای ساخت wait-all استفاده می‌شود تا با آن، بخش fetch بفهمد که باید دستور بعدی را به stage های بعدی بدهد یا ندهد. همانطور که در بخش قبل هم دیدیم، چون در pipeline قرار داریم، نباید رجیسترها را خیلی راحت عوض کنیم چون ممکن است در جایی دیگر مورد استفاده باشد. پس همانطور که قبل‌تر برای نگه داشتن ضرب‌های قبلی، متغیرهای موقت ساختیم، در این بخش هم متغیر موقت می‌سازیم. فقط در این بخش چون کلا ۲ مرحله وجود دارد، تنها به یک متغیر موقت نیاز داریم تا حاصل جمع یا تفریق اول را در آن بریزیم. و به این رجیستر نیاز داریم تا زمانی که داریم جمع یا تفریق دوم را انجام می‌دهیم حاصل جمع یا تفریق اول، از دست نرود. سپس به سراغ نمونه‌گیری از جمع‌کننده می‌رویم و ۲ ورودی آن و ۱ خروجی آن و فلگ مربوط به عمل جمع یا تفریق را هم به آن می‌دهیم و در ادامه، مقدار آن‌ها را مشخص می‌کنیم.

حال به مشخص کردن ورودی‌های واحد جمع‌کننده می‌پردازیم:

- اگر در مرحله اول جمع باشیم، عملوندها a و c هستند. البته در بخش قبل هم گفته شد که لزوما خود a و c نیستند و ممکن است مقادیری باشند که در بخش ضرب ساخته شده‌اند ولی با ترتیبی در رجیسترهای نهایی قرار گرفته‌اند که در این بخش لازم نیست این موضوع را چک کنیم و در بخش قبل کامل این موضوع توضیح داده شد. اگر هم در بخش دوم جمع باشیم، عملوندها b و d هستند و باز هم مثل قسمت قبل، لزوما خود b و d نیستند و ممکن است حاصلی باشند که از بخش ضرب ساخته شده است.
- ورودی فلگ که مشخص کننده جمع یا تفریق هست هم به این شکل مشخص می‌شود که یا باید بیت کم‌ارزش opcode یک باشد که یعنی جمع داریم و یا باید بیت پرارزش opcode یک باشد که یعنی دستور ضرب است و در این بخش باز هم جمع داریم اگر در بخش دوم جمع باشیم. در دو مورد بالا، این فلگ باید یک باشد که یعنی جمع داریم و در غیر این صورت این فلگ صفر است که یعنی تفریق داریم.

خط بعدی هم که همان بخشی است که wait-all بر اثر آن تغییر پیدا می‌کند و مشخص می‌کند که بخش آخر کارش تمام شده است یا نه و چون این بخش، بخش آخر است، با تمام شدن این stage عملا اجرای یک دستور پایان یافته است. در ادامه‌ی کار به لبه بالارونده کلاک توجه می‌کنیم.

- اگر reset یک بود یا wait-all صفر بود، یعنی اینکه می‌توانیم کارکان را شروع کنیم و برای شروع کار، شمارنده این بخش و همچنین سیگنال ready این بخش را صفر می‌کنیم تا از اول کار را شروع کنیم.
- در ادامه چک می‌کنیم که اگر سیگنال ready این بخش صفر بود، یعنی در اول کار هستیم و بعد از دیدن این موضوع، بر اساس شمارنده، حالت‌بندی می‌کنیم. اگر شمارنده، صفر بود، یعنی در بخش اول هستیم و برای اینکه خروجی این بخش را از دست ندهیم، خروجی نمونه adder را به stage۳-first می‌ریزیم و بعد از آن، شمارنده را یکی زیاد می‌کنیم تا به حالت دوم برویم. اگر هم شمارنده در حالت دوم باشد، یعنی خروجی این instruction آماده است و خروجی مرحله قبل را به عنوان بخش حقیقی ALU بر می‌گردانیم و خروجی این حالت دوم را به عنوان بخش موهومی ALU بر می‌گردانیم. در نهایت هم شمارنده را یکی زیاد می‌کنیم تا از ۱ به ۱۰ برود و چون رجیستر مربوطه یک بیتی است، فقط صفر آن می‌ماند و به حالت اولیه برای instruction بعدی بر می‌گردیم. به عنوان آخرین کار هم متغیر ready مربوط به این بخش را یک می‌کنیم تا بخش‌های دیگر بدانند که کار این stage هم تمام شده است.

ماژول Test Bench

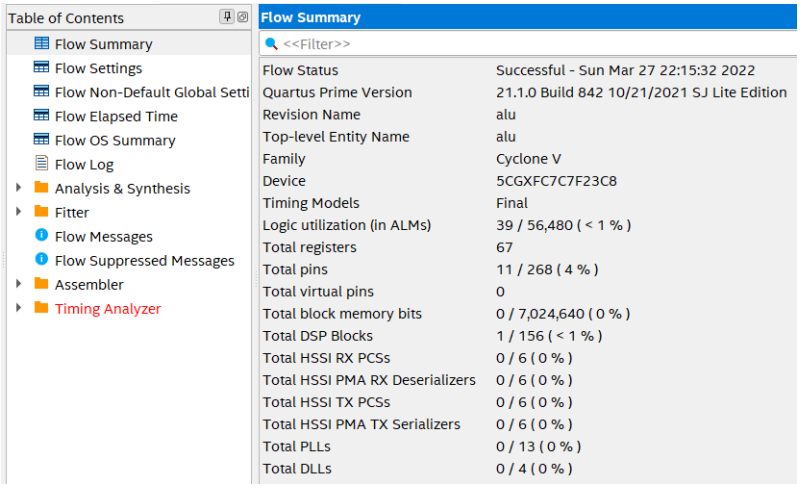
حال به بخش نوشتن تست‌بنچ می‌رسیم که کد آن به شکل زیر است:

```
module alu_tb;
  reg reset, clk;
  wire [3:0] imaginary_part;
  wire [3:0] real_part;
  wire ready;
  alu cmp(reset, clk, real_part, imaginary_part, ready);
  initial clk = 1;
  always #5 clk = ~clk;
  initial
```

```
10    begin
11      reset = 1;
12      #5;
13      reset = 0;
14    end
15  endmodule
```

در این بخش هم که نخست ورودی‌های ماژول alu را می‌سازیم یعنی همان reset و clk و ۲ رجیستر ۴ بیتی real-part و imaginary-part و کار نمونه‌گیری را انجام می‌دهیم و ورودی‌های لازم را به آن می‌دهیم. سپس طبق روش همیشگی clock را می‌سازیم که اول آن را صفر یا یک قرار می‌دهیم داخل یک بلاک initial و سپس در یک بلاک always هر ۵ نانوثانیه یک بار آن را تغییر می‌دهیم. برای به کار افتادن مدار، نیاز است که reset اول یک شود و بعد از آن صفر شود تا مدار شروع به کار کند. مانند عکس‌های زیر می‌بینیم که کد با موفقیت، کامپایل و سنتز می‌شود:

شکل ۴: کامپایل موفقیت‌آمیز

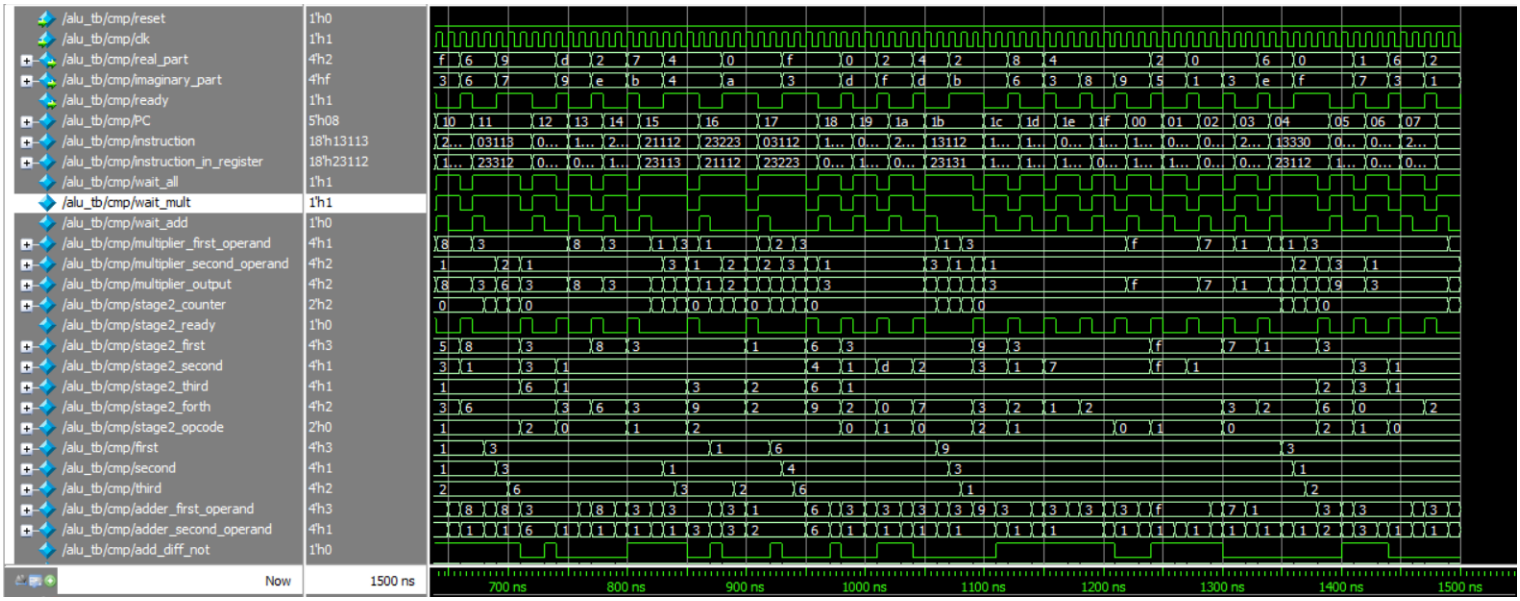


شکل ۵: کامپایل موفقیت‌آمیز

```
# Compile of adder.v was successful.
# Compile of multiplier.v was successful.
# Compile of memory.v was successful.
# Compile of alu.v was successful.
# Compile of alu_tb.v was successful.
ModelSim> iles, 0 failed with no errors.
```

شکل ۶: کامپایل موفقیت‌آمیز

حال تست‌بنچ را اجرا می‌کنیم:



شکل ۷: اجرای تست‌بنچ

و همانطور که مشاهده می‌شود، خروجی‌ها بر حسب چیزی که در memory بود، به درستی محاسبه شدند. البته خروجی تست‌بنچ بسیار بزرگ است و هم افقی و هم عمودی، دارای scroll است که در عکس نمی‌شد آن‌ها را آورد ولی فایل‌های پروژه به پیوست است و می‌توانید آن‌ها را اجرا کنید. البته دقت کنید که به خاطر بحث پایپلاین، نتیجه، چند کلاک بعدتر مشخص می‌شود و متغیر real-part و imaginary-part دقیقا همان دستور زیر خودشان را نشان نمی‌دهند و جدود ۲ یا ۳ دستور قبل‌تر از خود را نشان می‌دهند.

نتیجه‌گیری

پس در این آزمایش توانستیم یک ALU برای اعداد مختلط درست کنیم که از قابلیت pipeline استفاده می‌کند. یعنی در بخش ALU ۳ کار fetch کردن دستور و جمع و ضرب به شکل موازی انجام می‌دهد و در این مسیر، تنها از یک واحد جمع‌کننده و صرب‌کننده استفاده می‌کند. همچنین این آزمایش دارای بخش memory هم بود که اطلاعات و دستورها در این بخش قرار داشت و هر دستور شامل ۳ بخش بود. اول opcode مشخص می‌شد و بعد از آن ورودی اول و سپس، ورودی دوم مشخص می‌شد. این آزمایش با کمک زبان توصیف سخت‌افزار verilog انجام شد و testbench مربوطه هم برای این آزمایش نوشته شد.