Contents

[**تئوری عملکرد** 2](#_Toc111499476)

[**بررسی عملکرد درایور ماژول BH1750** 3](#_Toc111499477)

[**BH1750\_timing** 3](#_Toc111499478)

[**instruction\_sender\_FSM** 3](#_Toc111499479)

[**Reading\_result\_FSM** 3](#_Toc111499480)

[**BH1750\_application** 4](#_Toc111499481)

[**بررسی عملکرد کنترلر فازی** 5](#_Toc111499482)

[**Calculating membership functions equations** 5](#_Toc111499483)

[**Fuzzification** 6](#_Toc111499484)

[**sugeno implication** 7](#_Toc111499485)

[**بررسی عملکرد ماژول تبدیل Duty cycle به PWM** 8](#_Toc111499486)

# **تئوری عملکرد**

میزان روشنایی محیط توسط ماژول BH1750 اندازه گیری شده و از طریق پروتکل I2C به FPGA ارسال میگردد، راه اندازی این ماژول توسط کدی به زبان VHDL صورت میپذیرد که خروجی این بلوک میزان روشنایی محیط با یکای LUX با عرض 11 بیت می باشد(عددی بین 2047-0).

حال این باس 11 بیتی که حاوی میزان دقیق روشنایی محیط است باید با میزان روشنایی رفرنس مقایسه شده تا مقدار خطا حاصل گردد، این کار به سادگی، طبق کد زیر انجام شده است:

                ----- error calculation ----

        lux\_error           <= desired\_lux\_int - lux;

میزان خطا که یک باس 11 بیتی است در مرحله بعد به ورودی بلوک کنترل کننده فازی اعمال شده است. کنترل کننده فازی بر اساس توابع عضویت، سیستم استنتاج تعیین شده و هم چنین شروط فازی که از قبل برای آن مشخص شده اند، یک عدد 8 بیتی به عنوان میزان سیکل وظیفه که قرار است درادامه به منابع روشنایی اعمال گردد را تولید کرده و در خروجی قرار میدهد.

برای اعتبار سنجی این عدد 8 بیتی، کنترل کننده فازی در خروجی، سیگنالی تک بیتی به نام data\_valid هم تولید میکند، بنابراین هر زمان که خروجی مقداری Valid داشت به مرحله بعدی اعمال میشود.

        Duty\_cycle      <= change\_in\_DC when data\_valid = '1';

در ادامه این عدد 8 بیتی به بلوکی اعمال میشود تا براساس آن، سیگنال PWM را روی یکی از پایه های خروجی تولید کند.

در نهایت سیگنال PWM تولید شده به منابع روشنایی اعمال میشوند.

# **بررسی عملکرد درایور ماژول BH1750**

این درایور شامل دو فایل VHDL (دو بلوک یا ماژول ) است، در ادامه به بررسی کلی و نحوه عملکرد هر کدام از این دو بلوک میپردازیم :

## **BH1750\_timing**

اصول اینترفیس و تایمینگ ماژول BH1750 در این فایل توصیف شده است.

دراینطرح دو ماشین حالت برای اجرای روتین ارسال و دریافت از طریق پروتکل I2C در نظر گرفته شده است، تعریف سیگنال این دو ماشین حالت در زیر آورده شده است:

        signal      instruction\_sender\_FSM  : unsigned(3 downto 0) := (others => '0');

        signal      Reading\_result\_FSM      : unsigned(3 downto 0) := (others => '0');

در ادامه به بررسی کلی و نحوه عملکرد هر کدام از این دو ماشین حالت میپردازیم :

### **instruction\_sender\_FSM**

این ماشین حالت، مقدار دهی های اولیه و instruction های مورد نیاز را برای ماژول BH1750 ارسال میکند.

نحوه عملکرد این ماشین حالت به صورت زیر است:

1. waiting for 1.31 us (Tbuff) 1.3 us
2. waiting for (Thdsta) 0.61 us to create start condition and 0.75 us to start sending data
3. sending module address which is 92 in decimal
4. sending RW bit which is ‘0’ for now (we want to write)
5. receiving ack (no need to read acknowledge because we got only one slave here)
6. sending instruction (we get this instruction from “BH1750\_application” module which specifies the operation mode of BH1750).
7. receiving ack (no need to read acknowledge because we got only one slave here)
8. Stop condition

### **Reading\_result\_FSM**

این ماشین حالت، میزان روشنایی محیط را از ماژول BH1750 دریافت میکند.

نحوه عملکرد این ماشین حالت به صورت زیر است:

1. waiting for 1.31 us (Tbuff)
2. waiting for (Thdsta) 0.61 us to create start condition and 0.75 us to start sending data
3. sending module address which is 92 in decimal
4. sending RW bit, which is ‘1’ for now (we want to read)
5. reciving ack )no need to read acknowledge cause we got only one slave here(
6. recieving High byte
7. sending ack which is ‘0’
8. receiving Low byte
9. sending nack (we dont wanna receive anything else) so we send ‘1’
10. stop condition

## **BH1750\_application**

کنترل کننده ماژول BH1750\_timing است که محسابات لازم برای بدست آوردن میزان روشنایی محیط و همچنین مد کاری ماژول BH1750 را تعیین میکند.

ماژولBH1750\_timing در این ماژول instant شده است. دو کار اساسی که این ماژول میکند اولا این است که مد کاری BH1750 را تعیین میکند و ثانیا خروجی ماژول BH1750\_timing را بر 1.2 تقسیم میکند. (طبق گفته دیتاشیت برای محاسبه lux نهایی این کار حتما باید انجام شود) البته ممکن است این تقسیم بدون استفاده از هیچ آی پی خاصی کمی چالش برانگیز باشد. برای این کار طبق کد زیر ابتدا ضریبی تعریف میکنیم که در دیتای خام ضرب شود(به جای تقسیم بر 1.2 آنرا در 0.8 ضرب میکنیم).

constant coef               : signed (4 downto 0) := to\_signed(13,5); --- 0.8

در کد بالا مشاهده میشود که ثابت علامتدار 5 بیتی با مقدار 13 دسیمال تعریف شده است، اما این عدد چطور میتواند نمایانگر 0.8 باشد ؟

بیایید این عدد را به صورت باینری بررسی کنیم:

"01101"

حال اگر برای این عدد 4 رقم کسری در نظر بگیریم به صورت زیر درمیاید:

"0.1101"

این عدد 4 بیت اعشار دارد، و یک بیت علامت و بدون بیت صحیح است که نمایانگر عدد 0.8125 میشود.

در ادامه به راحتی این ثابت در دیتای خام ضرب شده است :

        product                 <= coef \* signed('0' & data\_out);

دیتای خام (data\_out) یک سیگنال 16 بیتی است که 16 بیت آن صحیح هستند و بیت علامت و کسری ندارد، چون در ادامه محاسبات علامتدار انجام شده اند بنابراین یک بیت علامت 0 به ابتدای دیتای خام میچسبانیم تا علامتدار شود سپس در coef آنرا ضرب میکنیم.

سیگنال product یک سیگنال 22 بیتی است، 4 بیت کسری، 17 بیت صحیح و 1 بیت علامت.

بدین ترتیب محاسبات لازم جهت بدست آوردن مقدار صحیح lux محیط انجام میپذیرد.

# **بررسی عملکرد کنترلر فازی**

این ماژول Fuzzifier نام دارد، و همه ی مراحل محاسبه توابع عضویت، فازی سازی، استنتاج و نافازی سازی در یک فایل توصیف شده اند.

کد این ماژول در سه section نوشته شده است که در ادامه هرکدام بررسی میشود:

## **Calculating membership functions equations**

در این section در واقع قرار است توابع عضویت با توجه به پارامترهای ورودی تعیین شده محاسبه شوند، چون در این کنترلر توابع عضویت مثلثی هستند بنابراین برای سه راس این مثلث ها مقادیر a,b,c باید تعیین شوند، چون این کنترلر داری یک ورودی است که این ورودی شامل 4 تابع عضویت مثلثی به نام های level 1 تا level 4 میشود بنابراین در مجموع برای این کنترلر 12 پارامتر ورودی جهت تعیین رئوس مثلث باید مقدار دهی شوند.

نحوه محاسبه شیب هر تابع عضویت به صورت زیر است :

Slope =

نحوه محسابه عرض از مبدا هرتابع به صورت زیر است :

Intercept = - (slope \* x1)

در واقع هر تابع مثلثی شامل دوتابع، یکی با شیب مثبت و یکی با شیب منفی است، بنابراین برای این کنترلر در مجموع باید 8 شیب و 8 عرض از مبدا محاسبه گردد.

باتوجه با اینکه برای محاسبه شیب و بعد ها برای محاسبات دیگر از آی پی تقسیم کننده استفاده شده است، نکته حائز اهمیت در پیاده سازی این محاسبات در FPGA این است که برای کاهش منابع مصرفی در عمل تنها از یک تقسیم کننده استفاده شده، چرا که زمان انجام این محاسبات چندان برای ما اهمیت ندارد(برای مثال نیازی نیست همه ی این 8 تقسیم به صورت همزمان انجام شوند) به صورت ترتیبی از این آی پی برای محاسبات استفاده شده، برای راحتی در دیباگ کردن طرح نهایی از ماشین حالت بهره گرفته شده است، این ماشین حالت FSM نام دارد و دارای حالات زیر است.

                ------------- FSM --------------

type state\_type is (idle,level\_1\_rise\_function\_state,level\_1\_fall\_function\_state,level\_2\_rise\_function\_state,level\_2\_fall\_function\_state,level\_3\_rise\_function\_state,level\_3\_fall\_function\_state,level\_4\_rise\_function\_state,level\_4\_fall\_function\_state,fuzzification\_state,sugeno\_implication\_state);

signal FSM :state\_type := idle;

ترتیب انجام محاسبات توسط این ماشین حالت در این section به صورت زیر است :

1. Level 1 calculating MF params
2. Level 2 calculating MF params
3. Level 3 calculating MF params
4. Level 4 calculating MF params

## **Fuzzification**

دراین section تنها کافی است مقدار ورودی را به توابع عضویت که در section قبل محاسبه شده بودند اعمال کنیم تا در نهایت میزان تعلق ورودی به هر تابع عضویت بدست آید.

این section تنها دارای یک حالت به نام fuzzification\_state است و درواقع کل فرایند فازی سازی در 1 کلاک (10 ns) انجام میشود.

در ادامه نحوه انجام اینکار توضیح داده میشود:

                    --- "level 1" MF ---

if (lux\_error\_int = level\_1\_a\_int ) then

 product1     <= (others => '0');

elsif (lux\_error\_int > level\_1\_a\_int and lux\_error\_int < level\_1\_b\_int ) then

 product1     <= (lux\_error\_int \* Level\_1\_rise\_slope) + level\_1\_rise\_intercept;

elsif (lux\_error\_int >= level\_1\_b\_int and lux\_error\_int <= level\_1\_c\_int) then

 product1     <= (lux\_error\_int \* Level\_1\_fall\_slope) + level\_1\_fall\_intercept;

elsif (lux\_error\_int > level\_1\_c\_int) then

 product1     <= (others => '0');

end if;

همانطور که مشخص است بیس اصلی مرحله فازی سازی، مقایسه است، اینکه تشخیص دهیم ورودی کنترلر در هرلحظه در بازه کدام یک از توابع عضویت قرار میگیرد (شروط if) و پس از اعمال آن ورودی به عنوان ورودی تابع مربوطه (ارجاع انجام شده بعد از هر شرط if) خروجی آن که عددی اعشاری بین 0 و 1 است بیانگر میزان تعلق ورودی، به آن تابع عضویت را نشان میدهد.

بعد از مرحله فازی سازی، چهار عدد اعشاری بین 0 و1 متناظر با چهار تابع عضویت بدست می آیند.

## **sugeno implication**

چرا که سیستم فازی سوگنو بهینه ترین روش برای پیاده سازی سخت افزاری است از سوگنو در این کنترلر استفاده شده است.

نحوه محاسبه مقدار نافازی شده نهایی در سیستم فازی سوگنو به شکل زیر است :

defuzzified value =

که W1 … Wn میزان وزن تابع عضویت اول تا n م و Z1 …Zn ضرایبی وابسته یا مستقل هستند که با توجه به خروجی کنترلر تنظیم میشوند.

کد VHDL این section در ادامه آورده شده است:

weight\_1     <= (level\_1\_int \* z1\_int) + (level\_2\_int \* z2\_int);

weight\_2     <= (level\_3\_int \* z3\_int) + (level\_4\_int \* z4\_int);

dividend     <= weight\_1 + weight\_2;

divisor      <= resize(level\_1\_int,11) + level\_2\_int + level\_3\_int + level\_4\_int;

همانطور که مشخص است در این قسمت هم از همان تقسیم کننده که در ابتدای کد استفاده شده بود استفاده شده است.

# **بررسی عملکرد ماژول تبدیل Duty cycle به PWM**

احتمالا این کد، ساده ترین ماژول در کل این پروژه است، در ادامه کد VHDL آن آورده شده است :

process(clock)

begin

    if rising\_edge(clock) then

        clock\_counter           <= clock\_counter +1;

        if (clock\_counter = to\_unsigned(0,10)) then

            counter             <= counter +1;

            PWM\_int             <= '0';

            if (counter < Duty\_cycle) then

                PWM\_int     <= '1';

            end if;

            if (counter = to\_unsigned(100,7)) then

                counter             <= (others=> '0');

            end if;

        end if;

    end if;

end process;

از آنجایی که همه ی اجزای این پروژه با کلاک 100 مگاهرتز کار میکنند، این ماژول هم از آنها مستثنا نیست، اما نکته اینجا است که ساخت PWM با فرکانس 100 مگاهرتز کاری غیر معقول به نظر میرسد، برای کم کردن فرکانس، شمارنده ای 10 بیتی به نام clock\_counter هر 1024 کلاک (تقریبا 10 میکروثانیه) سرریز میکند، و شمارنده دیگری به نام counter را افزایش میدهد، شمارنده counter وقتی به عدد 100 میرسد سرریز میکند، میزان duty cycle ورودی با شمارنده counter مقایسه شده و تا وقتی که بزرگتر از آن است خروجی PWM مقدار منطقی 1 به خود میگیرد و در غیر این صورت مقدار صفر به آن ارجاع داده میشود، به این ترتیب سیگنال PWM ساخته میشود.