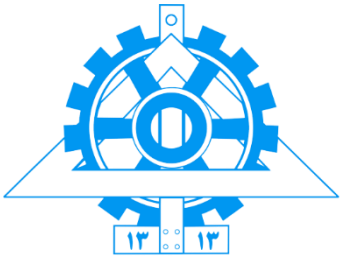
باسمه‌ی تعالی



Object-Oriented Modeling of Electronic Circuits

Computer Assignment 3

RTL Design and SystemC Simulation

Student Name: Moein Maleki
Student Number: 810197591

Electrical and Computer Engineering Department
Spring 1401

Table of Contents:

Deliverables:

بررسی ماژول FileReader و تصمیم های طراحی اتخاذ شده:

این کلاس در دو SC_THREAD جداگانه، فایل‌های داده‌ی کانال های تصویر را خوانده و برای ماژول GrayScaler میفرستند.

تصمیم های طراحی:

۱- آرایه‌ی دو بعدی channelData ، داده‌های پیکسل ها را ذخیره میکند. اندیس دهی در پارامتر اول آن (از صفر تا دو)، سه رنگ متفاوت را میدهد و اندیس دهیِ دوم آن (از 0 تا 1 - 512*512)، پیکسل مورد نظر را میدهد. با ذخیره‌ی داده های به این صورت، براحتی میتوان به روش زیر، یک پیکسل مسقل از آرایه‌ی ما، ساخت و بعدا برای ماژول بعدی فرستاد.

```
sc_lv<8> pixel[3] = {
    channelsData[RED][px],
    channelsData[GREEN][px],
    channelsData[BLUE][px]
};
```

۲- ارسال داده ها همانطور که در صورت پروژه گفته است، به گونه‌ای باید باشد تا داده های کانال مختلف پیکسل در کلاک سایکل های متوالی بتواند برای ماژول GrayScaler فرستاده شود. بنابراین Interface ارتباطی این دو ماژول باید در سطح پیکسل فعالیت کند. (این نکته در GrayScaler بیشتر توضیح داده شده است.)

```
fileReader_out->send_pixel(pixel);
```

۳- به منظور جداسازی دو عملِ "خواندن فایل های داده" و "ارسال داده ها به GrayScaler" نیاز است یک sc_event در کلاس FilerReader داشته باشیم تا این دو SC_THREAD را بتوانیم Synchronize کنیم. ولی به دلیل اینکه سیمولیشن عملا به هیچ زمانی در دنیای واقعی جلو نمیرود، اگر صرفا از Notify و Wait استفاده کنیم، چون SC_THREAD ها به صورت سری Active میشوند، هیچوقت SC_THREAD ای که Wait میکند، Notify را نمیبینند. بدین منظور، وقتی میخواهیم sc_event را Notify کنیم، این کار را به میزان یک دلتا سایکل به تاخیر می‌اندازیم تا خط Wait اجرا شود تا در مرحله‌ی بعد، Notify ما دیده شود.

```
    cout << "reading done" << endl;
    readingDone.notify(SC_ZERO_TIME);
}

void fileReader::sendDataToGrayscale()
{
    cout << "sendDataToGrayscale active" << endl;
    wait(readingDone);
```

۴- کانال پیاده سازی شده برای ارتباط FileReader به GrayScaler، pixel_channel نام دارد و دو Method با نام های send_pixel و get_pixel دارد. sc_event های این کانال dataReady و receiverReady هستند. این کانال برای اطمینان از ارسال داده ها به روش صحیح، و برای افزایش پایداری، از sc_mutex ای به نام channelBusy استفاده میکند.

بررسی ماژول GrayScaler و تصمیم های طراحی اتخاذ شده:

این ماژول در دو SC_THREAD جداگانه، داده ها را دریافت میکند و برای EdgeDetector میفرستد. این ماژول پس از دریافت هر پیکسل از FileReader، مقدار GrayScaler آنرا حساب میکند و در آرایه‌ی داخلی grayScaledData ذخیره میکند. پس از آماده شدن هر Segment، مقدار پارامتر lastReadySegment که شماره آخرین Segment آماده شده را در خود ذخیره دارد، یک عدد افزایش پیدا میکند. پارامتر lastServedSegment نیز برای ذخیره شماره آخرین Segment ای که به صورت موفقیت آمیز برای EdgeDetector فرستاده شده است، تعبیه شده است. در ادامه به جزئیات بیشتری میپردازیم.

تصمیم های طراحی:

۱- همانطور که در FileReader اشاره کردیم. Interface بین FileReader و GrayScaler در سطح پیکسل پیاده سازی شده است. نکته‌ی جدید حول این مسئله، اینست که چون در GrayScaler پس از دریافت هر پیکسل، میخواهیم مقدار میانگین سه کانال را محاسبه کنیم، اگر به صورت Burst داده‌های پیکسل ها را برای GrayScaler بفرستیم، آن موقع دیگر نمیتوانیم به صورت سایکل به سایکل، تعداد Segment های آماده شده را کنترل کنیم. زیرا محاسبه‌ی مقدار میانگین داده های ورودی، گره‌ی تنگاتنگی با ارسال داده ها به EdgeDetector دارد.

۲- در SC_THREAD ای که مسئول ارسال داده ها به EdgeDetector است، به صورت مداوم باید چک کنیم که اگر Segment جدیدی برای ارسال وجود دارد، برای EdgeDetector آنرا بفرستیم. بدین منظور ار ساختار زیر استفاده شده است.

```
while (1) {
    if (lastServedSegment == NUM_OF_SEGMENTS - 1)
        break;
    wait(newSegmentReady);
    while (lastServedSegment != lastReadySegment) {
        sc_lv<8>* newSegment = selectSegment(grayScaledData, lastServedSegment + 1);
        bool aRequestWasObserved = grayScaler_out->send_burst(newSegment, SEGMENT_SIZE);
        if (aRequestWasObserved == false)
            continue;
        lastServedSegment++;
```

۳- در ارسال داده ها به EdgeDetector میخواهیم در هر سایکل، داده‌های هشت بیتی آماده شده‌ی Segment مورد نظر را ارسال کنیم. بدین منظور برای ارسال داده ها، از روش burst استفاده شده است.

۴- ارسال داده ها برای EdgeDetector، گیرنده-محور است. بدین معنا که حتما طرف گیرنده باید با اجرای Method get_burst درخواستی ثبت کند تا داده‌ای برایش فرستاده شود. اگر درخواستی ثبت نشود، send_burst داده‌ای نمیفرستد و مجددا منتظر میماند تا درخواستی ثبت شود.

```
bool burst_channel_reciever_based::send_burst(sc_lv<8>* segment, int burstSize) {
    channelBusy.lock();
    if (requestedSegment == false) {
        channelBusy.unlock();
        return false;
    }
    segmentRequestReady.notify(SC_ZERO_TIME);
    wait(responceAcknowledged);
    cout << "REQUEST OBSERVED" << endl;
    for (int px = 0; px < burstSize; px++) {
        wait(clk->posedge_event());
        pixelData = segment[px];
        wait(SC_ZERO_TIME);
    }
    wait(clk->posedge_event());
    pixelData = 0;
    channelBusy.unlock();
    return true;
}

bool burst_channel_reciever_based::get_burst(sc_lv<8>* segment, int burstSize) {
    cout << "REQUESTED A SEGMENT" << endl;
    requestedSegment = true;
    wait(segmentRequestReady);
    responceAcknowledged.notify(SC_ZERO_TIME);
    cout << "NOW RECEIVING A SEGMENT" << endl;
    for (int px = 0; px < burstSize; px++) {
        wait(clk->posedge_event());
        wait(SC_ZERO_TIME);
        segment[px] = pixelData;
    }
    requestedSegment = false;
    return true;
}
```

بررسی ماژول EdgeDetector و تصمیم های طراحی اتخاذ شده:

این ماژول، دو حافظه‌ی درونی دارد. حافظه‌ی اصلی به نام segmentSlot که به وسیله‌ی آن، محاسبات کرنل های سوبل انجام میشود و اندازه‌ی آن (IMAGE_COLS_PX) * (SEGMENT_ROWS_PX + 2) میباشد. و حافظه‌ی جانبی به نام newSegment که Segment دریافت شده از GrayScaler به صورت موقت در آن قرار میگیرد و به اندازه‌ی یک Segment است. پس از دریافت هر Segment از GrayScaler قبل از آنکه آنها را در آرایه‌ی اصلی بگذاریم، ابتدا دو سطر پایینی موجود در segmentSlot که مربوط به Segment قبلی میباشد، به دو سطر بالایی segmentSlot منتقل میشوند. پس از آن newSegment در سطر سوم تا سطر آخر segmentSlot قرار میگیرد تا در مرحله‌ی بعدی، محاسبات کرنل انجام شود. در ادامه به جزئیات این ماژول میپردازیم.

تصمیم های طراحی:

۱- ارتباط این ماژول با GrayScaler از جانب Edgedetector، Initiate میشود. بدین ترتیب که هر موقع پردازش Segment کنونی تمام شد، با اجرای خط

```
edgeDetector_in->get_burst(newSegment, SEGMENT_SIZE);
```

درخواست Segment جدید، در کانال ارتباطی این دو ماژول قرار میگیرد تا بعدا GrayScaler با اجرای خط:

```
grayScaler_out->send_burst(newSegment, SEGMENT_SIZE);
```

آنرا پاسخ دهد.

۲- یک SC_THREAD برای جابجایی دو سطر پایینی و گذاشتن آنها در دو سطر بالایی segmentSlot استفاده شده است.

```
while (1) {
    wait(reorganize);
    for (int px = 0; px < 2 * IMAGE_COLS_PX; px++) {
        segmentSlot[px] = segmentSlot[SEGMENT_SIZE + px];
    }
    reorganizeDone.notify(SC_ZERO_TIME);
```

۳- یک SC_THREAD برای قرار دادن newSegment در جای صحیح در segmentSlot استفاده شده است.

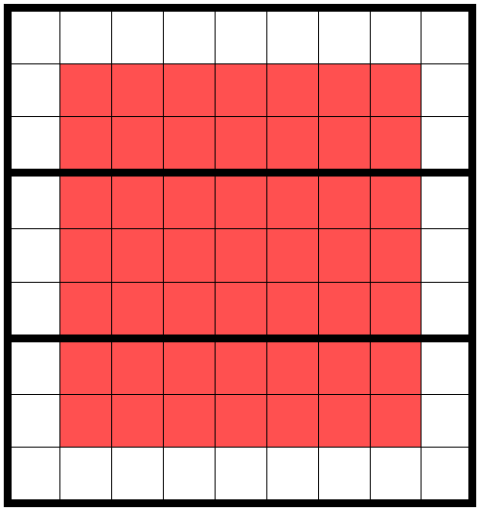
```
while (1) {
    wait(setNewSegmentIn);
    for (int px = 0; px < SEGMENT_SIZE; px++) {
        segmentSlot[px + 2 * IMAGE_COLS_PX] = newSegment[px];
    }
    setNewSegmentInDone.notify(SC_ZERO_TIME);
```

۴- یک SC_THREAD برای انجام محاسبات کرنل سوبل و اعمال آنها در segmentSlot استفاده شده است.

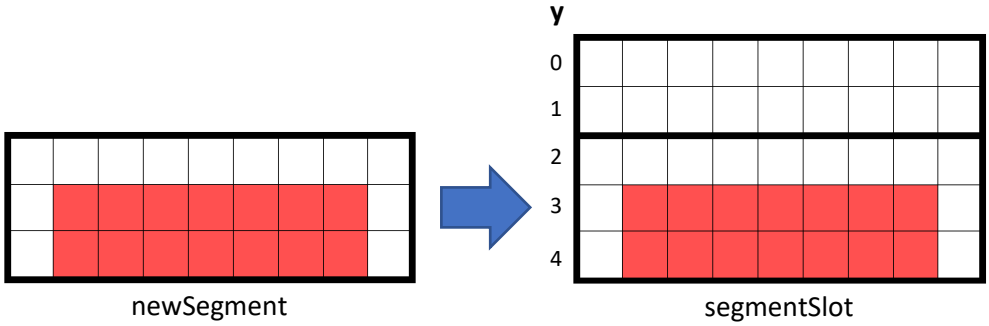
```
while (1) {
    wait(applyKernelsToSegment);
    for (int y = 1; y < SEGMENT_ROWS_PX + 1; y++) {
        for (int x = 1; x < IMAGE_COLS_PX - 1; x++) {
            Gx[y * IMAGE_COLS_PX + x] = calculateGx(x, y, segmentSlot);
            Gy[y * IMAGE_COLS_PX + x] = calculateGy(x, y, segmentSlot);
        }
    }
    applyKernelArrays(segmentSlot, Gx, Gy, currentSegmentUnder);
    applyKernelsToSegmentDone.notify(SC_ZERO_TIME);
```

```
void applyKernelArrays(sc_lv<8>* segmentSlot, int* Gx, int* Gy, int currentSegmentUnder) {
    for (int y = 1; y < SEGMENT_ROWS_PX + 1; y++) {
        if (currentSegmentUnder == 0 && y < 3)
            continue;
        for (int x = 1; x < IMAGE_COLS_PX - 1; x++) {
            segmentSlot[y * IMAGE_COLS_PX + x] =
                abs(Gx[y * IMAGE_COLS_PX + x]) +
                abs(Gy[y * IMAGE_COLS_PX + x]);
        }
    }
}
```

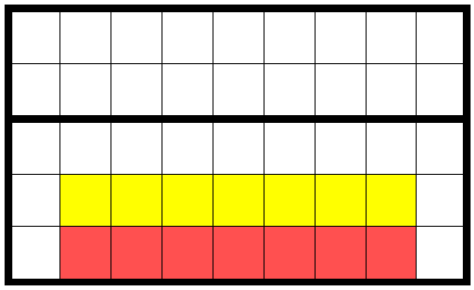
۵- در تابع applyKernelArrays مشاهده میکنیم که در Segment اول، برای سه سطر اول segmentSlot ، محاسبات سوبل اعمال نمیشوند. با ترسیم، بهتر میتوان دلیل این کار را فهمید. اگر شکل زیر تصویر ما باشد آنگاه میخواهیم در خانه‌های قرمز رنگ، محاسبات سوبل اعمال شود. در تصویر زیر Segment ها سه سطر دارند و با خطوط پر رنگ تر مشخص شده اند:



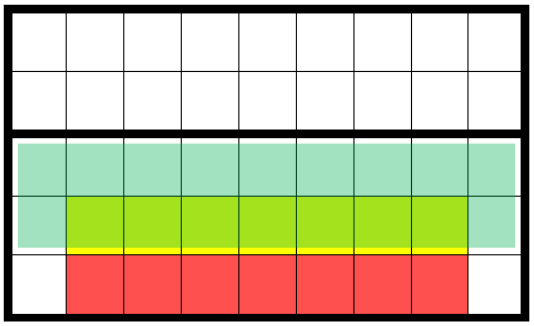
اگر Segment اول، از GrayScaler برای ما بیاید، به صورت زیر، در segmentSlot قرار داده میشوند. مشاهده میکنیم، برای دو سطر اول، نیازی نیست داده ای نوشته شود چون در ناحیه‌ی قرمز رنگ قرار ندارند و سطر سوم اتفاقا میخواهیم داده های GrayScaler خود را حفظ کند و نمیخواهیم محاسبات کرنل سوبل آنرا خراب کند، به همین منظور شرط گفته شده بررسی میشود:



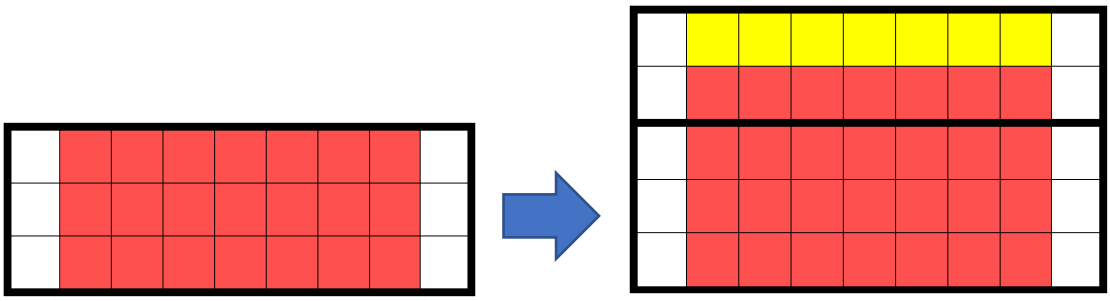
۶- در این قسمت نحوه‌ی انجام محاسبات را بررسی میکنیم. اگر همانند قبل Sement اول را دریافت کنیم، آنگاه در خانه های زرد رنگ محاسبات سوبل انجام شده و در segmentSlot قرار میگیرند. بنابراین هدف ما اینست که ناحیه های زرد رنگ فقط و به صورت کامل ناحیه های قرمز رنگ را پوشش دهند.



اولین Segment به صورت خاص ارسال میشود بدین ترتیب که ناحیه های سبز رنگ برای FileWrite ارسال میشوند.

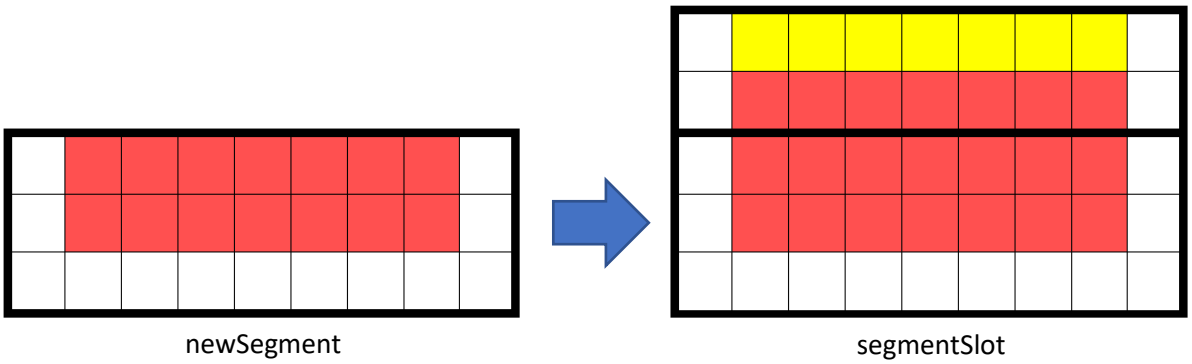


هنوز یک سطر قرمز مانده است که باید برایش محاسبات سوبل انجام شود و برای FileWriter ارسال شود. در این مرحله، دو سطر پایینی segmentSlot به دو سطر بالایی برده میشوند. Segment شماره‌ی دوم می‌آید و در جایگاه خود قرار میگیرد:

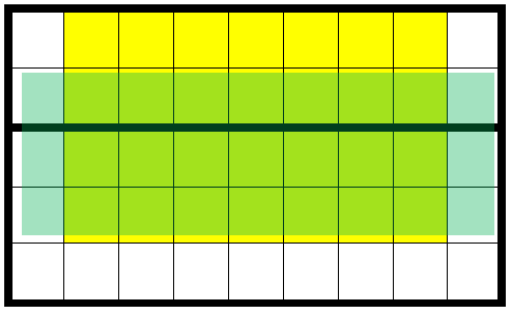


حال برای ناحیه های زرد جدید، محاسبات انجام شده و همانند قبل، ناحیه های سبز رنگ برای FileWriter ارسال میشوند.

در این مرحله، دو سطر پایینی segmentSlot به دو سطر بالایی برده میشوند. Segment شماره‌ی آخر می‌آید و در جایگاه خود قرار میگیرد:



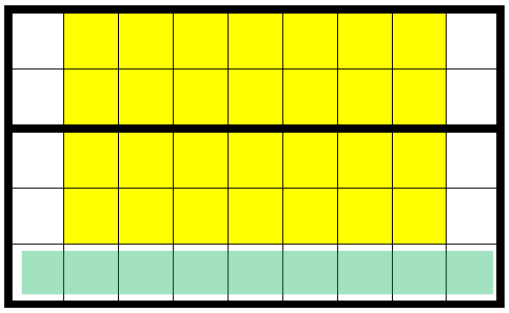
حال برای ناحیه های زرد جدید، محاسبات انجام شده و همانند قبل، ناحیه های سبز رنگ برای FileWriter ارسال میشوند.



حال، سطر آخر تصویر باقی مانده است که با اینکه نیازی به انجام محاسبات سوبل ندارد، ولی هنوز ارسال نشده است. این سطر در یک ارسال جداگانه در خط:

```
edgeDetector_out->send_burst(&segmentSlot[(SEGMENT_ROWS_PX + 1) * IMAGE_COLS_PX], IMAGE_COLS_PX);
```

میفرستیم.



۷- به خاطر این سطر آخر را در یک ارسال جداگانه میفرستیم و نه با Segment آخر، چون اینطوری دیگر نیاز نیست حافظه‌ی (SEGMENT_ROW + 1) * IMAGE_COLS_PX در FileWriter داشته باشیم.

۸- در ارتباط این ماژول به FileWriter نیاز به ارتباط burst داریم تا داده های Segment آماده شده، به صورت متوالی برای FileWriter فرستاده شود. در این پیاده سازی، از همان Interface ای که برای ارتباط GrayScaler و EdgeDetector، طراحی کردیم، استفاده میکنیم با این تفاوت که در پیاده‌سازی Method های کانال جدید، صرفا یک کانال جدید طراحی میکنیم که فرستنده-محور باشد.

```
bool burst_channel_initiator_based::send_burst(sc_lv<8>* segment, int burstSize) {
    channelBusy.lock();
    segmentReadyToBeSent.notify(SC_ZERO_TIME);
    wait(receiverReady);
    for (int px = 0; px < burstSize; px++) {
        wait(clk->posedge_event());
        pixelData = segment[px];
        wait(SC_ZERO_TIME);
    }
    return true;
}

bool burst_channel_initiator_based::get_burst(sc_lv<8>* segment, int burstSize) {
    wait(segmentReadyToBeSent);
    receiverReady.notify(SC_ZERO_TIME);
    for (int px = 0; px < burstSize; px++) {
        wait(clk->posedge_event());
        wait(SC_ZERO_TIME);
        segment[px] = pixelData;
    }
    channelBusy.unlock();
    return true;
}
```

بررسی ماژول FileWriter و تصمیم های طراحی اتخاذ شده:

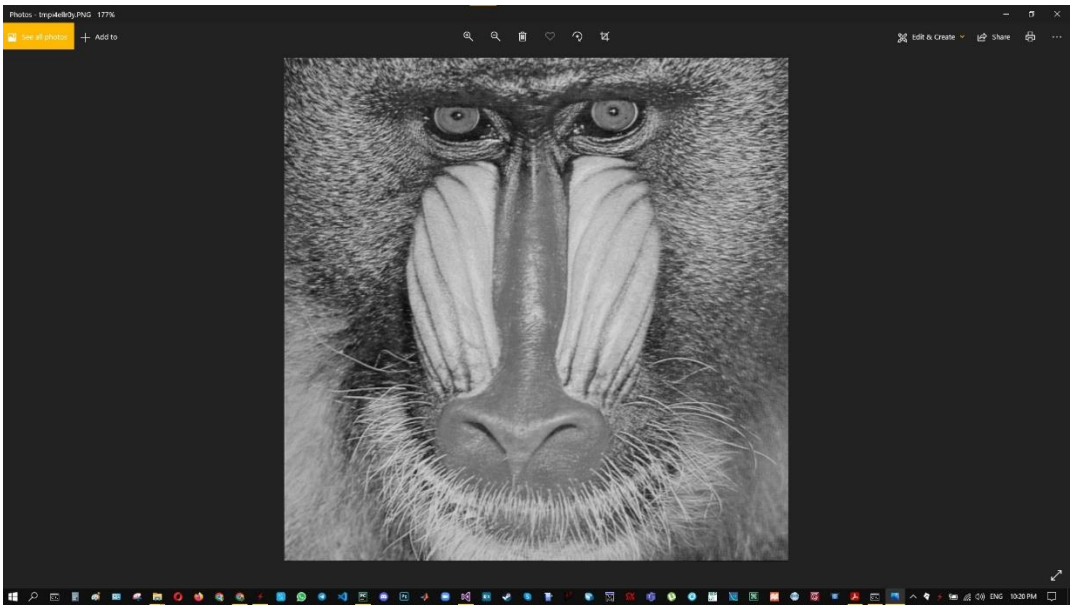
در این ماژول، داده ها به صورت بسته هایی به اندازه Segment از جانب EdgeDetector می‌آیند. این بسته ها همگی به دقیقا به اندازه‌ی یک Segment نیستند. در کل به تعداد IMAGE_ROWS / SEGMENT_ROWS + 1 بسته‌ی داده برای FileWriter فرستاده میشود. بسته‌ی متعلق به Segment اول، یک سطر کمتر دارد و بسته‌ی آخر تنها یک سطر است.

تصمیم های طراحی:

در قسمت های قبل به نکات مهم این بخش اشاره شده است.

تصاویر خروجی:

خروجی GrayScaler:



خروجی FileWriter: