# Пројекат: Детектовање правих линија на фотографији и њихово исцртавање

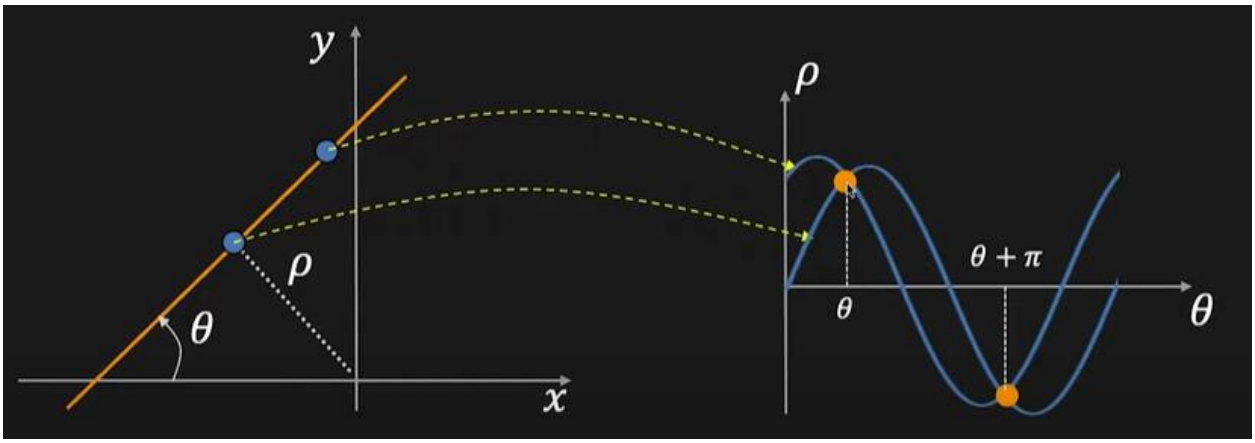
**Документација за предмет:**
**Микрорачунарски системи за рад у реалном времену**

**ТИМ „y23-g00“**
Петар Стаменковић EE18/2019
Момир Царевић EE3/2019
Дејан Пејић EE75/2019

# Одељак 1: Увод

Тема нашег пројекта јесте детектовање правих линија (праваца) на фотографији која се прослеђује програму, те њихово исцртавање. Централни део алгоритма јесте „Hough“ трансформација, која представља једну линеарну трансформацију. Принцип трансформације јесте следећи: Свака тачка (пиксел) из једног домена (наша фотографија) се пресликава у праву у другом домену. Информације о тим правама се складиште унутар акумулатора, који је представља матрицу, у којој је једна димензија коефицијент правца, а друг пресјек са у-осом трансформисане праве. Међутим, ту се наилази на проблем, јер коефицијент правца може да се налази у интервалу од – бесконачно до + бесконачно, те се зато примењује другачији облик алгоритма. У овом алгоритму се користи тригонометријски облик праве, где је она одређена са растојањем од координатног почетка (ρ) и углом под којим сече x-оцу (Θ). Зато се у овом случају добија акумулатор, у којем је једна димензија додељена за тета, а друга за ро. У овом случају почетног проблема нема, јер је угао ограничен (0 до 359). Заправо, (за сада) довољно је да се посматрају само углови од 0 до 179, јер се праве са угловима Θ и Θ + 180 поклапају. Иницијална вредност свих поља унутар акумулатора је нула, а током рада алгоритма он се испуњава информацијама о трансформисаним линијама. На крају трансформације унутар акумулатора ће се наћи велике вредности у пољима са информацијама о правим уочљивим на оригиналној фотографији.

За низ углова Θ добијају се одговарајуће вредности ρ. Ипоставља се да низ ових вредности даје синусоиду (као на слици 1). За две тачке ћемо добити две синусоиде, а пресек тих синусоида даје ρ и Θ праве која пролази кроз те две тачке.




**Слика 1.** Hough трансформација

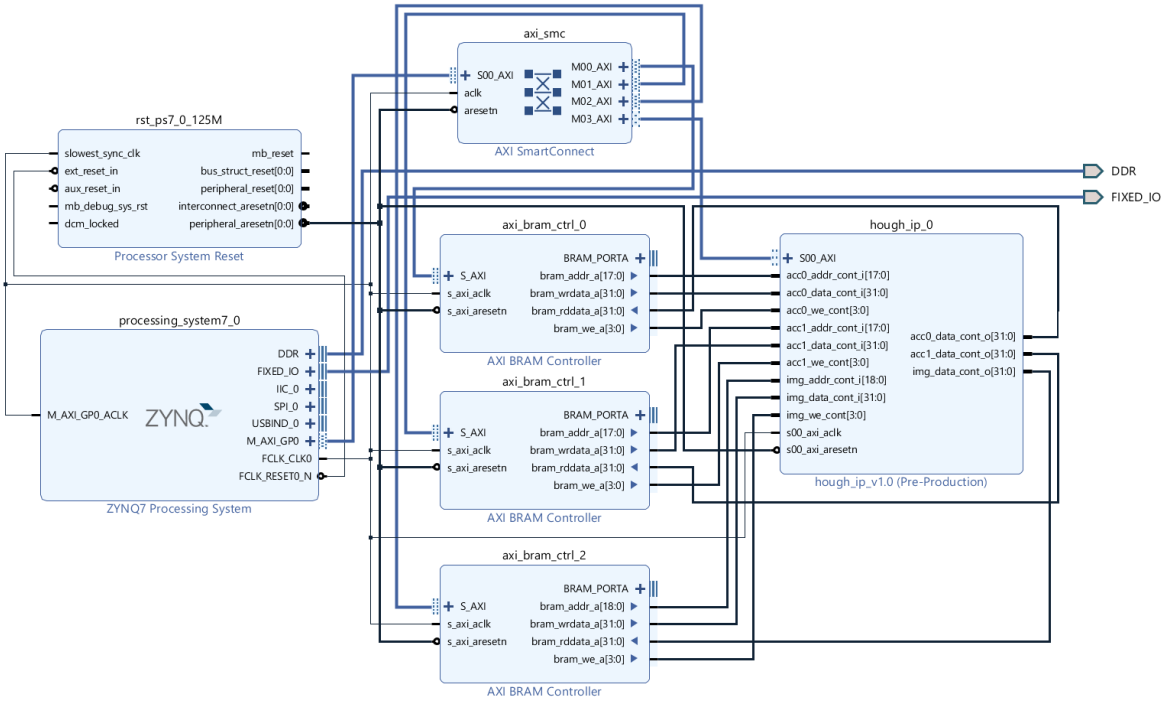
Изводи се де формула за рачунање ρ : ρ(Θ) = x*sin(Θ) - y*cos(Θ). Међутим, по овој формули када права сече x осу у тачки где је x < 0, добија се негативно ρ. Пошто је ρ растојање (за које се подразумева да је позитивно), могла би се наћи апсолутна вредност од ρ те увећати одговарајуће поље акумулатора. Међутим, треба обратити пажњу да постоји права која x-осу сече под истим углом за случај x > 0, те бисмо изгубили информацију о којој се заправо правој ради.

Зато се за ρ < 0 узима његова апсолутна вриједност, али се и вредност Θ повећава за 180 (што има смисла, јер је то и даље иста права) те се проблем превазилази. Ситуацију ρ < 0 је могуће добити само за Θ < 90 (права сече x осу у тачки где је x < 0 ,права са тупим улом

у том случају не би пролазила кроз први квадрант, који заправо и репрезентује фотографију, тако да та права није од интереса ). Зато се у акумулатору резервише простор за Θ од 0 до 269 (уместо првобитних 179).

На предмету Пројектовање сложених система уведена је оптимизација loop unrolling са фактором два, те је у складу с тим и акумулатор из алгоритма подељен на два двострука мања дела.

Коначан изглед система је представљен на слици 1.




**Слика 2.** Блок дизајн цијелог система

За овај пројекат је искоришћена *AXI4-Lite* магистрала за упис и читање одговарајућих регистара, који се налазе у меморијском подсистему магистрале. Меморијски подсистем јесте интерфејс између *AXI4-Lite* улазно-излазног контролера и модула *hough_core*. Овим интерфејсом се врши трансфер података потребних за извршавање трансформације (димензије фотографије), статусни и командни сигнали. Регистри за упис и читање су: *start*, *reset*, *width*, *height*, *rho* и *threshold*, док се из *ready* регистра може само читати. С друге стране, портови намењени за унутрашње меморијске јединице језгра су слободни, јер је предвиђено да се оне попуњавају путем *AXI-BRAM* контролера.

## Одељак 2: Развој драјвера

Прво се описује равзој драјвера за систем са слике 5.10. Као што се види, систем треба да има приступ *BRAM* контролерима и самом *IP* језгру. Целокупан драјверски модул се састоји из *platform_driver* структуре и неколико функција, које су описане у наставку.

## 2.1: Структура *platform_driver*

Улога ове структуре јесте иницијализација хардверског система. Унтар ње се налазе још једна структура *driver* и показивача на функције *hough_probe* и *hough_remove*, које су описане у једној од наредних секција. Структура *driver* има три поља:

1. *name* је стринг, који представља име драјвера. Њему се прослеђује макро *DRIVER_NAME* са вредношћу *"hough_driver"*.
2. *owner* представља власника драјвера и најчешће се поставља на марко *THIS_MODULE*, што је случај и у овом пројекту.
3. *of_match_table* је поље које показује на низ-структуру *of_device_id*, са називом *hough_of_match*.

```
static struct platform_driver my_driver = {
    .driver =
    {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = hough_of_match,
    },
    .probe = hough_probe,
    .remove = hough_remove,
};
```

```
static struct of_device_id hough_of_match[] =
{
    { .compatible = "acc0_bram_ctrl", },
    { .compatible = "acc1_bram_ctrl", },
    { .compatible = "img_bram_ctrl", },
    { .compatible = "hough_core", },
    { /* end of list */ },
};
```

**Слика 3.** Структуре *platform_driver* и *of_device_id*

Низ-структура *of_device_id* садржи compatible поља у којима се наводиме имена свих уређаја за које је намењен овај драјвер. Та имена су уписана тако да се подударају са називима у стаблу уређаја, на основу чега кернел може да утврди физичке адресе уређаја и повеже их са драјвером. Приликом креирања стабла уређаја, модули од интереса су названи „*acc0_bram_ctrl*“, „*acc1_bram_ctrl*“, „*img_bram_ctrl*“ и „*hough_core*“, па су та имена наведена и унутар *compatible* поља.

## 2.2: Функције *hough_init* и *hough_exit*

Функција *hough_init* служи за динамичко повезивање модула са кернелом у време. Она се позива помоћу команде *insmod* и њом почиње постојање модула у кернелу. Током њеног извршавања, она обавља неколико ствари. Пре свега се динамички алоцирају управљачки бројеви помоћу функције *alloc_chrdev_region*. С обзиром на то да се у систему налазе четири модула, овом функцијом се алоцира и исти број управљачких бројева. Након тога, креирају се „*node*“ фајлови у /dev директоријуму за сваки од четири модула из *compatible* поља. То се извршава помоћу две функције *class_create* и *device_create* – где је прва од њих неопходна јер се њена повратна вреднос користи као први параметар у другој функцији. Ако се пронађе одговарајућа вредности из *compatible* поља у стаблу уређаја, након *hough_init* функције се позива и *hough_probe* функција.

Помоћу *cdev* структуре функција *hough_init* повезује креиране „*node*“ фајлове са функционалностима које треба да се изврше услед позивања неке функције за рад са фајловима. На пример, уколико се нешто покуша уписати у фајл */dev/hough_core*, кернел ће позвати функцију на коју показује поље „*write*“ у структури *file_operations*. Уколико се само једна од наведених наредби не изврши успешно, поништавају се и све претходно извршене наредбе.

С друге стране, улога функција *hough_remove* све супротно у односу на *hough_init*. Помоћу ње се модул динамички искључује из кернела, тако што се брише структура cdev, бришу се сви „node“ фајлови (помоћу функција *device_destroy и class_destroy*) и ослобађају се управљачки бројеви (*unregister_chrdev_region*). Слично као са функцијама *hough_init hough_probe*, тако се и после функције *hough_exit* позива *hough_remove*.

## 2.3: Функције *hough_probe* и *hough_remove*

Основна улога hough_probe функције јесте да изврши иницијализација уређаја као и алоцирање потребних ресурса за рад са њим. Прво се прибављају физичке адресе уређаја (*mem_start, mem_end* и *base_addr*). Дати адресни простор, односно опсег адреса, потребно је резервисати, како их кернел не би доделио у неку другу сврху – то се ради преко функције *request_mem_region*. Уколико је заузимање меморије прошло успешно, физичке адресе се морају мапирати у виртуелно како би се могло читати и уписивати у те меморијске локације (функција *ioremap*).

Уколико се нешто од наведених корака није успешно завршило, и све претходни успешни кораци се поништавају. У супротном, драјвер поседује виртуелну адресу уређаја, преко које може да помоћу функција *iowrite32* и *ioread32* уписује и чита из произвољних локација уређаја, а самим тим да обезбеди жељену функционалност драјвера.

С друге стране, функција *hough_remove* ради супротно од *hough_probe* функције. Она зауставља рад уређаја и ослобађа све ресурсе који се заузимају probe функцији.

С обзиром на то да се ове две функције позивају више пута, дефинисана је једна помоћна променљива *probe_counter*, која води рачуна о редоследу иницијализације и ослобађања ресура.

## 2.4: Функције за рад са фајловима

У већпоменутој структури *file_operations* налазе се показивачи на све функције које раде са фајловима. За овај пројекат су од интереса функције за упис и читање, те ће им се посветити пажња.

```
static struct file_operations my_fops =
{
    .owner = THIS_MODULE,
    .open = hough_open,
    .release = hough_close,
    .read = hough_read,
    .write = hough_write
};
```

**Слика 4.** Функције за рад са фајловима

Функција за упис података јесте *hough_write*. Ова функција омогућава упис у сва четири модула система. Очекују се два улазна параметра из бафера као медијума: вредност и адреса на коју се треба уписати, у формату „вредност, адреса“. Помоћу управљачког броја одређује се у ком медијуму је на одговарајућу адресу потребно уписати дату вредност. Једини изузетак јесте уписивање у *start* и *reset* регистар *IP* језгра, јер се и за њих очекује нека одређено вредност, али није битно шта ће бити прослеђено. Ако драјвер препозна да се ради о *start* регистру, он ће у њега уписати 1, сачекати да вредност унутар

*ready* регистра падне на 0, након чега ће у *start* регистар уписати 0. Слично је и са *reset* регистром – драјвер ће за сваки случај и у *start* регистар уписати 0, у *ready* 1, сачекаће да вредност у *ready* регистру скочи на 1, након чега ће у *reset* регистар уписати 0. Тако се обезбеђују правилан почетак обраде и ресет система.

За читање се користи функција *hough_read*. Њом се такође омогућава чиатање из сва четири модула, а у зависност од управљачког броја ,она зна којиа је циљани модул. По њеном позиву, она кориснику шаље вредности са свих могућих локација у оквиру тог модула.

Битно је и навести да је уведени и ред за чекање (**readyQ**). Они прекидају процесе у случају покушаја уписавања у регистре или BRAM, или услед покушаја читања BRAM јединица док унутар језгра траје обрада (*ready* = 0). Такође, уведен је и семафор (**sem**), којим се огранчива да два процеса истовремено нешто уписују или читају из хардвера.

Имплементација свега наведеног се налази у фајлу *hough_driver.c* који се налази у прилогу.

## Одељак 3: Развој апликације

За претходно развијени драјвер, потребно је развити и одговарајућу апликацију. Идеја је да апликација личи на *cpu* компоненту из виртуелне платформе приликом моделовања на системском нивоу. То је и учињено, те се само променила имплементација функција за уписивање и читање из регистара језгра и његових блок меморија.

Функција за упис у регистре језгра је *write_hard*. Аргументи ове функције су вреднос и адреса за дату вредност, те се оне уписују у */dev/hough_core* фајл, како драјвер и очекује. Из језгра се чита путем функције *read_hard*. С обзиром на то да драјвер кориснику враћа вредности из свих регистара, излаз ове функције ће зависити од прослеђеног аргумента (адресе) уз функцију.

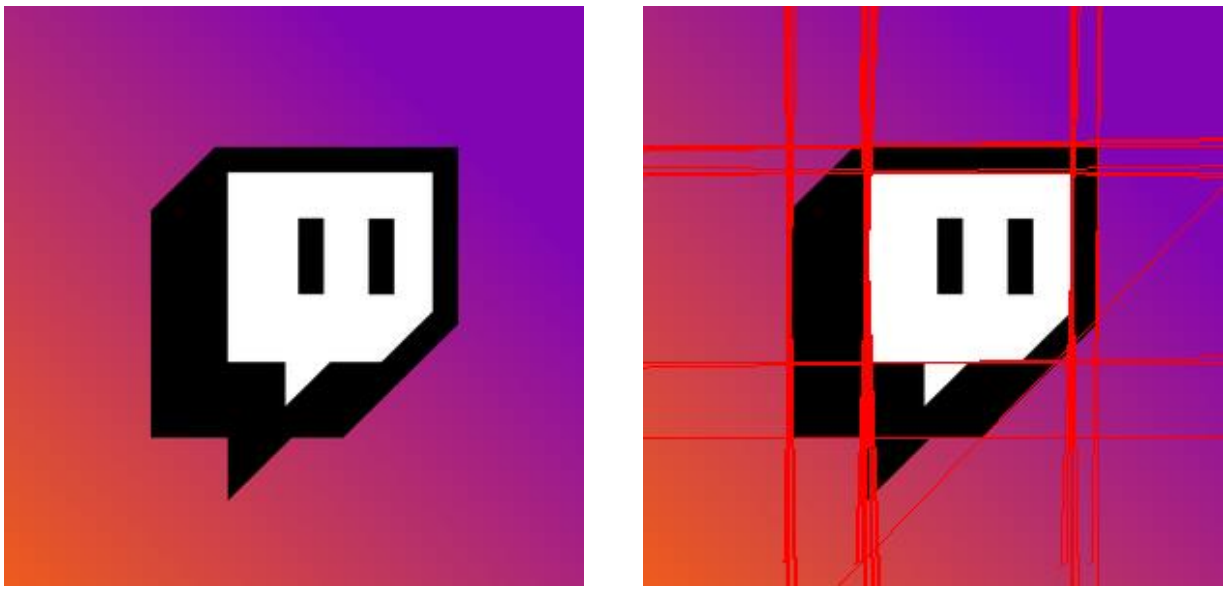
За уписивање у блок меморије је намењена функција *write_bram*. Аргументи ове функције су број од кога зависи у коју меморију се уписује, почетна адреса низа који се уписује и његова дужина. Слично томе, аргументи функције *read_bram*, која је намењена за читање, су број од кога зависи из које меморије се чита, почетна адреса на коју треба да се сачува учитани низ и његова дужина.

Имплементација свега наведеног се налази у фајлу *app_functions.c* који се налази у прилогу.

## Одељак 4: Подизање система и тестирање на уређају

За подизање Линукс оперативног систем на *Zybo* развојној платформи коришћено је развојно окружење Петалинукс (енгл. *Petalinux).* То је моћан алат којиа садржи све програме који су потребни за подизање, развијање, дебаговање и тестирање Линукс ембедед платформе. Петалинукс генерише четири главне компоненте које су неопходне за рад Линукс оперативног система: Линукс кернел (*image.ub*), стабло уређаја (*system.dtb*), *Bootloader* (*BOOT.BIN*) и *Root* фајл систем (*rootfs.tar.gz*). Ови фајлови се у правилном формату пребацују на *SD* картицу са које се Линук оперативни систем после подиже сваки пут када се плоча упали. Пошто ова развојна платформа има и *Ethernet* порта и самим тим могућ приступ интернету, са *git*-а су преузети кодови за драјвер и апликацију, након чега је уследило тестирање система.

Систем је тестиран за детектовање линија на фотографијама које су коришћене и за симулацију у виртуелној машини и при мешаној симулацији. Вредности унутар акумулатора након трансформације су идентичне онима раније, што уз верификацију, о којој се овде не говори, представља додатну потврду да систем ради како треба. С обизром на идентичне вредности унутар акумулатора, при истом фактору исцртавања исцртавају се и исте линије, те је и крајњи резултат истоветан.



*а)* *б)*

**Слика 5.** Пример улазне (а) и резултујуће (б) фотографије