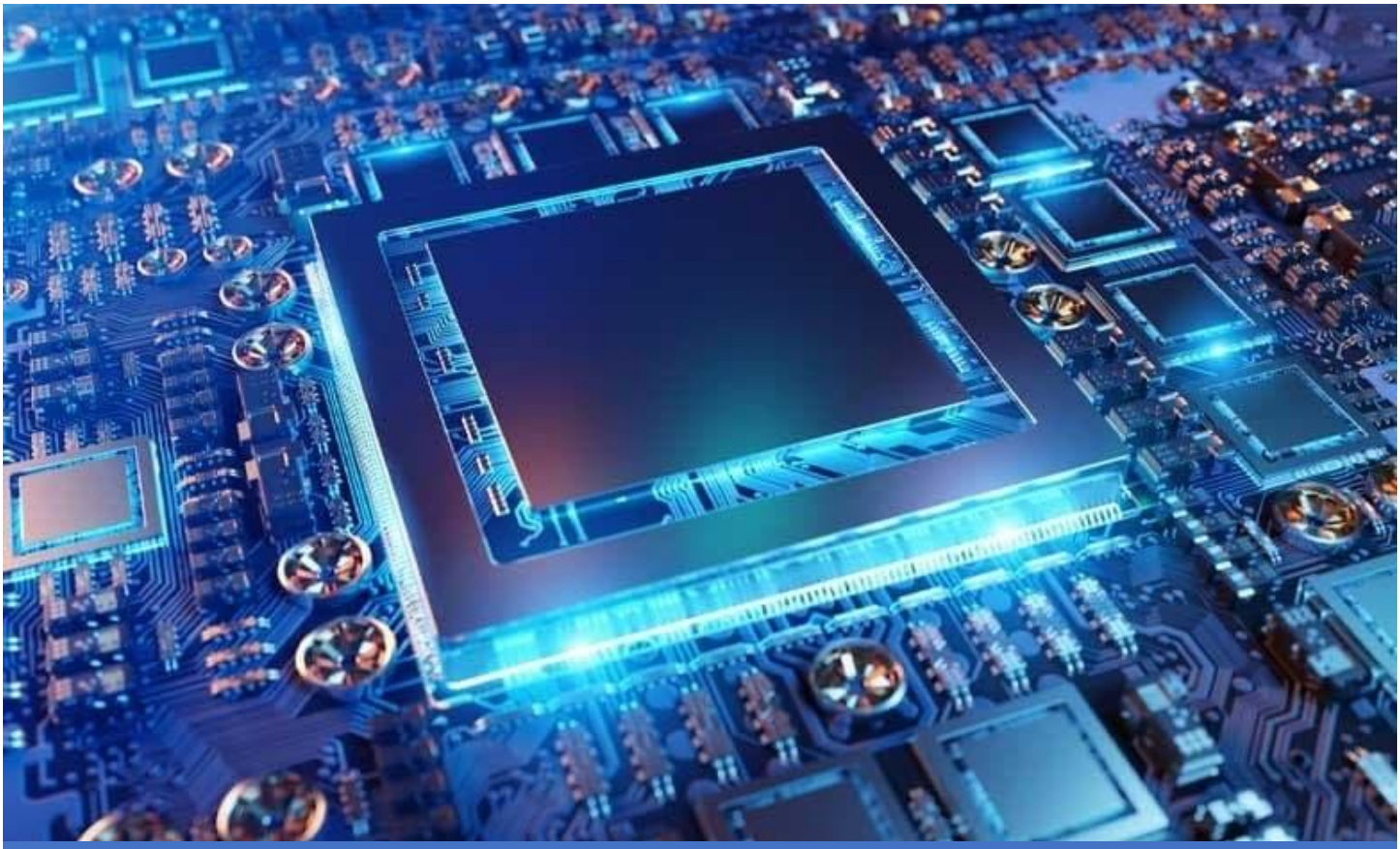
به نام خدا

عرفان تیموری

شماره دانشجویی : 402105813



exercise 3

Erfan Teymouri 7/27/25 Digital System and Design

این پروژه با هدف طراحی و پیاده‌سازی یک پردازنده ساده اما کاربردی انجام شده است. در این پردازنده، اجزای اصلی مانند واحد محاسبات و منطق (ALU)، رجیستر فایل، حافظه اصلی (Main Memory) و واحد کنترل (Control Unit) به‌صورت ماژولار طراحی شده‌اند تا هم ساختار پروژه قابل فهم و توسعه‌پذیر باشد و هم بتوان هر بخش را جداگانه تست و عیب‌یابی کرد.

یکی از اهداف اصلی پروژه، آشنایی با نحوه طراحی اجزای مختلف یک پردازنده از سطح گیت تا سطح سیستم است. برای مثال، عملیات‌های جمع و تفریق در ALU به کمک الگوریتم Carry Select Adderدر سطح گیت پیاده‌سازی شده‌اند و برای ضرب و تقسیم نیز از الگوریتم‌های Karatsuba و Restoring Division استفاده شده است.

این پردازنده از معماری word-addressable با کلمات ۱۶ بیتی پشتیبانی می‌کند و شامل چهار رجیستر عمومی (x0 تا x3) است که داده‌های میانی را ذخیره می‌کنند. دستورات پردازنده شامل عملیات محاسباتی (جمع، تفریق، ضرب، تقسیم)، دستورات حافظه (load, store) و کنترل جریان برنامه می‌باشند. تمامی این دستورات توسط واحد کنترلی مدیریت و دیکود شده و سیگنال‌های کنترلی لازم برای رجیستر فایل، حافظه و ALU تولید می‌شوند.

پروژه به‌صورت ماژول به ماژول توسعه یافته است؛ یعنی ابتدا اجزای پایه مانند رجیستر، رجیستر فایل و حافظه طراحی شده‌اند، سپس ALU و واحد کنترل به‌طور مستقل توسعه یافته‌اند و در نهایت همه این بخش‌ها در یک ماژول نهایی (Top Module) به هم متصل شده‌اند. این ساختار ماژولار، نه تنها کدنویسی و تست هر قسمت را ساده‌تر کرده، بلکه امکان گسترش یا تغییر معماری پردازنده را در آینده فراهم می‌کند.

با این توضیحات به سراغ نحوه پیاده‌سازی ماژول‌ها رفته و خلاصه‌ای از آن را بازگو می‌کنیم:

1. رجیستر و رجیسترفایل:

ماژول Register و RegisterFile در این پروژه به‌عنوان هسته‌ی ذخیره‌سازی داده‌ها طراحی شده‌اند و وظیفه‌ی آن‌ها مدیریت مقادیر میانی بین اجرای دستورات مختلف پردازنده است.

1.1. ماژول Register:

ماژول Register یک رجیستر ۱۶ بیتی ساده است که قابلیت نوشتن در لبه بالارونده کلاک را دارد. این ماژول همچنین دارای سیگنال reset است که مقدار رجیستر را به صفر بازنشانی می‌کند. علاوه بر این، یک سیگنال valid در آن وجود دارد که مشخص می‌کند آیا رجیستر داده‌ی معتبری ذخیره کرده یا خیر.

ویژگی‌های مهم:

Write Enable: فقط زمانی که write_enable فعال باشد، داده‌ی جدید در رجیستر ذخیره می‌شود.

Reset: با فعال شدن reset، رجیستر صفر می‌شود و داده ذخیره شده در آن دیگر معتبر نخواهد بود.

Valid: این فلگ برای جلوگیری از خواندن داده‌های نامعتبر (مثلاً قبل از اولین نوشتن) استفاده می‌شود.

1.2. ماژول RegisterFile:

RegisterFile مجموعه‌ای از ۴ رجیستر است که به کمک چیزی شبیه یک دیکودر ساده، هر بار یکی از آن‌ها را برای نوشتن انتخاب می‌کند. این ماژول دارای دو پورت خواندن و یک پورت نوشتن است. خواندن داده‌ها در لبه پایین‌رونده کلاک انجام می‌شود تا تداخل بین عملیات خواندن و نوشتن در یک چرخه‌ی کلاک کاهش یابد.

نکات مهم:

سیستم دیکودر در نوشتن: با بررسی write_addr و فعال بودن write_enable، فقط یک رجیستر مشخص داده جدید را دریافت می‌کند.

خواندن مشروط: اگر رجیستر معتبر نباشد (پرچم valid آن صفر باشد)، خروجی خواندن صفر می‌شود تا از داده‌های غیرقابل اعتماد جلوگیری شود.

ماژولار بودن: هر رجیستر از ماژول Register استفاده می‌کند؛ بنابراین کد ساختارپذیر و قابل توسعه باقی مانده است

```
// Instantiate 4 registers
Register reg0 (.clk(clk), .reset(reset), .write_enable(reg_write_en[0]), .data_in(write_data), .data_out(reg_out[0]), .valid(reg_valid[0]));
Register reg1 (.clk(clk), .reset(reset), .write_enable(reg_write_en[1]), .data_in(write_data), .data_out(reg_out[1]), .valid(reg_valid[1]));
Register reg2 (.clk(clk), .reset(reset), .write_enable(reg_write_en[2]), .data_in(write_data), .data_out(reg_out[2]), .valid(reg_valid[2]));
Register reg3 (.clk(clk), .reset(reset), .write_enable(reg_write_en[3]), .data_in(write_data), .data_out(reg_out[3]), .valid(reg_valid[3]));
```

2. Main Memory:

ماژول Main Memory به‌عنوان حافظه اصلی پردازنده طراحی شده است و وظیفه ذخیره‌سازی و واکشی داده‌ها و دستورات را برعهده دارد. این حافظه Word-addressable بوده و از 65536 خانه 16 بیتی (معادل 2 بایت) تشکیل شده است.

```
// Combinational read
always @(*) begin
    if (read_enable)
        data_out = memory_array[addr];
    else
        data_out = 16'b0;
end

// Synchronous write
always @(posedge clk) begin
    if (write_enable) begin
        memory_array[addr] <= write_data;
    end
end
```

نکات مهم و نحوه پیاده‌سازی:

1. ساختار حافظه: یک آرایه memory_array با طول 65536 و پهنای 16 بیت تعریف شده است که هم داده‌ها و هم دستورات در آن نگهداری می‌شوند.

2. خواندن ترکیبی (Combinational Read): وقتی سیگنال read_enable فعال باشد، مقدار حافظه در آدرس

addr به‌صورت بی‌درنگ روی خروجی data_out قرار می‌گیرد. اگر این سیگنال غیر فعال باشد، خروجی برابر صفر است.

3. نوشتن همزمان با کلاک (Synchronous Write):

نوشتن در حافظه تنها در لبه بالارونده کلاک و زمانی که write_enable فعال باشد انجام می‌شود.

4. پشتیبانی از داده‌های signed:

داده‌ها به‌صورت signed تعریف شده‌اند تا از عملیات صحیح در ALU و دستوراتی مانند ضرب و تقسیم پشتیبانی شود.

3. ALU:

ماژول ALU واحد محاسباتی و منطقی پردازنده است که عملیات اصلی جمع، تفریق، ضرب و تقسیم را اجرا می‌کند. این ماژول با دریافت دو عملوند a و b و یک کد عملیات (opcode) وظیفه دارد نتیجه محاسبه را در خروجی result قرار دهد و پس از اتمام عملیات، سیگنال done را فعال کند.

نکات مهم و نحوه پیاده‌سازی:

1. ساختار کلی:

ورودی opcode تعیین می‌کند که ALU چه عملیاتی انجام دهد؛ 000 برای جمع، 001 برای تفریق، 010 برای ضرب و 011 برای تقسیم است. همچنین سیگنال start آغاز محاسبه را مشخص می‌کند و done پایان آن را اعلام می‌کند.

2. جمع و تفریق با یک مدار واحد:

برای جمع و تفریق از Carry Select Adder استفاده شده است. به طوری که اگر opcode = 001 (تفریق) باشد، مقدار b به مکمل دو تبدیل شده (b_negated = ~b + 1) و به جمع‌کننده داده می‌شود؛ در غیر این‌صورت همان مقدار b وارد جمع‌کننده می‌شود.

3. ضرب و تقسیم:

ضرب با استفاده از ماژول KaratsubaMultiplier16_Parallel انجام می‌شود و تقسیم نیز توسط ماژول RestoringDivider16 انجام می‌گیرد. هر دو ماژول به‌صورت چند کلاکه عمل می‌کنند و پس از پایان، سیگنال‌های mul_done یا div_done را فعال می‌کنند.

4. کنترل خروجی:

در عملیات جمع و تفریق، نتیجه در همان کلاک start آماده است و done بلافاصله یک می‌شود.

در ضرب و تقسیم، خروجی پس از پایان عملیات (با بررسی mul_done یا div_done) در result ثبت و done فعال می‌شود.

حال که ساختار کلی ALU را بیان کردیم به سراغ نحوه پیاده‌سازی تک‌تک الگوریتم‌های جمع، ضرب و تقسیم می‌رویم:

3.1. ماژول CarrySelectAdder16:

ماژول CarrySelectAdder16 یک جمع‌کننده ۱۶ بیتی با استفاده از معماری Carry Select است که برای افزایش سرعت جمع نسبت به روش‌های ساده‌تر (مثل Ripple Carry) طراحی شده است.

نکات مهم و نحوه پیاده‌سازی:

1. تقسیم‌بندی به بلوک‌های ۴ بیتی:

جمع ۱۶ بیتی به ۴ بلوک ۴ بیتی تقسیم شده است. هر بلوک ۴ بیتی با استفاده از دو RippleCarryAdder4 محاسبه می‌شود؛ یکی با فرض cin = 0 و دیگری با فرض cin = 1.

2. انتخاب نتیجه با MUX:

بعد از مشخص شدن carry واقعی بلوک قبلی، با یک مالتی‌پلکسر بین نتایج دو جمع‌کننده‌ی هر بلوک (sum0 و sum1) تصمیم گرفته می‌شود؛ همچنین همین منطق برای انتخاب carry خروجی (cout0 و cout1) هم اعمال می‌شود.

3. RippleCarryAdder4 و FullAdder:

هر بلوک ۴ بیتی (در واقع هر RippleCarryAdder 4 بیتی) از چهار Full Adder پشت سر هم تشکیل شده که ساختار استانداردی دارند و نقش آن‌ها صرفاً جمع ساده و تولید carry است.

3.2. ماژول KaratsubaMultiplier16_Parallel:

ماژول KaratsubaMultiplier16_Parallel یک ضرب‌کننده‌ی ۱۶ بیتی را بر اساس الگوریتم Karatsuba پیاده‌سازی می‌کند که به‌صورت موازی اجرا می‌شود تا سرعت ضرب افزایش یابد.

نکات مهم و نحوه پیاده‌سازی:

1. ایده اصلی الگوریتم Karatsuba:

اعداد ۱۶ بیتی a, b به دو نیمه‌ی ۸ بیتی تقسیم می‌شوند:

$$a = a\_high \,||\, a\_low, b = b\_high \,||\, b\_low$$

سپس سه ضرب ۸×۸ انجام می‌شود:

$$z0 = a\_low \times b\_low, z2 = a\_high \times b\_high$$

$$z1 = (a\_low + a\_high) \times (b\_low + b\_high) - z0 - z2$$

2. ضرب‌های موازی با ShiftAddMultiplier8:

سه ماژول مجزای ShiftAddMultiplier8 به‌صورت موازی اجرا می‌شوند و نتیجه‌ی سه ضرب بالا را تولید می‌کنند. این سه ماژول الگوریتم ضرب عادی (Shift and Add) را انجام می‌دهند. این کار باعث کاهش زمان کل ضرب نسبت به ضرب‌های متوالی می‌شود.

3. تبدیل علامت به صورت نرم‌افزاری:

اگر یکی از ورودی‌ها منفی باشد، تبدیل به قدرمطلق خودش می‌شود؛ پس از محاسبه‌ی حاصل‌ضرب بدون علامت، در صورت نیاز، نتیجه با استفاده از مکمل ۲ به حالت علامت‌دار برگردانده می‌شود.

4. محدودیت به ۱۶ بیت:

اگرچه ضرب Karatsuba ذاتا می‌تواند تا ۳۲ بیت نتیجه تولید کند، این پیاده‌سازی نتیجه را تا ۱۶ بیت نگه می‌دارد و z2 (که مربوط به بیت‌های بالا است) عملاً نادیده گرفته می‌شود.

5. کنترل زمان اجرا:

با استفاده از شمارنده wait_cycles زمان لازم برای پایان یافتن هر سه ضرب ۸ بیتی تخمین زده می‌شود (۸ سیکل کلاک).

3.2. ماژول RestoringDivider16:

این ماژول پیاده‌سازی الگوریتم کلاسیک Restoring Division برای تقسیم دو عدد ۱۶ بیتی علامت‌دار است. خروجی فقط شامل خارج‌قسمت (quotient) است و باقی‌مانده (remainder) در کد نادیده گرفته شده است.

نکات مهم و نحوه پیاده‌سازی:

1. پیش‌پردازش ورودی‌ها:

ورودی‌های a (مقسوم) و b (مقسوم‌علیه) به قدر مطلق خودشان تبدیل می‌شوند تا عملیات تقسیم بدون علامت انجام شود.

سیگنال‌های sign_dividend و sign_divisor برای تشخیص علامت اولیه ذخیره می‌شوند.

ترکیب دو مقدار a و صفر در رجیستر ۳۲ بیتی A_Q نگهداری می‌شود؛ به طوری که 16 بیت بالا A (باقی‌مانده موقت) است و 16 بیت پایین Q (خارج‌قسمت در حال ساخت) است.

2. چرخه‌های تقسیم:

تقسیم در ۱۶ سیکل انجام می‌شود (یک سیکل برای هر بیت). در هر سیکل ابتدا A_Q یک بیت به چپ شیفت داده می‌شود، سپس M (مقسوم‌علیه) از بخش بالایی (A) کم می‌شود. اگر نتیجه منفی شود (بیت ۳۱، یعنی بیت علامت، برابر ۱ باشد)، عملیات بازگرداندن (restoring) انجام می‌شود (مقدار M دوباره به A اضافه می‌شود) و بیت خروجی Q[0] برابر صفر می‌شود؛ اما اگر نتیجه مثبت شود، بیت Q[0] برابر یک می‌شود.

3. تبدیل خروجی نهایی به خروجی علامت‌دار:

پس از اتمام چرخه‌ها، نتیجه خروجی (همان خارج‌قسمت) در A_Q[15:0] ذخیره شده است؛ البته اگر علامت‌های a و b متفاوت باشد، نتیجه نهایی به مکمل ۲ تبدیل می‌شود تا حاصل منفی شود.

سیگنال‌های کنترلی مهم (فلگ‌ها):

سیگنال / رجیستر	توضیح
start	شروع عملیات تقسیم
done	پایان عملیات و آماده بودن خروجی
busy	ماژول در حال انجام عملیات است
count	شمارنده‌ی مراحل تقسیم (تا ۱۶)
compute_result	فلگ فعال‌سازی مرحله‌ی نهایی محاسبه‌ی خروجی
sign_dividend, sign_divisor	ذخیره علامت اولیه‌ی a و b برای اصلاح خروجی
A_Q	رجیستر ترکیبی شامل A (باقی‌مانده) و Q (خارج‌قسمت)
M	قدر مطلق مقسوم‌علیه (divisor)

3. Control Unit:

ماژول controller بخش مرکزی و هدایت‌گر پروژه به شمار می‌رود که وظیفه مدیریت جریان اجرای دستورات را بر عهده دارد. این ماژول مطابق با یک ماشین حالت متناهی (FSM) طراحی شده و فرآیند اجرای دستورالعمل‌ها را در چند مرحله متوالی (Fetch، Decode، Execute، Memory، Write Back) پیاده‌سازی می‌کند.

نکات مهم و نحوه پیاده‌سازی:

در ابتدای هر چرخه، کنترلر با ورود به حالت FETCH_1 آدرس دستور بعدی را از حافظه دریافت می‌کند و سپس در مرحله FETCH_2 وارد فاز دکود می‌شود.

```
MEM_1: begin
    if (alu_done) begin
        case (instr_reg[15:13])
            3'b100: begin // LOAD
                mem_addr <= alu_out;
                mem_read <= 1;
                state <= MEM_2;
            end
            3'b101: begin // STORE
                mem_addr <= alu_out;
                mem_write <= 1;
                state <= MEM_2;
            end
        endcase
    end
end

MEM_2: begin
    state <= WRITEBK_1;
end
```

در مراحل DECODE_1 و DECODE_2، دستور خوانده‌شده در instr_reg ذخیره شده و فیلدهای مربوط به رجیسترها (مبدأ و مقصد)، نوع عملیات (عملیات ALU یا دستورات Load/Store)، و حالت Immediate بودن یا نبودن استخراج می‌گردد.

در مرحله EXEC_1 سیگنال alu_start فعال می‌شود و اجرای محاسبات در واحد ALU آغاز می‌گردد. در مرحله EXEC_2، بسته به نوع دستور، کنترلر تصمیم می‌گیرد به مرحله حافظه (MEM) یا مرحله ثبت نتیجه (WRITEBK) برود.

در حالت‌های MEM_1 و MEM_2، عملیات خواندن یا نوشتن در حافظه صورت می‌گیرد. در دستور Load، داده از حافظه خوانده می‌شود و در دستور Store، داده در حافظه نوشته می‌شود.

در مراحل WRITEBK_1 و WRITEBK_2، نتایج حاصل از ALU یا حافظه در رجیستر مقصد ذخیره شده و سیگنال rf_write جهت نوشتن در رجیستر فایل فعال می‌گردد.

سیگنال‌های کنترلی متعددی همچون alu_op, rf_write, rf_read, mem_read, mem_write, immediate_sel به طور دقیق در هر حالت تنظیم شده‌اند تا ارتباط صحیح میان اجزای مختلف سیستم برقرار گردد.

سیگنال ready نشان‌دهنده اتمام کامل اجرای یک دستور و آمادگی برای اجرای دستور بعدی است.

آدرس دستور جاری توسط pc نگهداری و پس از هر سیکل کامل (اجرای کامل یک دستور)، یک واحد افزایش می‌یابد.

```
WRITEBK_2: begin
    pc <= pc + 16'd1;
    ready <= 1;
    state <= FETCH_1;
end
```

3. Top:

ماژول top به‌عنوان بالاترین سطح یکپارچه‌کننده در طراحی ایفای نقش می‌کند و ارتباط میان زیرماژول‌های اصلی از جمله حافظه، ALU، رجیستر فایل و کنترلر را برقرار می‌سازد. این ماژول، سامانه‌ای کامل برای اجرای دستورالعمل‌های پردازشی را فراهم می‌کند.

نکات مهم و نحوه پیاده‌سازی:

1. اتصال مؤلفه‌های داخلی (ماژول‌های زیرین):

Memory (Main Memory):

- وظیفه خواندن و نوشتن داده و دستورالعمل‌ها را بر عهده دارد.
- ورودی‌ها: mem_read, mem_write, mem_addr, rf_data_rs1
- خروجی: mem_data_out

ALU (واحد حساب و منطق):

- عملیات ریاضی همچون جمع، تفریق، ضرب و تقسیم را بسته به کد عملیاتی (opcode) انجام می‌دهد.
- سیگنال کنترل شروع (alu_start) و اتمام (alu_done) از کنترلر مدیریت می‌شود.

Register File:

- وظیفه خواندن از دو رجیستر ورودی و نوشتن در رجیستر خروجی را دارد.
- آدرس‌دهی خواندن و نوشتن از طرف کنترلر انجام می‌گیرد.
- در حالت Immediate، به‌جای rs1 از rd به‌عنوان آدرس خواندن استفاده می‌شود (توسط read_addr1_real).

Controller:

- کنترل‌کننده اصلی فرآیند اجرای دستورالعمل‌ها است.
- سیگنال‌های کنترلی نظیر rf_read, rf_write, alu_start, mem_read و غیره را تولید می‌کند.
- سیگنال‌های خروجی آن به سایر واحدها ارسال شده و تعامل بین آن‌ها را تنظیم می‌کند.

2. مدیریت Immediate:

اگر دستور از نوع Immediate باشد (immediate_sel = 1)، آدرس رجیستر اول (به جای rs1) با rd جایگزین می‌شود.

3. سیگنال‌های خروجی خارجی:

pc_out: مقدار جاری شمارنده برنامه (PC)

ready_out: علامت اتمام اجرای یک دستور

Testbench

ماژول Processor_TB یک تست‌بنچ برای بررسی عملکرد ماژول top است که نقش پردازنده را ایفا می‌کند. هدف این تست‌بنچ، بررسی اجرای صحیح دستورالعمل‌ها، تغییرات رجیسترها و حافظه، و عملکرد کلی سیستم می‌باشد.

نکات مهم و نحوه پیاده‌سازی:

1. تولید کلاک:

با استفاده از بلاک always #5 clk = ~clk; سیگنال ساعت (clock) با تناوب 10 نانوثانیه تولید شده‌است.

2. راه‌اندازی اولیه سیستم:

در ابتدا، سیگنال ریست فعال شده و پس از 10 نانوثانیه (به اندازه یک کلاک) غیرفعال می‌شود تا پردازنده کار خود را آغاز کند.

3. مقداردهی اولیه به حافظه داده:

چهار موقعیت از حافظه (mem[16] تا mem[19]) با مقادیر مختلف مقداردهی شده‌اند. این مقادیر بعداً در دستورات LOAD و STORE استفاده می‌شوند.

4. مقداردهی به حافظه دستور (Instruction Memory):

دستورالعمل‌ها به‌ترتیب از آدرس 0 تا 10 در حافظه نوشته شده‌اند و شامل عملیات‌هایی از جمله

LOAD, ADD, MUL, DIV, SUB, STORE

نمونه روند اجرای برخی دستورات و خروجی‌ها:

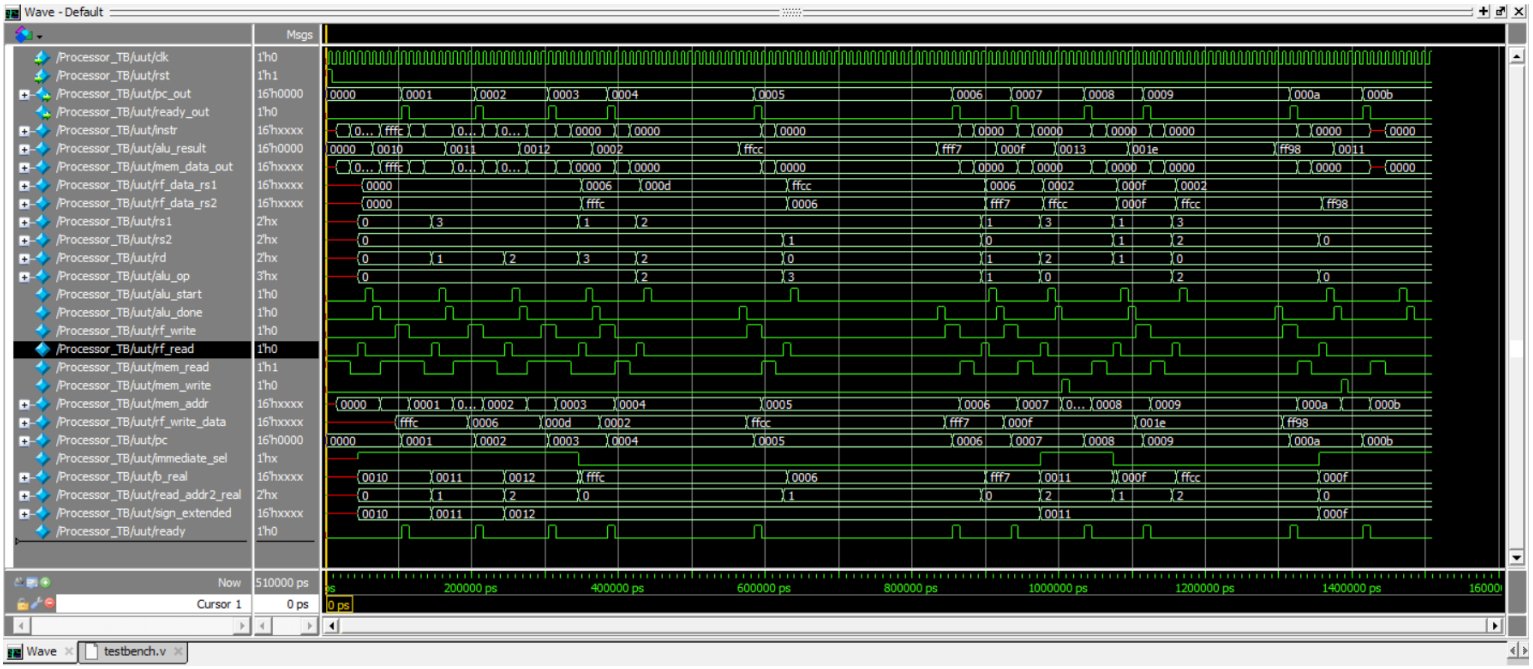
1. LOAD x0, -4 (از mem[16]):
   - x0 = -4
2. LOAD x1, 6 (از mem[17]):
   - x1 = 6
3. LOAD x2, 13 (از mem[18]):
   - x2 = 13
4. ADD x3, x1, x0:
   - x3 = 6 + (-4) = 2
5. MUL x2, x2, x0:
   - x2 = 13 * (-4) = -52
6. DIV x0, x2, x1:
   - x0 = -52 / 6 ≈ -9 (تقریب صحیح)
7. SUB x1, x1, x0:
   - x1 = 6 - (-9) = 15
8. STORE x2 → Mem[x3 + 17] = Mem[19]:
   - Mem[19] = -52
9. ADD x1, x1, x1:
   - x1 = 15 + 15 = 30
10. MUL x0, x3, x2:
    - x0 = 2 * (-52) = -104
11. STORE x0 → Mem[x3 + 15] = Mem[17]:
    - Mem[17] = -104

هستند. هر دستور 16 بیت است و فیلدهایی مانند opcode، رجیسترها و Immediate را شامل می‌شود.

خروجی‌ها:

در هر لبه بالا رونده‌ی کلاک، مقادیر رجیسترهای x0 تا x3 و حافظه‌های mem[16] تا mem[19] نمایش داده می‌شوند. همچنین مقدار PC نیز در هر سیکل نشان داده شده و تغییرات آن روند اجرای دستورها را تأیید می‌کند.

در مجموع، این تست‌بنچ با تعریف کلاک، اعمال ریست، مقداردهی اولیه به حافظه، و نمایش خروجی‌ها، عملکرد پردازنده را مرحله به مرحله بررسی و صحت اجرای آن را تأیید می‌نماید. دستورات تست‌بنچ نوشته شده در تصویر بالا نمایش داده شده است؛ همچنین waveform خروجی به صورت زیر است:



دقت کنید خروجی این تست در فایل output.txt قرار داده شده است.