**خدا هست**

توضیحات تمرین 3 – DSD – دکتر فصحتی

امیرمحمد شربتی 402106112

1404/4/14

* در ابتدا بنده از ماژول جمع کننده پیاده سازی رو شروع کردم:

بنده ابتدا یک مدار تماما ترکیبی بدون کلاک و بدون FSM پیاده سازی کردم. در واقع یک مدار تک چرخه ای combinational ساده:

* ماژول بعدی که طراحی کردم، ضرب کننده بود:

ضرب کننده از دو بخش اصلی تشکیل شده است. یک بخش که ضرب دو عدد 8 بیتی است که به کمک الگوریتم shift and add پیاده سازی شده است و بخش دیگر استفاده از الگوریتم Karatsuba است که برای محاسبه ضرب دو عدد 16 بیتی استفاده میشود.

برای پیاده سازی بخش shift and add برای ضرب 8 بیتی، می‌توان این روش که مشابه ماژول پیاده سازی شده است را هم استفاده کرد. (خیلی مشابه با اندکی تفاوت: این خلاصه تر است)

module mult\_8bit\_shift\_add (input [7:0] A, B, output [15:0] result);

reg [15:0] product;

integer i;

always @(\*) begin

product = 16'b0;

for (i = 0; i < 8; i = i + 1)

if (B[i]) product = product + (A << i);

end

assign P = product;

endmodule

* یک سری نکات در مورد پیاده سازی register file :

اول اینکه برخلاف ماژول های ضرب و تقسیم ازstart و doneاستفاده نشد. چون در این واحد های محاسباتی عملیات ها چند چرخه طول می‌کشند. پس این ماژول ها نیاز به ماشین حالت متناهی (FSM) کنترل داخلی دارند. اما در رجیستر فایل، خواندن و نوشتن فقط یک چرخه طول می‌کشند، پس هیچ نیازی به start و done نیست.

در مورد reset اما، می‌توانست استفاده نشود. معمولا reset برای این استفاده می‌شود تا internal FSM را پاک کند و حالت آن را به حالت اول برگرداند. اما بنده این را هم اضافه کردم. اگر reset فعال شود، تمام ثبات ها صفر می‌شوند.

یک نکته مهم که در ماژول های قبلی هم وجود داشت مربوط به سنتز پذیری کد در قسمت if-else ها است. اینکه ما else نداریم، برای سنتز مشکلی ایجاد نمی‌کند؟ پاسخ این است که خیر. چون در مدارات ترتیبی (always @(posedge clk)) فلیپ فلاپ ها به صورت اتوماتیک داده و دیتای قبلی خود را نگه می‌دارند. اما در مدارات ترکیبی (always @(\*))، نبود else مشکل ایجاد میکند چون ناخواسته یک حافظه و latch ایجاد میکند که زمان بندی و درستی مدار را زیر سوال میبرد.

* قسمت بعدی، حافظه پردازنده است.

می‌توان برای خواندن و نوشتن از دو always مجزا بدین صورت استفاده کرد. ولی ترکیب کردن آنها در یک بلوک هم مانعی ندارد. یعنی حالت دیگر این است:

// READ

always @(posedge clk) begin

if (read\_enable( read\_data <= mem[addr];

end

// WRITE

always @(posedge clk) begin

if (write\_enable) mem[addr] <= write\_data;

end

نکته دیگر در مورد این memory این است که برخلاف ماژول های قبلی از reset استفاده نکردم. (به نظرم ریست کردن کل حافظه، این هم حافظه ای به این بزرگی بی معنی است. ولی خب، اضافه کردن آن کاری ندارد. مشابه register file می‌توان این کار را انجام داد)

چالش دیگر در طراحی حافظه که در رجیستر فایل هم وجود دارد، تعریف ثبات ها یا حافظه به صورت signed یا unsigned است. بنده در اینجا چون **دستورات** را هم در حافظه ذخیره می‌کنیم، mem را unsigned فرض کردم. اگر بخواهیم برای اعداد علامت دار استفاده کنیم، داده خروجی از حافظه را به صورت signed تعریف می‌کنیم تا عملیات مطابق انتظار پیش رود. اما چون فقط برای ذخیره دیتا از رجیستر فایل استفاده میکنیم، این را به صورت signed تعریف کردم. (تا اونجایی که تحقیق کردم، کلا چیز مهمی نیست، مهم تفسیر ما از این دیتای ذخیره شده در حافظه یا ثبات است که به صورت علامت دار یا بی علامت به آن نگاه کنیم. تفسیر ما هم چندان ربطی به تعریف حافظه و رجیستر فایل ندارد، بلکه در لحظه **استفاده** مهم است)

* قبل از پیاده سازی آخرین ماژول که در واقع کنترل کننده و مسئول اتصال بخش های مختلف است، بهتر است ماژول ALU هم پیاده سازی شود تا دستورات محاسباتی را بهتر مدیدریت کنیم:
* پس از پیاده سازی sub-module ها حالا نوبت به سرهم کردن آنها و طراحی top module ها هست که مسئولیت کنترل دستورات و دیکود و اجرای آنها را دارند: