УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
Трг Доситеја Обрадовића 6, 21000 Нови Сад, Република Србија
Деканат: 021 6350-413; 021 450-810; Централа: 021 485 2000
Рачуноводство: 021 458-220; Студентска служба: 021 6350-763
Телефакс: 021 458-133; e-mail: ftndean@uns.ac.rs
ИНТЕГРИСАНИ СИСТЕМ МЕНАЏМЕНТА СЕРТИФИКОВАН ОД:

ДОКУМЕНТАЦИЈА ЗА ПРЕДМЕТ: ПРОЈЕКТОВАЊЕ СЛОЖЕНИХ ДИГИТАЛНИХ СИСТЕМА

ТЕМА ПРОЈЕКТА:

Детекција слободних паркинг места применом функција за обраду слике

ПРОЈЕКАТ ИЗРАДИЛЕ:

Група 3: Ивана Гордић ЕЕ 23/2020
Ана Мокан ЕЕ 37/2020
Нина Антић ЕЕ 47/2020

Садржај

1. Опис алгоритма

1.1 Уводни део

Тема пројекта је проналажење и маркирање слободних паркинг места на видеу надзорне камере паркинга. Такође, омогућено је праћење броја заузетог у односу на укупан број паркинг места. Рад система заснован је на примени различитих функција за обраду слике где су у средишту *Gaussian Blur* и *Adaptive Threshold* као методе обраде слике модификацијом вредности на нивоу пиксела.

1.2 Главне функције

1.2.1 *Gaussian Blur*

Ефекат замућења, *Image blurring* постиже се конволуцијом оригиналне слике са матрицом (кернелом) нископропусног филтра. Користи се за одстрањивање садржаја високе фреквенције, попут шума и ивица. *Gaussian blur* је један специфичан тип филтрирања слике где се за конволуцију користи *Gaussian kernel* који на основу подешавања одређених параметара (величине кернела, стандардне девијације - сигма) добија коефицијенте израчунате Гаусовом функцијом.

Коефицијенти кернела прате Гаусову расподелу, што значи да ће средиште квадратне матрице имати највећу тежину а крајеви мању. Подешавање вредности сигме одређују колико ће се постепено одвијати тај прелаз, а тиме и колико ће "јак" утицај имати суседни пиксели на онај који је тренутно у фокусу, док величина матрице одређује површину суседства које узимамо у обзир при обрађивању тренутног пиксела.

1.2.2 *Adaptive Threshold*

Функција која зависно од вредности прага компарације додељује пикселу одређену вредност - црно или бело. Код *Adaptive Thresholding*-а вредност прага (*threshold*) није унапред дефинисана већ се израчунава у односу на неки мањи регион слике коју обрађујемо и пикселе суседне оном у фокусу. За наше потребе коришћен је Гаусов тип израчунавања прага функције.

1.3 Опис алгоритма

Програм за одређивање броја слободних паркинг места се састоји из неколико делова.

Део програма написан је помоћу *python* програмског језика. Овај део узима фрејмове улазног видеа, фрејм, односно фотографију конвертује у црно-белу фотографију. Ова црно-бела фотографију представља улаз самог *C++* алгоритма.

Део кода написан *python* програмским језиком такође прима и број слободних паркинг места (излаз *C++* алгоритма), те врши њихово обележавање на фотографији и приказује поменуту фотографију Овај део кода није био укључен у профајлирање перформанси.

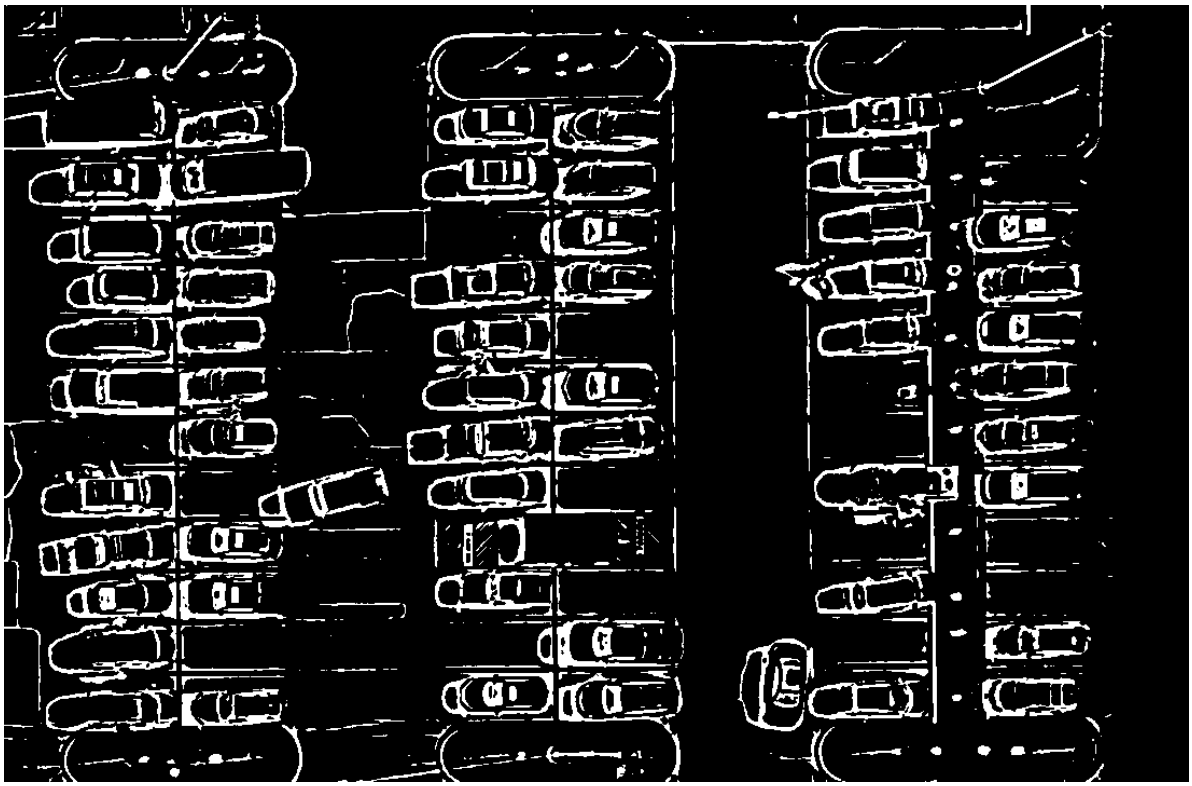
Сам *C++* алгоритам за проналазак слободних паркинг места се састоји из следећих сегмената:

1. Стварање Гаусовог кернела за замућење фотографије уз помоћ Гаусове функције, врши се два пута: први пут са кернелом величине 3 пута 3, а затим са величином кернела од 15 пута 15, до 25 пута 25, пропорционално величини фотографије (функција ***gaussianBlur***).
2. Примена Гаусовог кернела на фотографију, односно замућење фотографије, (функција ***applyGaussian***), позиви ове функције врше се унутар функције ***gaussianBlur,***
3. Одређивање вредности прага у односу на који пиксели фотографије постају или црни или бели, врши се у односу на вредности пиксела у околини одређеној величином кернела при другом замућивању фотографије (функција ***adaptiveThreshold***); други позив ***gaussianBlur*** функције одиграва се унутар ***adaptiveThreshold*** функције,
4. Проширивање области фотографије на којима се налазе бели пиксели (функција ***dilateImage***),
5. Проналазак слободних паркинг места, (функција ***checkParkingSpace***), унутар ове функције се врши отварање текстуалног фајла са координатама паркинг места, издвајање делова фотографије означених тим координатама позивом функције ***cropImage***. Затим се врши пребројавање белих пиксела на издвојеним деловима фотографије функцијом ***countNonZero*** и поређење броја белих пиксела са прослеђеном граничном вредношћу како би се одредило да ли је паркинг место слободно. Ова гранична вредност представља потребан број белих пиксела у оквиру једног паркинг места, да би се оно сматрало заузетим. Она се утврђује експериментално7 и зависи од величине слике.

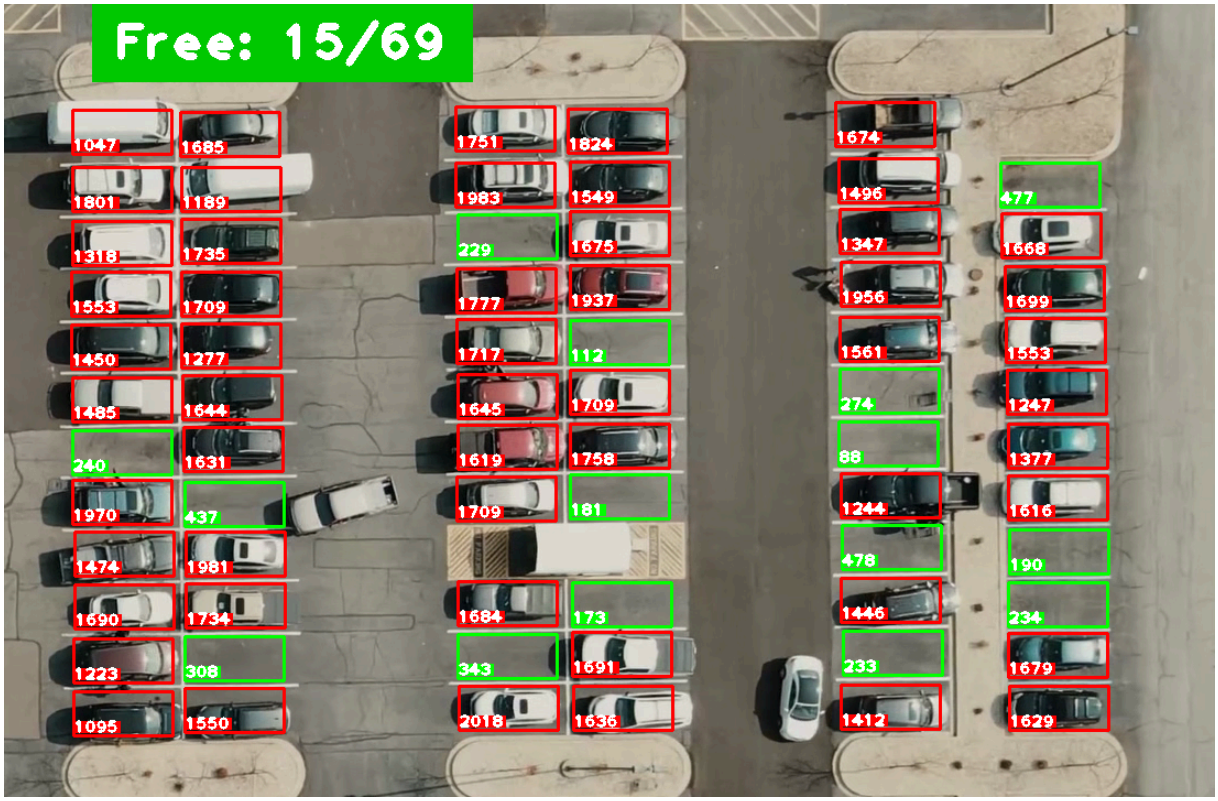
Функција ***checkParkingSpace*** враћа низ нула и јединица, где јединице означавају заузета паркинг места, а нуле слободна паркинг места. Овај низ представља излаз алгоритма.

Претходним анализама перформанси утврђено је да се временски критичан део кода налази унутар ***applyGaussian*** функције и одлучено је да ће се он убрзати имплементацијом у хардверу. Такође, резултати битске анализе показали су оптималан број бита за ширину података регистара у којима ће се складиштити вредности променљивих од значаја.

Слика 1. Изглед фотографије након замућења Гаусовим кернелом 25x25



Слика 2. Изглед фотографије након примене прага у оквиру функције adaptiveThreshold



Слика 3. Приказ слободних паркинг места у python скрипти

2. Уклањање петљи

Корак након одабира дела кода који се треба убрзати хардвером је уклањање петљи како би се дате операције могле превести на *RTL* ниво. На *сликама 4* и *5* следе редом део кода из изворног *C* алгоритма и код након уклањања угњежђених петљи који ће даље послужити као смерница за цртање *ASM* дијаграма и имплементацију контролне логике и тока података.

```
for (int i = 0; i < height; i++)
{
    for (int j = 0; j < width; j++)
    {
        sumPixel = 0.0;

        // konvolucija matrice kernela (filtra) i matrice slike

        for (int k = -half_size; k <= half_size; k++)
        {
            for (int l = -half_size; l <= half_size; l++)
            {
                rowIndex = min(max(i + k, 0), height - 1);
                colIndex = min(max(j + l, 0), width - 1);
                sumPixel += kern_mat[k + half_size][l + half_size] * gray_img[rowIndex][colIndex];

            }

        }


        blurred[i][j] = conv2int(sumPixel,16,8);

    }

}
```

Слика 4. C++ код који ће се бити акцелиран хардвером

```
initial:
        j = 0; i = 0;
        l = -half_size; k = -half_size;
        sumPixel = 0.0;
        rowIndex = 0; colIndex = 0;
start_sum:
        rowIndex = min(max(i + k, 0), height - 1);
        colIndex = min(max(j + l, 0), width - 1);
        sumPixel += kern_mat[k + half_size][l + half_size] * gray_img[rowIndex][colIndex];
        if( k >= half_size)
        {
            k = -half_size;
            if( l >= half_size)
            {
                l = -half_size;
                goto write_res;
            }
            else
            {
                l++;
                goto start_sum;
            }
        }
        else
        {
            k++;
            goto start_sum;
        }

    write_res:
            blurred[i][j] = sumPixel;
            sumPixel = 0.0;
            if(j >= width)
            {
                j = 0;
                if( i >= height)
                {
                    i = 0;
                    goto end_loop;
                }
                else
                {
                    i++;
                    goto start_sum;
                }
            }
            else
            {
                j++;
                goto start_sum;
            }
    end_loop:
            nop;
```

Слика 5 - Код након уклањања петљи

Треба напоменути да унутар кода са *слике 5* нису приказана решења приступања меморијама путем адреса, за које је потребна додатна логика израчунавања. У току развоја

виртуалне платформе издвајају се следећи меморијски подсистеми : *DRAM* (као део софтвера где се складиште оригинална и обрађена слика), *kernel_bram (*уместо променљиве *kernel_matrix*, чува коефицијенте кернела) и *img_bram* (меморија за складиштење делова слике који се више пута користе у току обраде). Посебно је значајан *img_bram* јер захтева допуну података након што се испуни одређени услов, где је потребно израчунати позиције података из оригиналне слике које треба прочитати, као и адресе унутар брама на које их треба уписати.
Осим рачунања адреса и допуне *img_bram* меморије, реимплементирана је *C*-овска функција дељења по модулу (“%”) како би се смањио број потребних ресурса на плочи.

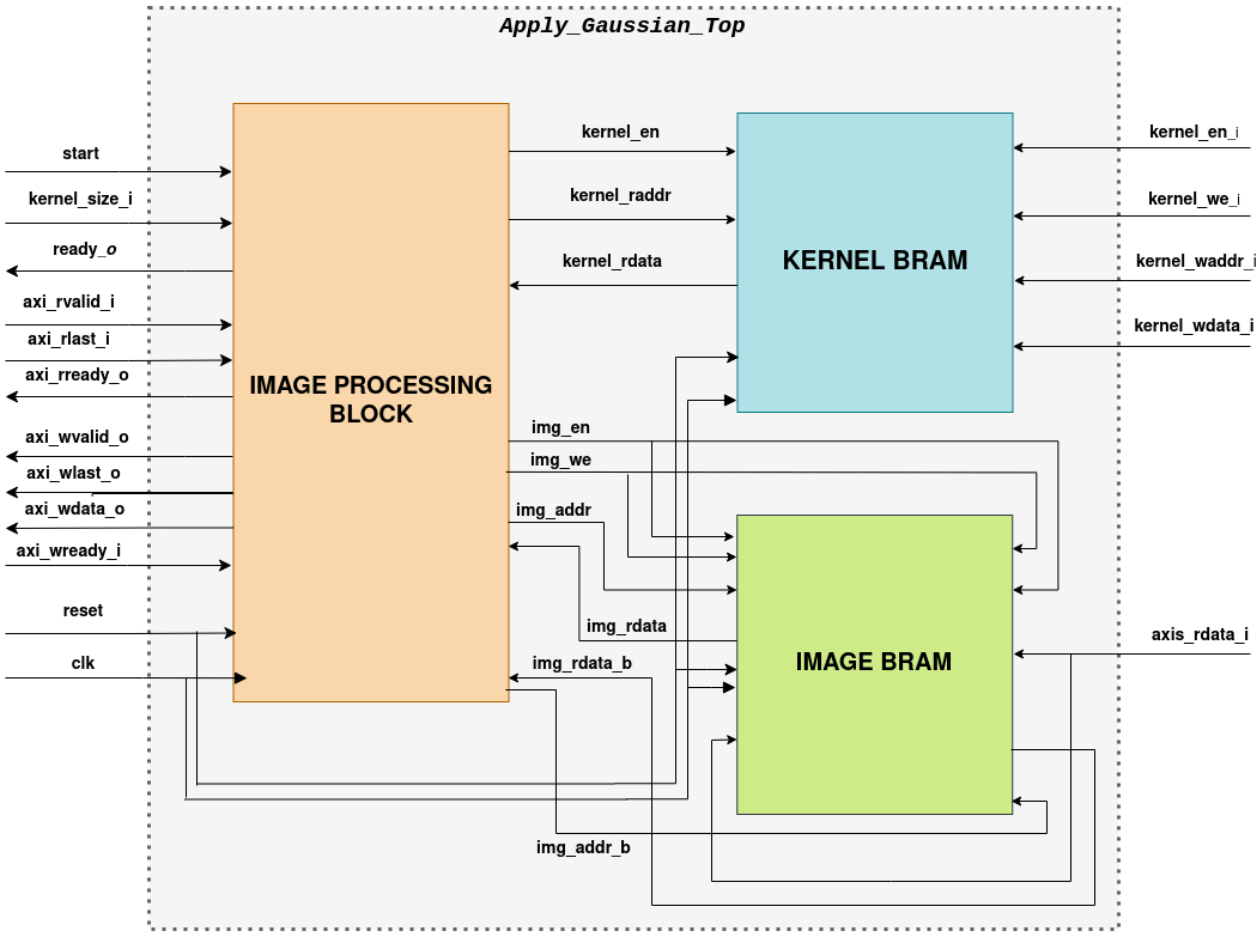
Број стања *ASM* дијаграма разликује се од броја лабела у коначном коду без петљи датом на *слици 6.* Разлог томе је додатно време потребно да се додели вредност сигналу и време да се добије податак из меморије (новодобијене вредности могу се користити тек у наредном такту).

```
init:
    sc_uint<11> rowIndex = 0, colIndex = 0;
    sc_int<5> halfSize = kernel_size/2;
    sc_uint<8> image = 0;
    cache_size = kernel_size*height_hw;
    req_cnt = 0;
    bram_addr = 0;
    sc_uint<5> req_cnt_mod = 0;   //represents req_cnt % kernel_size
    sc_uint<16> cache_size_mod = 0;     // represents ((colIndex - req_cnt + req_cnt_mod) * height_hw + rowIndex) % cache_size
    sumPixel = 0.0;

working:
// indexing logic
    if(i + k > 0)
    {
        if(i + k < height_hw - 1)
            rowIndex = i + k;
        else
            rowIndex = height_hw - 1;
    }
    else
        rowIndex = 0;

    if(j + l > 0)
        if(j + l < width_hw - 1)
            colIndex = j + l;

        else
            colIndex = width_hw - 1;
    }
    else
        colIndex = 0;
//--------------------------------
    matrix_kernel = read_kernel_bram((l + halfSize)*kernel_size + k + halfSize);
    cache_size_mod =  (colIndex - req_cnt + req_cnt_mod) * height_hw + rowIndex;

    if(cache_size_mod >= cache_size)
        image = read_img_bram(cache_size_mod -cache_size);
    else
        image = read_img_bram(cache_size_mod);

    sumPixel += matrix_kernel * image;
    //loop finished conditions
    if(k >= halfSize)
    {   //end of the inner k-loop
        k = -halfSize;
        if(l >= halfSize)
        {
            //end of the outer l - loop
            l = -halfSize;
            if (i==719 && j < (width_hw - halfSize - 1) && j >= halfSize)
            {
                dram_pos = kernel_size + req_cnt;
                bram_addr = req_cnt_mod*height_hw;
                n = 0;
                goto cache_add;
            }
            else
                goto write_res;
```

```
        else
        {
            l++;
            goto  working;
        }
    }
    else
    {
        k++;
        goto working;
    }

cache_add:
        pixel = read_dram(dram_pos * height_hw + n);
        write_img_bram(bram_addr, pixel);
        bram_addr++;
        if(n = height_hw - 1)
            goto cache_req_count;
        else
        {
            n++;
            goto cache_add;
        }

cache_req_count:
        req_cnt++;
        req_cnt_mod++;
        if(req_cnt_mod == kernel_size){ req_cnt_mod = 0; }
        goto write_res;

write_res:
        write_dram((j*height_hw + i), conv2int(sumPixel,16,8));
        sumPixel = 0;
        if(j >= width_hw)
        { //end of the inner j-loop
            j = 0;
            if(i >= height_hw)
            {//end of the outer i-loop
                i = 0;
                goto the_end;
            }
            else
            {
                i++;
                goto working;
            }
        }
        else
        {
            j++;
            goto working;

        }
the_end: nop;
```

Слика 6 - Комплетан код након уклањања петљи

3. Интерфејс и окружење



Слика 7 - Блок схема топ модула

На слици 7 налази се блок схема модула који се састоји се од следећих блокова:

IMAGE PROCESSING BLOCK - Главни блок који имплементира тражену функционалност и унутар кога се налази контролна логика и логика токова података. Реализован је принципом проточне обраде у циљу побољшања временских перформанси. Блок бива стартован од стране софтвера, који му притом прослеђује и величину кернела који ће се примењивати при обради слике. Блок комуницира са *KERNEL BRAM*-ом одакле узима одговарајуће коефицијенте Гаусовог кернела потребне за обраду пиксела које преузима из *IMAGE BRAM* меморије. По завршетку обраде датог пиксела он се уписује у *DDR* меморију, путем *AXI-Stream* протокола.

IMAGE BRAM - Блок који представља *BRAM* меморију која складишти делове слике. Алгоритам за обраду једног пиксела узима у обзир и вредности пиксела из његовог суседства, неопходно је више пута у току рада приступити истим пикселима. Како је приступ *DDR* меморији временски "најскупљи", било је неопходно учитавати мање делове слике у меморију ближу процесуирајућем блоку (кеширање). На тај начин се добија уштеда на времену добављања податка из меморије, поготово ако се истом податку мора приступити више стотина пута као што је овде био сличај. IMAGE BRAM је реализован тако да се у њега учитавају делови слике по колонама, где број колона одговара димензији величине кернела

(овде су случаји 3 и 25). Након што се обради једна читава колона пиксела потребна је допуна кеша следећом колоном, путем *DMA* приступа *DDR* меморији. При томе, IPB блок је одговоран за израчунавање адреса са којих се чита *DDR*, као и адреса на које добијени подаци треба да буду уписани. Иницијално учитавање података у *IMAGE BRAM* такође се изводи путем *Axi Stream* интерфејса.

KERNEL BRAM - BRAM меморија која садржи коефицијенте Гаусовог кернела за одређене димензије кернела. Вредности се унапред израчунавају (унутар софтвера), где број коефицијената одговара броју поља матрице кернела (kernel_size * kernel_size). Иницијално учитавање коефицијената врши се посредством *BRAM* контролера, специјализованом елементу *AXI4* протокола за рад са типичним *BRAM* меморијама.

4. *ASM* дијаграм

На *слици 8* приказан је *ASM* (*Algorithmic State Machine*) дијаграм, добијен помоћу претходно приказаног кода са уклоњеним петљама.

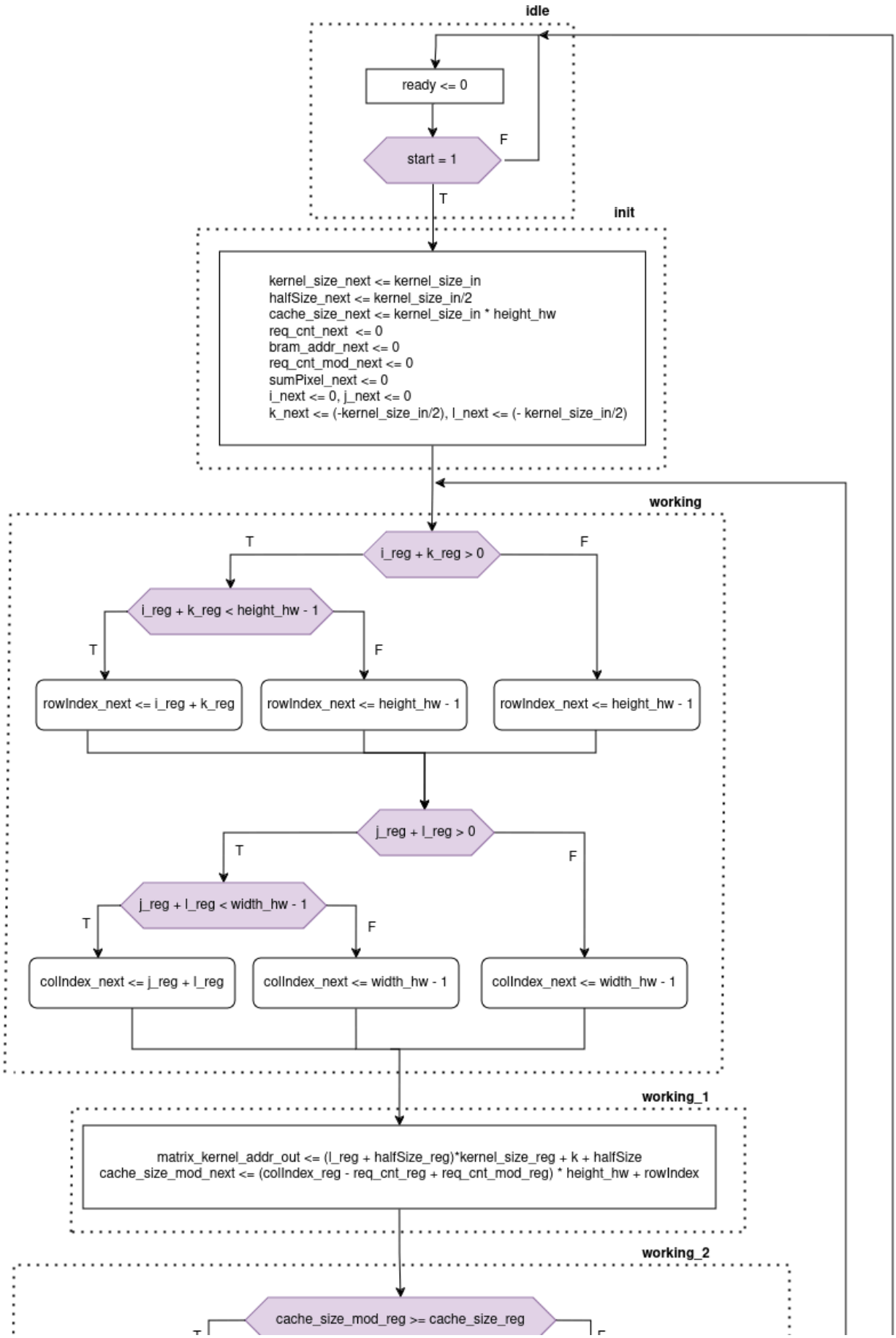
Уочено је да део кода са размотаним петљама са лабелом *cache_reg_count* може да се изврши раније, у делу кода са лабелом *working*, те је поменути део кода померен у *ASM* дијаграму. Такође, део кода у ком се проверава услов за допуну *Img Bram-a* је одвојен у посебно стање, које означава почетак допуне, како би се назначило да припада засебном режиму рада у крајњој имплементацији.

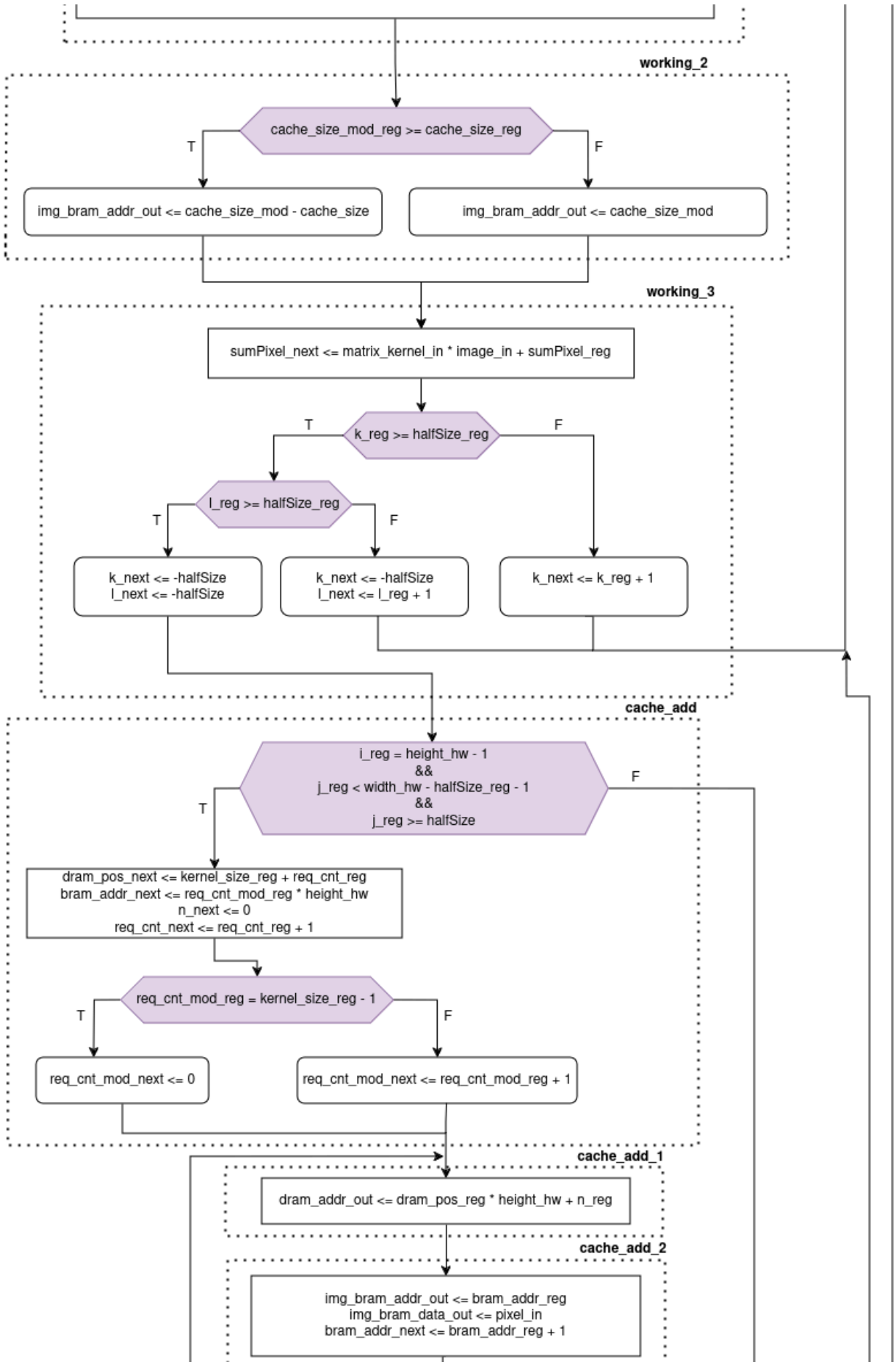
Приликом процене перформанси у оквиру предмета Пројектовање електронских уређаја на системском нивоу, уочено је да би директна имплементација, без проточне обраде захтевала чак 30 секунди за потпуну обраду једне фотографије димензија 720x1100. У случају постојања проточне обраде, потребно време за обраду једне фотографије смањује се на око 5,2 секунде. Из овог разлога, одлучено је да се имплементира проточна обрада.

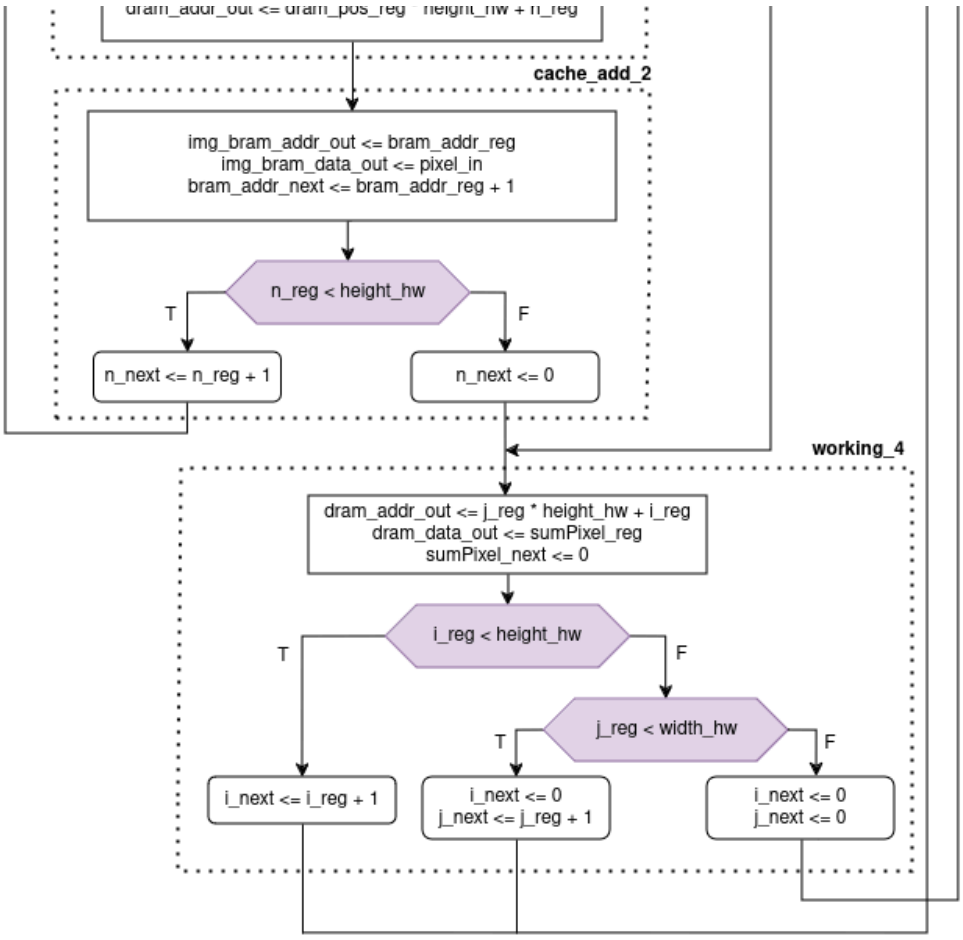
Стања обележена на *ASM* дијаграму су служила као водич за одређивање фаза проточне обраде. Она стања која носе назив working_x претворена су у фазе проточне обраде, а повећање и контрола бројача петљи издвојена је у посебну фазу проточне обраде. Такође, допуна *Img Bram-a* подацима из *DDR* меморије одвија се у посебном стању. Дакле, настао је нови *FSM,* са одговарајућим дијаграмом. у оквиру *control path-a* који контролише различите режиме рада система:

- *idle* - стање мировања које одговара idle стању оригиналног *ASM* дијаграма
- *init* - стање иницијализације које одговара init стању оригиналног *ASM* дијаграма
- *working* - стање уобичајеног рада, чини га рад у фазама проточне обраде, а фазе одговарају *working_x* стањима оригиналног дијаграма
- *cache_init* і *cache_add* - стања за допуну *Img Bram-a*, одговарају *cache_add_x* стањима оригиналног *ASM* дијаграма

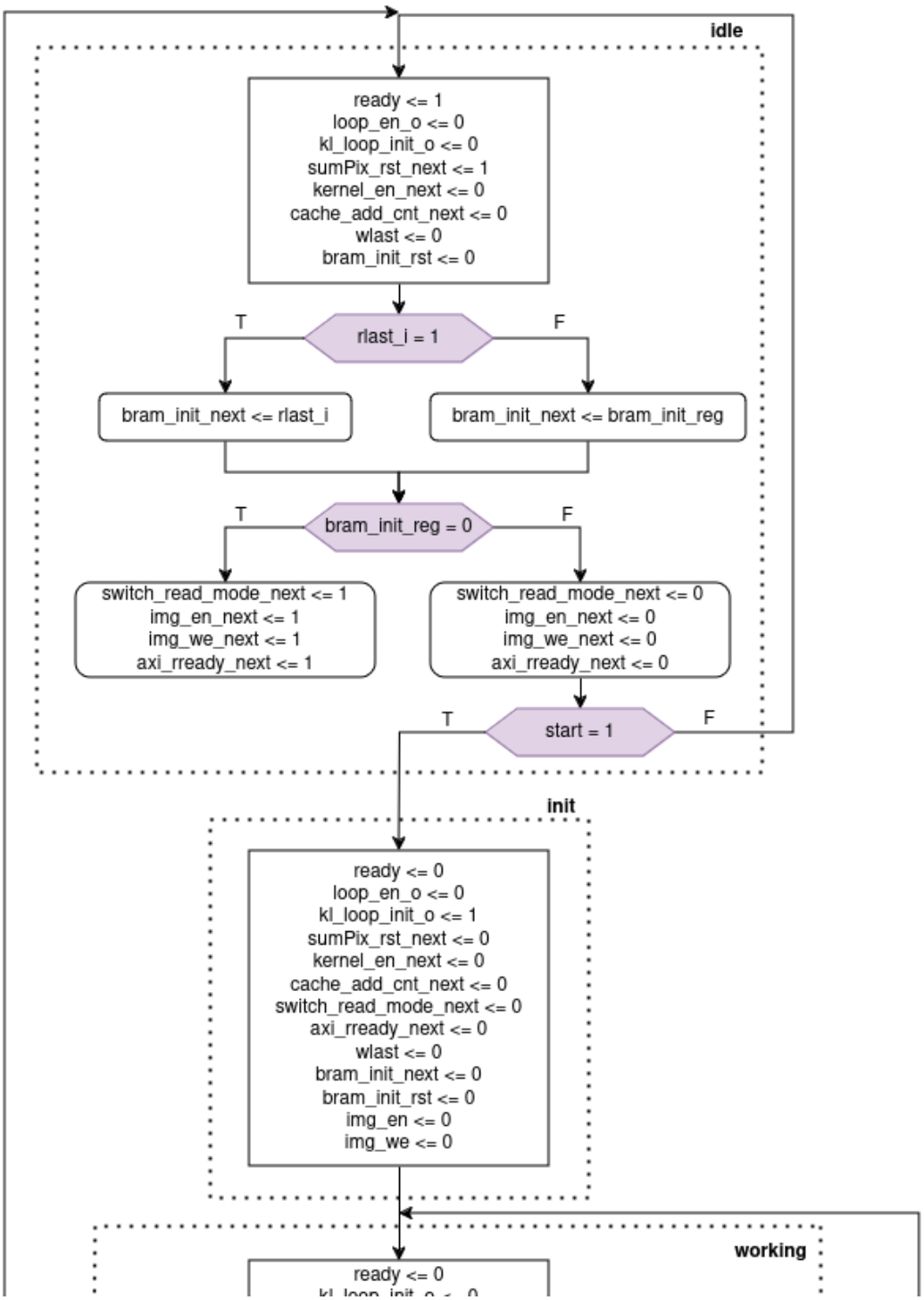
Овај изведени *ASM* дијаграм приказан је испод оригиналног дијаграма, на *слици 9*, а промењљиве које се у њему појављују представљају контролне и статусне сигнале, којима се управља радом система.

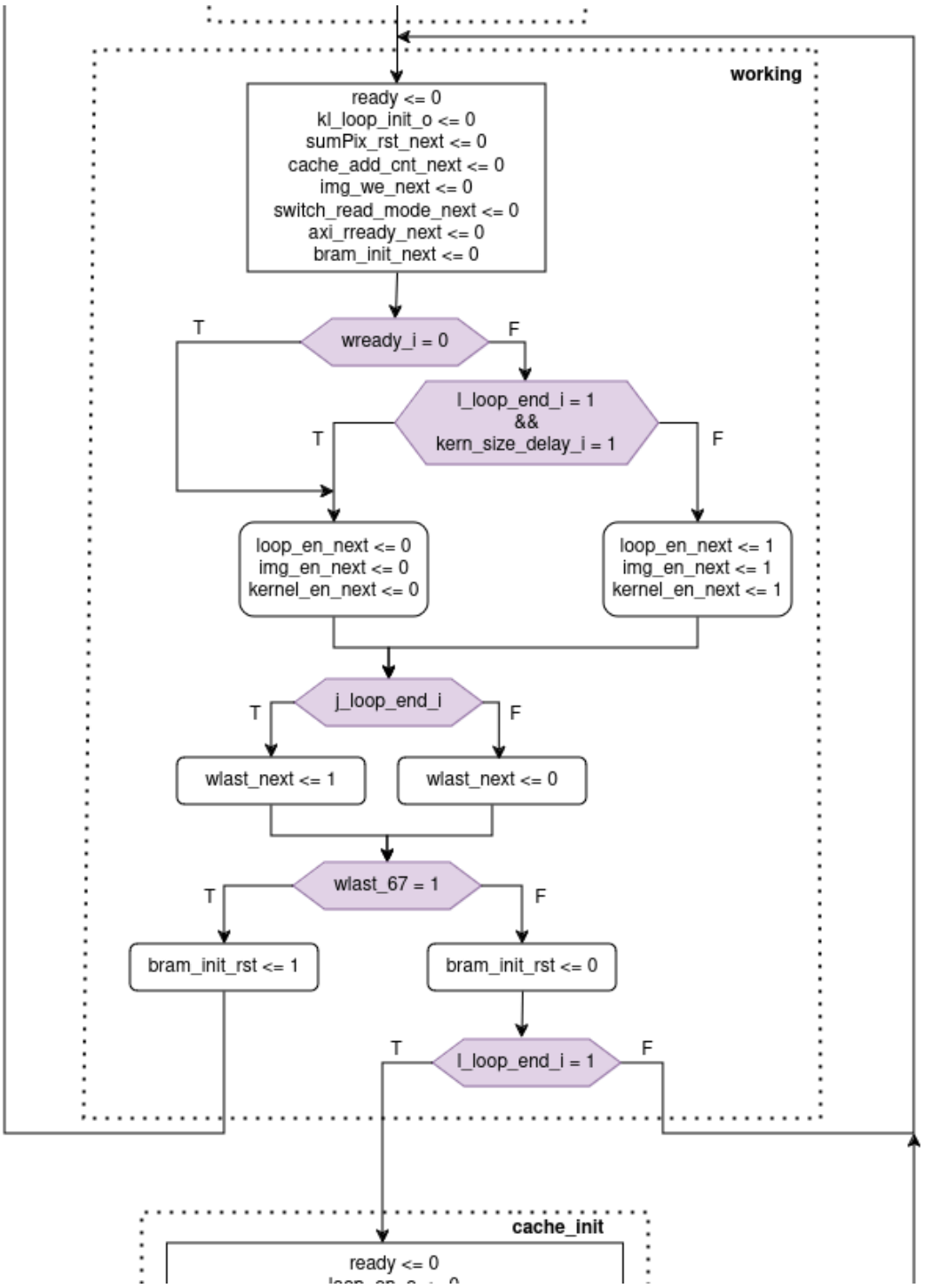

idle
ready <= 0
start = 1
F
T
init
kernel_size_next <= kernel_size_in
halfSize_next <= kernel_size_in/2
cache_size_next <= kernel_size_in * height_hw
req_cnt_next <= 0
bram_addr_next <= 0
req_cnt_mod_next <= 0
sumPixel_next <= 0
i_next <= 0, j_next <= 0
k_next <= (-kernel_size_in/2), l_next <= (- kernel_size_in/2)
working
i_reg + k_reg > 0
T
F
i_reg + k_reg < height_hw - 1
T
F
rowIndex_next <= i_reg + k_reg
rowIndex_next <= height_hw - 1
rowIndex_next <= height_hw - 1
j_reg + l_reg > 0
T
F
j_reg + l_reg < width_hw - 1
T
F
colIndex_next <= j_reg + l_reg
colIndex_next <= width_hw - 1
colIndex_next <= width_hw - 1
working_1
matrix_kernel_addr_out <= (l_reg + halfSize_reg)*kernel_size_reg + k + halfSize
cache_size_mod_next <= (colIndex_reg - req_cnt_reg + req_cnt_mod_reg) * height_hw + rowIndex
working_2
cache_size_mod_reg >= cache_size_reg

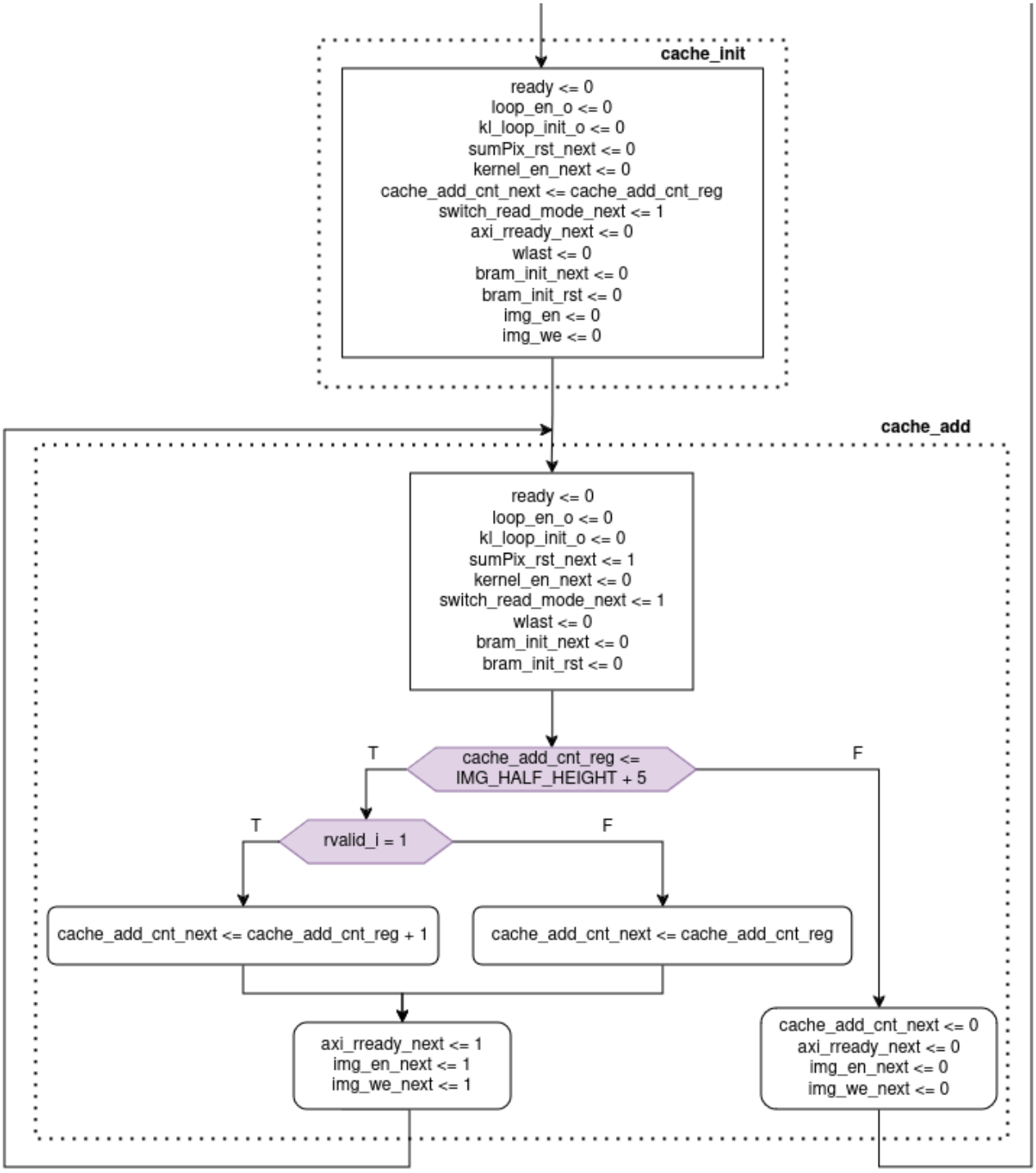

working_2
cache_size_mod_reg >= cache_size_reg
T
F
img_bram_addr_out <= cache_size_mod - cache_size
img_bram_addr_out <= cache_size_mod
working_3
sumPixel_next <= matrix_kernel_in * image_in + sumPixel_reg
T
k_reg >= halfSize_reg
F
l_reg >= halfSize_reg
T
F
k_next <= -halfSize
l_next <= -halfSize
k_next <= -halfSize
l_next <= l_reg + 1
k_next <= k_reg + 1
cache_add
i_reg = height_hw - 1
&&
j_reg < width_hw - halfSize_reg - 1
&&
j_reg >= halfSize
T
F
dram_pos_next <= kernel_size_reg + req_cnt_reg
bram_addr_next <= req_cnt_mod_reg * height_hw
n_next <= 0
req_cnt_next <= req_cnt_reg + 1
req_cnt_mod_reg = kernel_size_reg - 1
T
F
req_cnt_mod_next <= 0
req_cnt_mod_next <= req_cnt_mod_reg + 1
cache_add_1
dram_addr_out <= dram_pos_reg * height_hw + n_reg
cache_add_2
img_bram_addr_out <= bram_addr_reg
img_bram_data_out <= pixel_in
bram_addr_next <= bram_addr_reg + 1



Слика 8 - ASM дијаграм изведен из кода са размотаним петљама


idle
ready <= 1
loop_en_o <= 0
kl_loop_init_o <= 0
sumPix_rst_next <= 1
kernel_en_next <= 0
cache_add_cnt_next <= 0
wlast <= 0
bram_init_rst <= 0
T
rlast_i = 1
F
bram_init_next <= rlast_i
bram_init_next <= bram_init_reg
T
bram_init_reg = 0
F
switch_read_mode_next <= 1
img_en_next <= 1
img_we_next <= 1
axi_rready_next <= 1
switch_read_mode_next <= 0
img_en_next <= 0
img_we_next <= 0
axi_rready_next <= 0
T
start = 1
F
init
ready <= 0
loop_en_o <= 0
kl_loop_init_o <= 1
sumPix_rst_next <= 0
kernel_en_next <= 0
cache_add_cnt_next <= 0
switch_read_mode_next <= 0
axi_rready_next <= 0
wlast <= 0
bram_init_next <= 0
bram_init_rst <= 0
img_en <= 0
img_we <= 0
working
ready <= 0


working
ready <= 0
kl_loop_init_o <= 0
sumPix_rst_next <= 0
cache_add_cnt_next <= 0
img_we_next <= 0
switch_read_mode_next <= 0
axi_rready_next <= 0
bram_init_next <= 0
wready_i = 0
T
F
l_loop_end_i = 1
&&
kern_size_delay_i = 1
T
F
loop_en_next <= 0
img_en_next <= 0
kernel_en_next <= 0
loop_en_next <= 1
img_en_next <= 1
kernel_en_next <= 1
j_loop_end_i
T
F
wlast_next <= 1
wlast_next <= 0
wlast_67 = 1
T
F
bram_init_rst <= 1
bram_init_rst <= 0
T
l_loop_end_i = 1
F
cache_init
ready <= 0



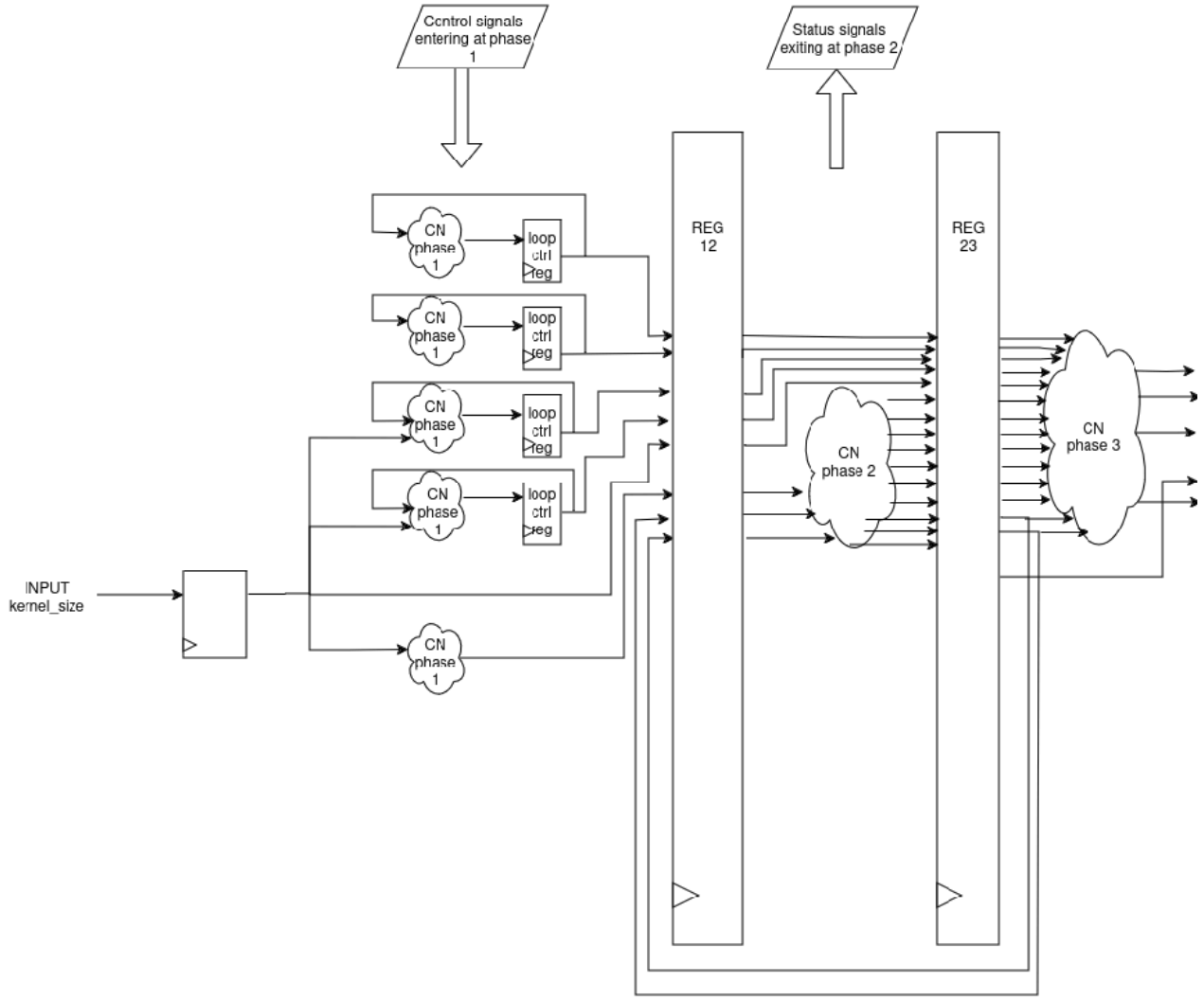
Слика 9 - ASM дијаграм који описује рад контролне јединице у controlpath-у

5. Блок дијаграм - Controlpath, Datapath

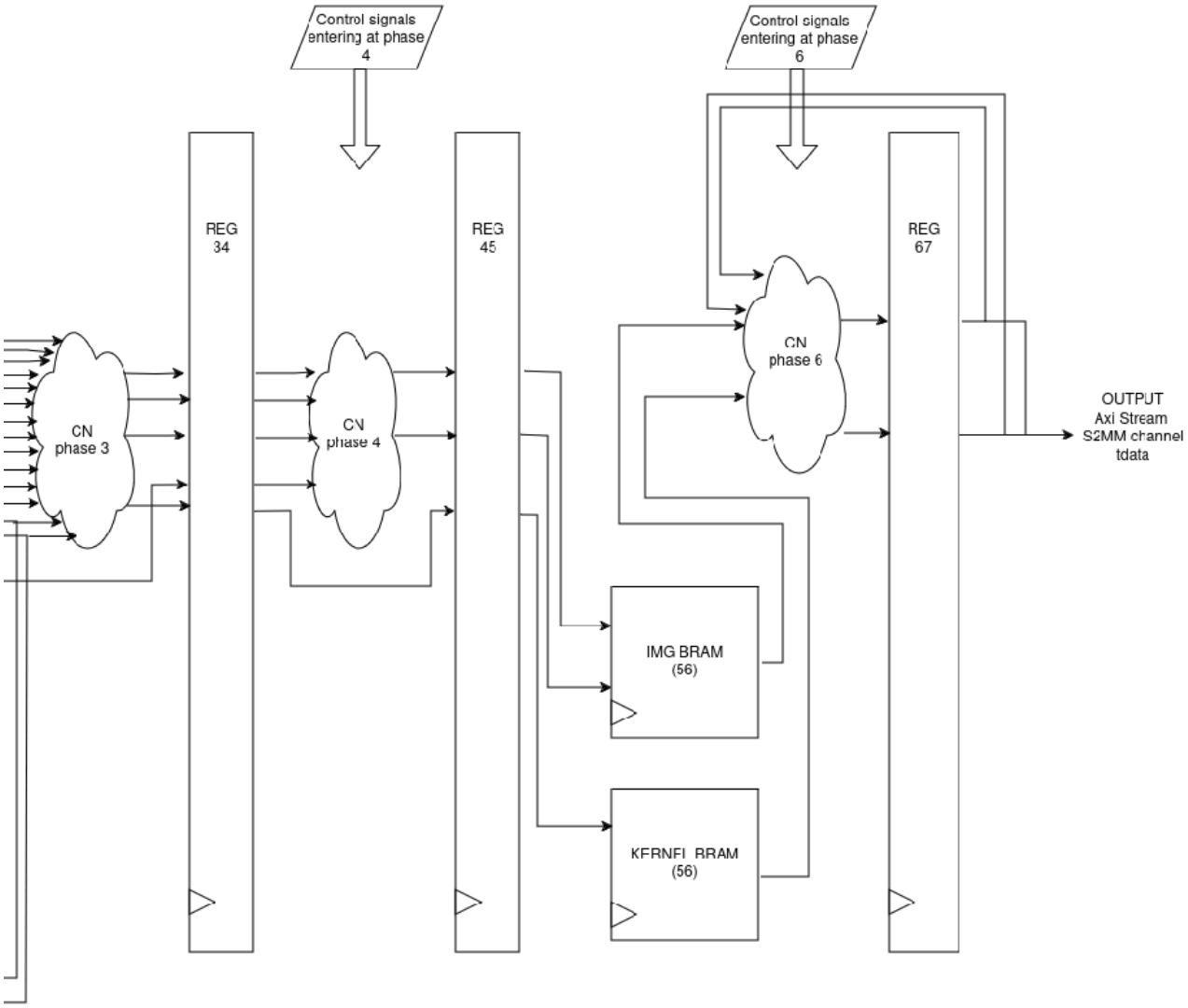
Datapath чине улазни регистар, који прима величину кернела, излазни регистар (*REG 67*), који даје вредност sumPixel, односно обрађени пиксел за сваку итерацију петљи, променљиве, и остали регистри који раздвајају фазе проточне обраде. Прва фаза проточне обраде контролише контролне променљиве петљи: *i, j , k, l*. У фази 5 се врши приступ *Img Bram*-у и *Kernel Bram*-у, а у фази 6 се добијају жељене вредности из *Img* и *Kernel Bram*-ова.

Извршена је и оптимизација у виду размотавања петље і фактором k = 2. Услед тога, добијају се две излазне вредности REG 67, које се затим конкатенирају. На овај начин је постигнуто и да се истовремено шаље више резултата ка *DDR*-у, односно два пиксела истовремено путју 32-битним каналом за податке.

Такође *datapath* и *controlpath* комуницирају путем контролних и статусних сигнала. *Datapath* шаље статусне сгинале из одговарајућих фаза обраде *controlpath*-у, који затим генерише потребне контролне сигнале и добравља их у одговарајућу фазу *datapath*-а.

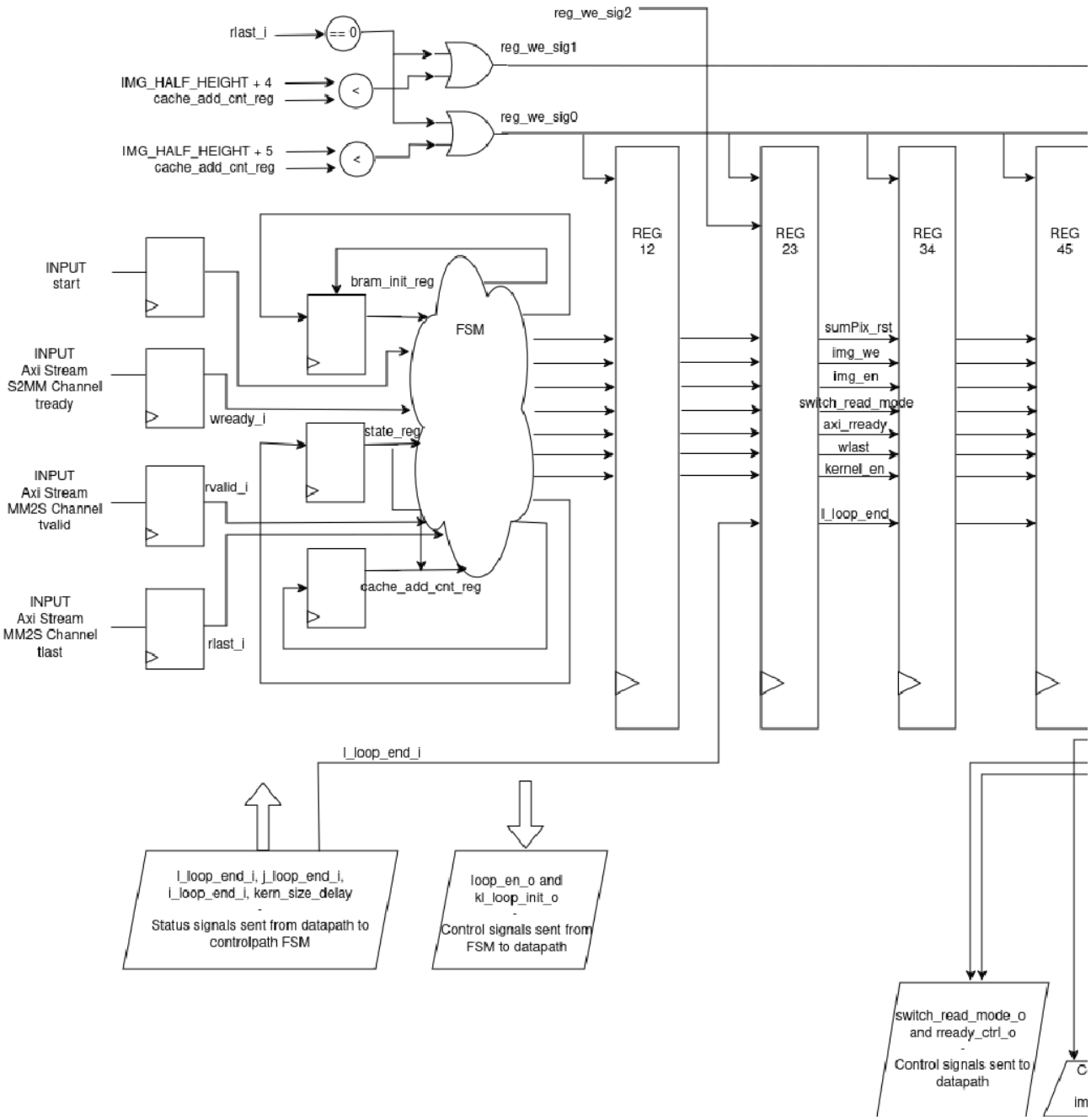


Слика 10 - Datapath, први део

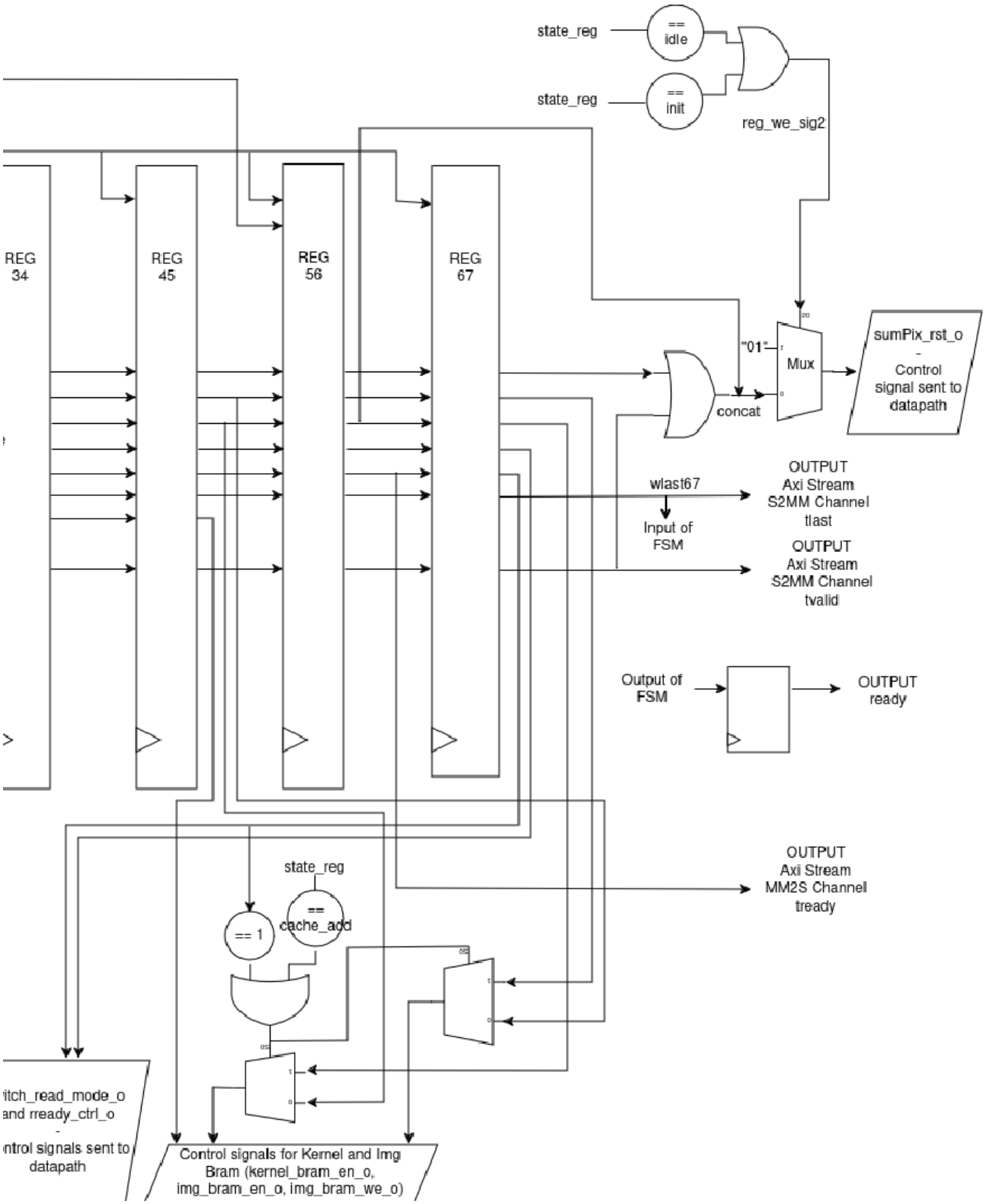


Слика 11 - Datapath, други део

Controlpath се састоји из улазних регистара који примају *Axi Stream* сигнале, регистра за чување тренутног стања у коме се систем налази (*state_reg*), регистра који чува информацију о томе да ли је *Img Bram* иницијализован (*bram_init_reg*) и регистра који служи као бројач при допуни *Img Bram-a* (*cache_add_cnt_reg*). Осим тога, *controlpath* садржи и *FSM*, који на основу тренутног стања система, улаза у систем и статусних сигнала из *datapath-a* генерише одговарајуће контролне сигнале. Како би се ови контролни сигнали у одговарајућој фази допремили до *datapath-a*, *contropath* такође садржи регистре проточне обраде. *REG 67* служи и као излазни регистар система.



Слика 12 - Controlpath, први део



Слика 13 - Controlpath, други део

6. Анализа моделованог система (пре паковања у *IP* језгро)

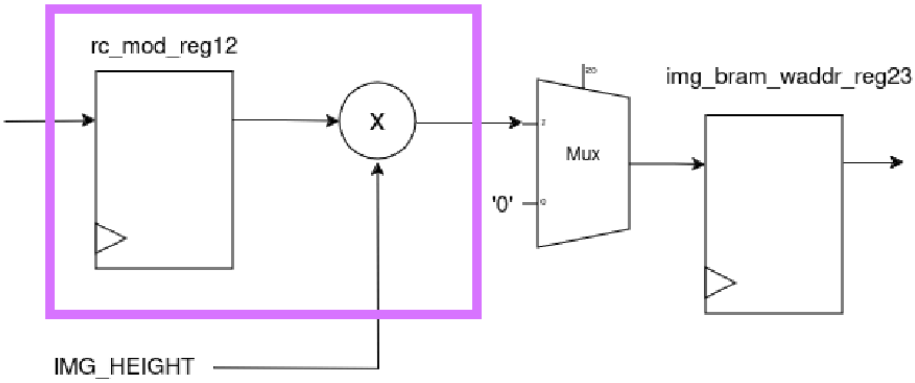
6. 1 Утрошени ресурси

Приказана је табела утрошених ресурса након имплементације *apply_gaussian_top-a:*

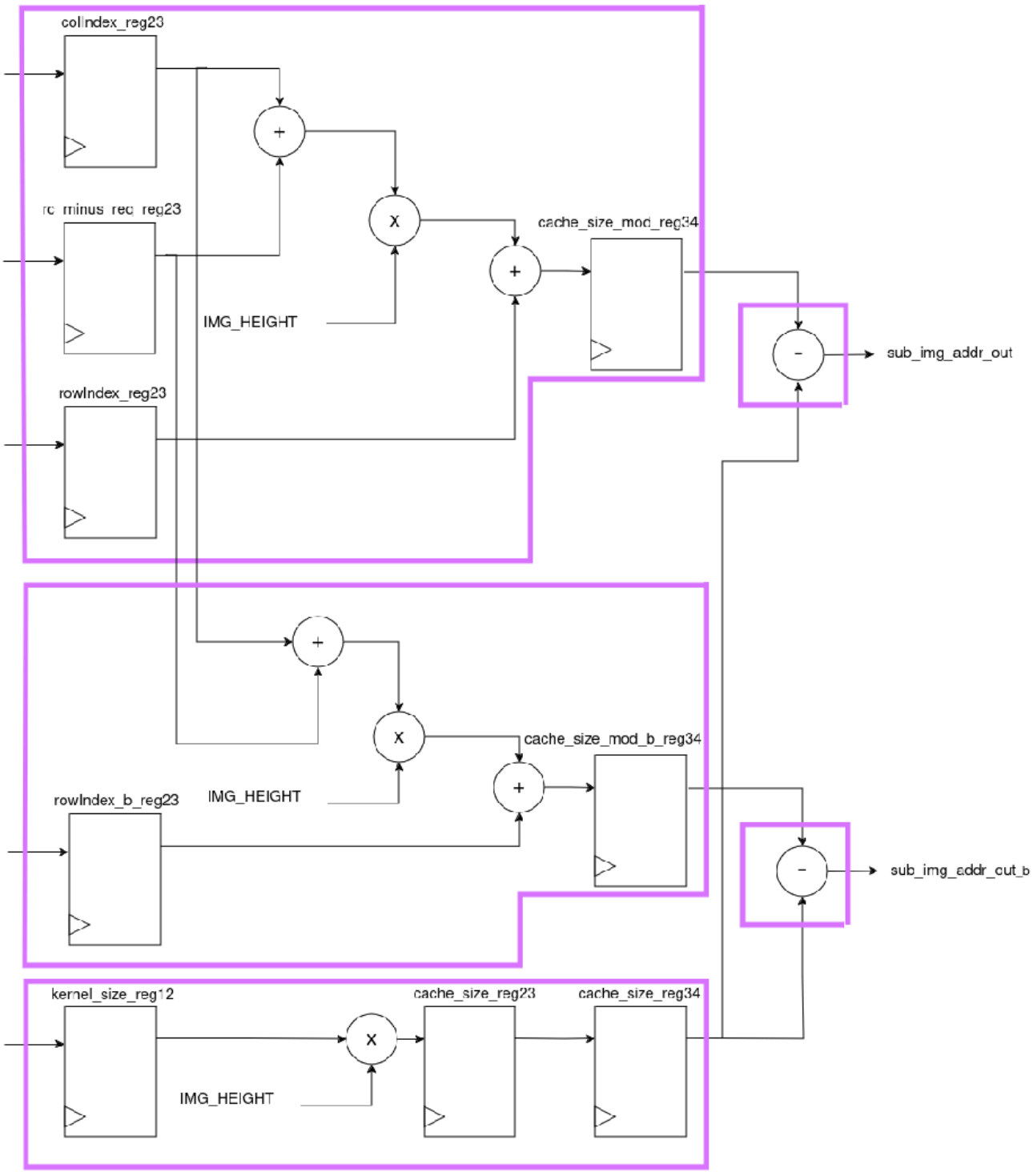
Resource	Utilization	Available	Utilization %
LUT	325	17600	1.85
LUTRAM	8	6000	0.13
FF	272	35200	0.77
BRAM	8.50	60	14.17
DSP	8	80	10.00
IO	94	100	94.00
BUFG	1	32	3.13

Слика 14 - табела утрошених ресурса

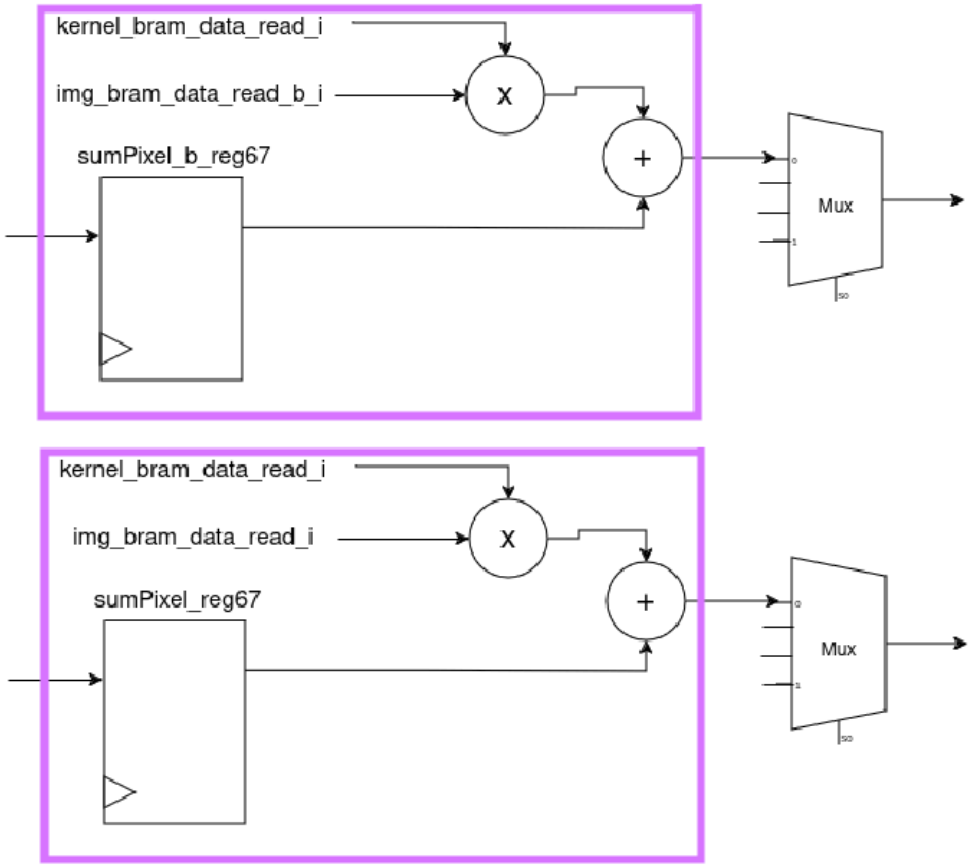
Искоришћено је осам *DSP* јединица, а све од њих се налазе у *datapath*-у. На *сликама 15, 16 и 17* су приказани и уоквирени делови система који су мапирани на *DSP* блокове:



Слика 15 - део комбинационе мреже из фазе 2 проточне обраде, који се мапира на DSP



Слика 16 - део комбинационе мреже из фазе 3 проточне обраде, који се мапира на више DSP јединица



Слика 17 - делови комбинационе мреже из фазе 6 проточне обраде, који се мапирају на DSP

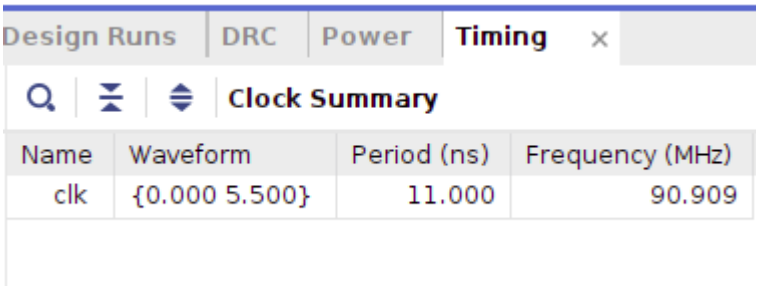
Меморија за фотографију и меморија за вредности Гаусовог кернела се мапирају на *BRAM* јединице:

функционална јединица	Kernel Bram	Img Bram
број BRAM-ова	0.5	8

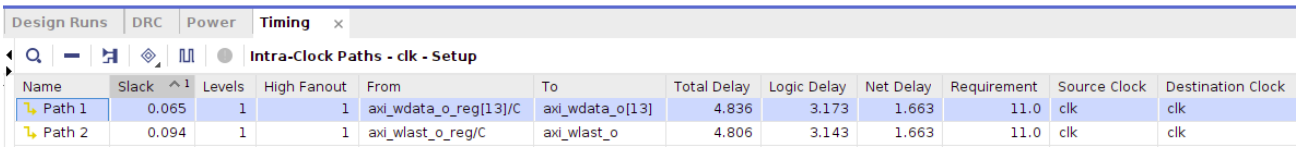
Табела 1 - Преглед искоришћених BRAM јединица

6. 2 Критична путања и максимална фреквенција

На нивоу дизајна пре креирања *IP*-а дизајн може да ради на фреквенцији 90.909 MHz.



Слика 18 - Максимална фреквенција рада система



Слика 19 - Критична путања система

6. 3 Пропусна моћ (throughput) и кашњење (latency)

Коришћењем формула за кашњење и пропусну моћ добијене су вредности ова два параметра за наш систем. Због несавршености дизајна и имплементације проточне обраде, поменуте формуле су модификоване.

Кашњење нашег система приближно је кашњењу проточне обраде унутар њега. Чине га *clock_to_q* кашњење, *setup* време и максимално кашњење међу кашњењима комбинационих кола фаза проточне обраде *Tmax*. Периода клока мора бити бар оноликог трајања да се ту могу сместити сва три поменута времена, па је кашњење проточне обраде једнако производу броја фаза *n* (у нашем случају 6) и периоде такт сигнала Tc:

$$Tsystem \approx \mathrm{T}pipe = 6Tc = 6Tmax + 6(Tsetup + Tcq) = 6 * 11\ ns = 66\ ns \quad формула(1)$$

Пропусна моћ система је приближна пропусној моћи проточне обраде. Међутим, уобичајено је да проточна обрада има иницијално кашњење од *n-1* тактова, док се не попуне све фазе и не произведе први резултат. Због несавршености дизајна, у нашем случају, ово кашњење се јавља при сваком преласку из стања *working* у стања за допуну *Img BRAM*-а (*cache_init*).

$$throughput \approx \frac{k}{2(img\ width - 2{*}halfsize)(n-1)Tc + kTc} \qquad формула(2)$$

У *формули 2*, *k* представља број узастопних података који се обрађују, у нашем случају то су пиксели и свака итерација *k* и *l* петље за одговарајући пиксел. *2(img width - 2*halfsize)* је број пута колико се дешава прелазак из *working* стања у стања за допуну *BRAM-a* и обрнуто. А променљива *halfsize* јесте целобројни резултат дељења димензије кернела са два, након одбацивања децималног дела. *kTc* представља устаљен рад проточне обраде, када се нека резултантна вредност генерише.

У конкретном примеру за фотографију висине 720 пиксела и ширине 1100 пиксела, коју је потребно обрадити применом кернела димензије 3, па затим димензије 25, *k* је једнако *(image height * image width)*(kernel size * kernel_size) / 2*. Дељење двојком на крају се врши јер је наш систем у могућности да паралелно обрађује два пиксела, захваљујући размотавању петље.

За случај са димензијом кернела 3 добија се да једна слика може бити обрађена за 39,325 *ms*, а за случај са димензијом кернела 25 обрада једне слике је готова након 2,7226 секунди. Обе поменуте величине кернела се примењују како бисмо добили фотографију која је спремна за даљу обраду у циљу проналаска слободних паркинг места на фотографији. Дакле, укупно време потребно за комплетну обраду једне фотографије јесте 2,762 секунде.

7. Паковање *IP* језгра

Да би се омогућила комуникација са процесором, успостављен је *Axi Lite* протокол који служи за упис и читање одговарајућих регистара:

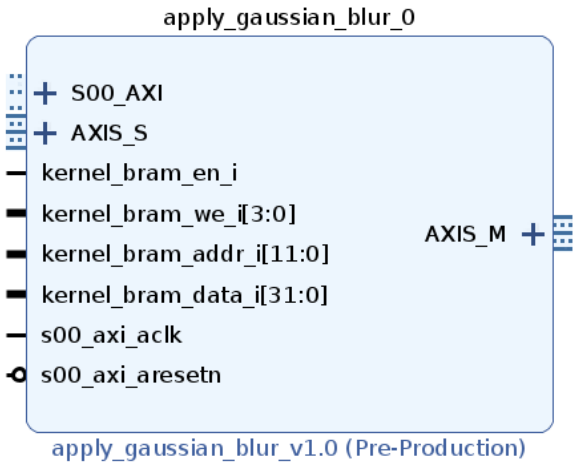
РЕГИСТАР	АДРЕСА
start - slv_reg0	0
kernel_size - slv_reg1	4
restart - slv_reg2	8
ready - slv_reg3	12

Табела 2 - Преглед Axi Lite регистара

Портови за *Kernel BRAM* остали су слободни и они ће касније бити повезани на порт *Axi Bram Controller*-а.

Наш *IP* садржи и два порта за *Axi Stream* протокол, *AXIS_S*, преко ког *IP* прима податке од *DMA* (*MM2S* канал) и *AXIS_M*, преко ког *IP* уписује резултате у *DMA* (*S2MM* канал).

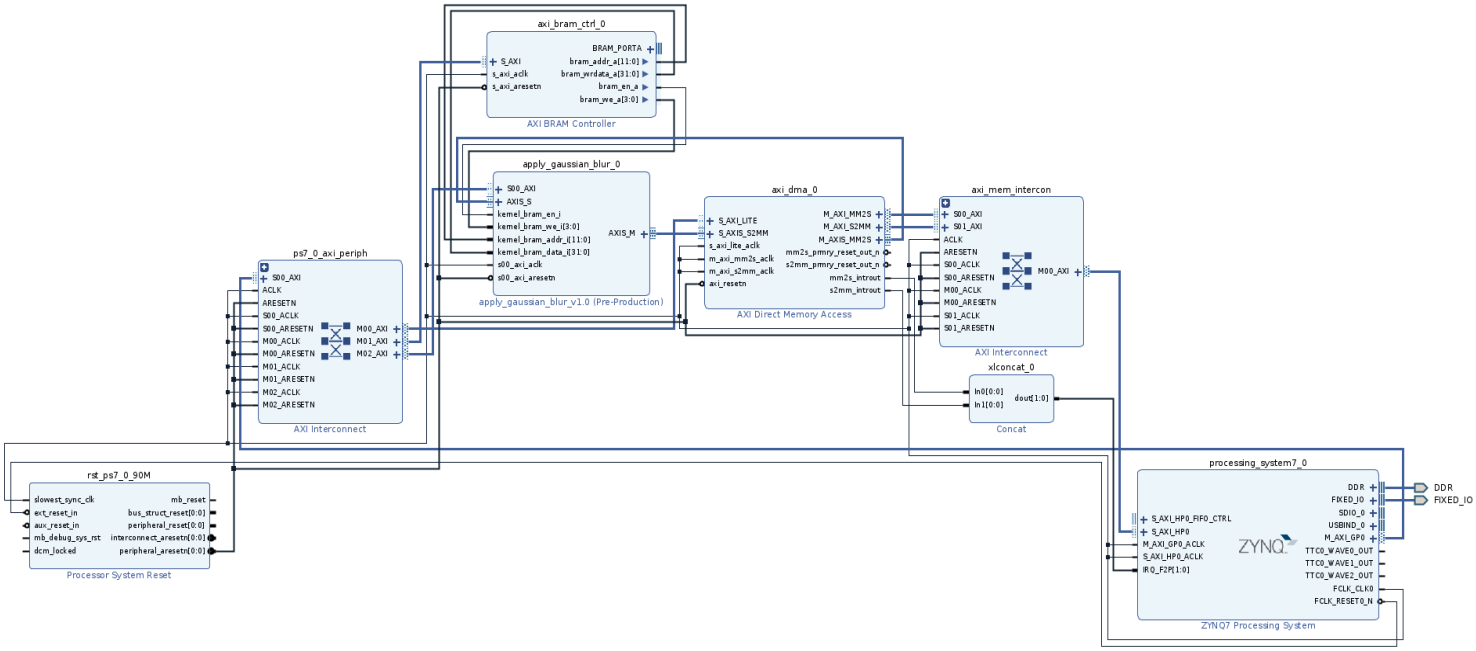
Такође, *IP* има могућност подешавања жељених димензија фотографије.



Слика 20 - Изглед apply_gaussian_blur IP-a

8. Анализа интегрисаног система након синтезе и имплементације

8. 1 Изглед интегрисаног система



Слика 21 - Блок схема система након повезивања унутар алата Vivado IP Integrator

Околне компоненте са којима се повезује *IP* језгро су: *Zynq 7 Processing System, AXI DMA (Direct Memory Access)* и *Axi Bram Controller*.

Прво је постављен *Zynq 7 Processing System*, а затим извршена следећа подешавања: омогућен је рад различитих периферија као што су *SPI0, I2C0, UART1, GPIO, SD0, USB0, ENET0* і *QUAD SPI*, повезан је *DDR* интерфејс, постављена су два пина за прекиде, како би DMA компонента могла да обавести процесор о завршетку рада, омогућен је један *HP Slave Axi* интерфејс, који служи за рад са *DMA,* и подешена је фреквенција рада.

Затим је постављен *DMA*, конфигурисан је да ради у *Direct Mode*, са параметром *Max Burst Size* од 256 и ширином *Buffer Length* регистра од 22 бита.

Након тога постављен је *AXI BRAM Controller* који има један *BRAM* интерфејс и подешен је *Memory Depth* параметар.

На крају је додат *IP apply_gaussian_blur_v1.0* и повезан са претходно поменутим компонентама.

8. 2 Утрошени ресурси

Resource	Utilization	Available	Utilization %
LUT	4136	17600	23.50
LUTRAM	306	6000	5.10
FF	5145	35200	14.62
BRAM	12.50	60	20.83
DSP	8	80	10.00
BUFG	1	32	3.13

Слика 22 - Утрошени ресурси након имплементације интегрисаног система

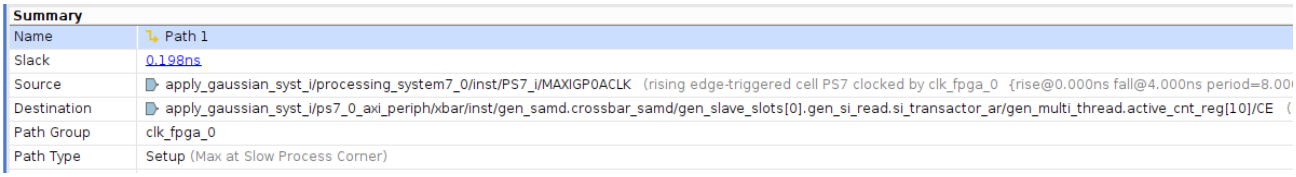
8. 3 Критична путања и максимална фреквенција

Након извршене синтезе и имплементације показало се да систем може да ради на фреквенцији од 125 MHz.

Clock Summary

Name	Waveform	Period (ns)	Frequency (MHz)
clk_fpga_0	{0.000 4.000}	8.000	125.000

Слика 23 - Максимална фреквенција интегрисаног система



Слика 24 - Критична путања интегрисаног система

8. 4 Пропусна моћ (throughput) и кашњење (latancy)

Уз помоћ *формуле 1* добије се кашњење интегрисаног система које је приближно једнако:

$6 * Tc = 6 * 8\ ns = 48\ ns.$

За потпуну обраду једне слике, као у примеру из одељка *6.3,* потребно је 2 секунде. Заправо, ово време је у стварности нешто веће, јер нису урачуната кашњења додатних регистара у меморијском подсистему.

9. Поређење добијене фреквенције рада система и *throughput-a* са резултатима добијеним на предмету “Пројектовање електронских уређаја на системском нивоу”

На предмету “Пројектовање електронских уређаја на системском нивоу” процењена фреквенција рада јесте 100 MHz. Она је добијена одабиром дела кода са највише аритметичких операција, односно интуитивном проценом критичне путање. Само овај део је моделован и над њим су извршене синтеза и имплементација, како би се добила процена фреквенције рада.

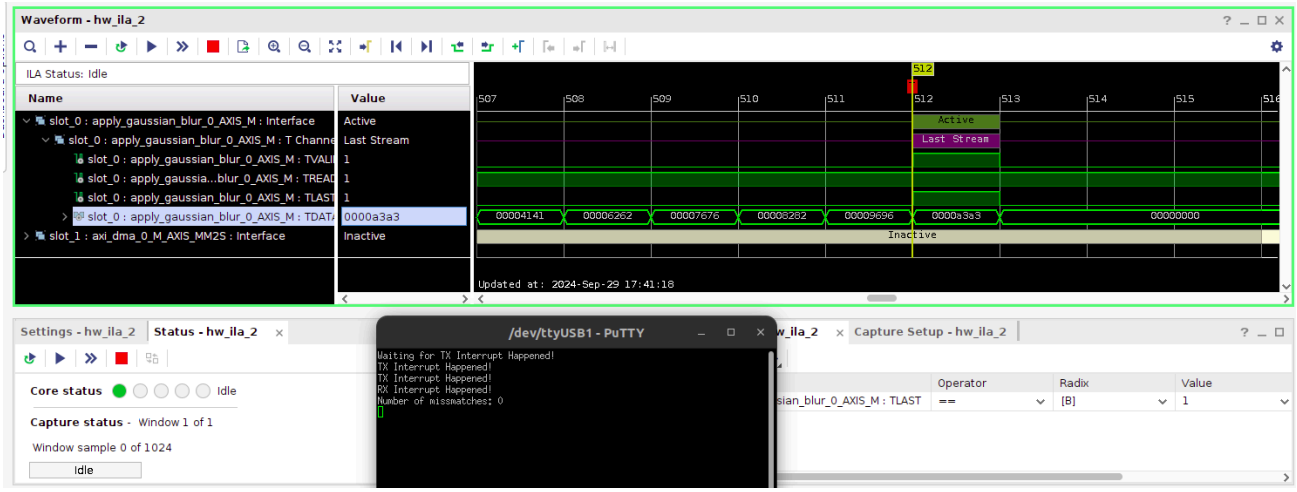
Након моделовања језгра које врши потребну фунцкионалност, *apply_gaussian_top*, добијена је фреквенција 90,9 MHz. Након интегрисања читавог система, алат за синтезу и имплементацију успева да изврши оптимизације и коначна фреквенција рада јесте 125 MHz.

Осим тога, на предмету “Пројектовање електронских уређаја на системском нивоу” процењено време потребно за потпуну обраду једне фотографије је 2,591 секунди. Након моделовања језгра које врши потребну фунцкионалност, ово време било је 2,762 секунде, а након интегрисања целокупног система време обраде једне фотографије, захваљујући повећању фреквенције рада, пада на 2 секунде.

10. Тестирање рада у *Vitis*-у

Након синтезе и имплементације, генерисан је *bitstream*, и систем је спуштен на *Zybo* плочицу.

Слика 24 приказује исходе тестирања система *bare metal* апликацијом. Променљивом *missmatches* праћен је број неслагања у добијеним и очекиваним резултатима. Такође, на *слици 24* се види и изглед *ILA* прозора, коришћен за дебаговање и проверу рада. Конкретно, овде се види да је сигнал *tlast S2MM Axi Stream* канала *DMA* на јединици, односно да је потребан број резултата успешно примљен.



Слика 24 - ILA прозор и PuTTY прозор са информацијама о броју неслагања очекиваних и добијених података

11. Литература

[1] Материјали са предавања и вежби предмета Пројектовање сложених дигиталних система: https://www.elektronika.ftn.uns.ac.rs/projektovanje-slozenih-digitalnih-sistema/specifikacija/specifikacija-predmeta/ (4.9.2024.)

[2] Pong P. Chu, *RTL Hardware Design using VHDL*, Wiley-Interscience, 2006

[3] *AMBA AXI-Stream Protocol Specification*, ARM, 2021:
https://developer.arm.com/documentation/ihi0051/latest/ (16.9.2024.)

[4] Документација FPGA плочице zybo-z7 серије:
https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual (1.9.2024.)

[5] *AXI BRAM Controller v4.1- LogiCORE IP Product Guide,* AMD, 2019:
https://docs.amd.com/v/u/en-US/pg078-axi-bram-ctrl (18.9.2024.)

[6] *AXI DMA LogiCORE IP Product Guide (PG021) (v7.1),* AMD, 2024:
https://docs.amd.com/r/en-US/pg021_axi_dma/Introduction (20.9.2024.)

[7] *AMBA AXI and ACE Protocol Specification,* AMD, 2011:
http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf (15.9.2024.)