دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

معماری کامپیوتروریزپردازنده:
پردازش تصویر

استاد درس:

دکتر سیدین

اعضای گروه:

مازیار دادور | 9723029
علی مطلبی |9723086

زمستان 1400

فهرست مطالب

الگوریتم پیاده شده:

کرنل گوسی :

1. دیتا های تصویر نویزی بعد از اعمال پدینگ را به صورت بایت های پشت سر هم در حافظه ذخیره میکنیم.
2. آدرس شروع دیتا ها را در رجیستر r0 ذخیره میکنیم.
3. مقدار رجیستر r0 را با 18 جمع میکنیم و در r0 میریزیم، تا از دیتا های پد سطر اول و همچنین اولین دیتای سطر دوم که آن نیز دیتای پد است بگذریم و به آدرس اولین دیتای اصلی تصویر برسیم.
4. آدرس شروع حافظه که میخواهیم پیکسل های فیلتر شده را از آن به بعد در آن ذخیره کنیم را در رجیستر r5 میریزیم.
5. مقدار 15 را در رجیستر r1 به عنوان شمارنده سطر ها میریزیم.
6. لیبل حلقه سطر ها را قرار میدهیم.
7. مقدار 15 را در رجیستر r2 به عنوان شمارنده ستون ها میریزیم.
8. لیبل حلقه ستون ها را قرار میدهیم.
9. در ابتدای حلقه ستون ها مقدار 0 را در رجیستر r3 قرار میدهیم.(این رجیستر حاصل عملیات کرنل(پیکسل فیلتر شده) را در خود قرار میدهد.)
10. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) با آفست 18- را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 11 ماتریس کرنل است)
11. از آنجایی که وزن این پیکسل در کرنل 1 است، بنابراین فقط با مقدار r3 جمع شده و در r3 ذخیره میشود.
12. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) با آفست 17- را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 12 ماتریس کرنل است)
13. از آنجایی که وزن این پیکسل در کرنل 2 است، بنابراین r4 را یک واحد به چپ شیفت میدهیم سپس با مقدار r3 جمع شده و در r3 ذخیره میشود.
14. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) با آفست 16- را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 13 ماتریس کرنل است)
15. از آنجایی که وزن این پیکسل در کرنل 1 است، بنابراین فقط با مقدار r3 جمع شده و در r3 ذخیره میشود.
16. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) با آفست 1- را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 21 ماتریس کرنل است)
17. از آنجایی که وزن این پیکسل در کرنل 2 است، بنابراین r4 را یک واحد به چپ شیفت میدهیم سپس با مقدار r3 جمع شده و در r3 ذخیره میشود.
18. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 22 ماتریس کرنل است)
19. از آنجایی که وزن این پیکسل در کرنل 4 است، بنابراین r4 را دو واحد به چپ شیفت میدهیم سپس با مقدار r3 جمع شده و در r3 ذخیره میشود.
20. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) با آفست 1+ را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 23 ماتریس کرنل است)
21. از آنجایی که وزن این پیکسل در کرنل 2 است، بنابراین r4 را یک واحد به چپ شیفت میدهیم سپس با مقدار r3 جمع شده و در r3 ذخیره میشود.
22. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) با آفست 16+ را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 31 ماتریس کرنل است)
23. از آنجایی که وزن این پیکسل در کرنل 1 است، بنابراین فقط با مقدار r3 جمع شده و در r3 ذخیره میشود.

24. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) با آفست 17+ را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 32 ماتریس کرنل است)
25. از آنجایی که وزن این پیکسل در کرنل 2 است، بنابراین r4 را یک واحد به چپ شیفت میدهیم سپس با مقدار r3 جمع شده و در r3 ذخیره میشود.
26. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) با آفست 18+ را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 33 ماتریس کرنل است)
27. از آنجایی که وزن این پیکسل در کرنل 1 است، بنابراین فقط با مقدار r3 جمع شده و در r3 ذخیره میشود.
28. مقدار رجیستر r3 (رجیستر حاصل اعمال کرنل) را بر مقدار 16 تقسیم میکنیم.
29. حاصل رجیستر r3 را در آدرس موجود در رجیستر r5 (آدرس ذخیره پیکسل فیلتر شده در حافظه)
30. رجیستر r5 (آدرس ذخیره پیکسل فیلتر شده در حافظه) را یکی اضافه میکنیم.
31. رجیستر r0 را یکی اضافه میکنیم.(برای رفتن به پیکسل نویزی بعدی)
32. رجیستر r2 (شمارنده ستون) را یکی کم میکنیم.
33. رجیستر r2 (شمارنده ستون) را با مقدار 0 مقایسه میکنیم.
34. اگر برابر نبود به 9 بازمیگردیم.
35. به آخر سطر رسیده ایم بنابراین رجیستر r0 را دوتا اضافه میکنیم تا به آدرس پیکسل اصلی در سطر بعد، پس از طی کردن پیکسل های پد شویم.
36. رجیستر r1 (شمارنده سطر) را یکی کم میکنیم.
37. رجیستر r1 (شمارنده سطر) را با مقدار 0 مقایسه میکنیم.
38. اگر برابر نبود به 7 بازمیگردیم.
39. پایان

کرنل تشخیص لبه :

1. دیتا های تصویر اصلی بعد از اعمال پدینگ را به صورت بایت های پشت سر هم در حافظه ذخیره میکنیم.
2. آدرس شروع دیتا ها را در رجیستر r0 ذخیره میکنیم.
3. مقدار رجیستر r0 را با 18 جمع میکنیم و در r0 میریزیم، تا از دیتا های پد سطر اول و همچنین اولین دیتای سطر دوم که آن نیز دیتای پد است بگذریم و به آدرس اولین دیتای اصلی تصویر برسیم.
4. آدرس شروع حافظه که پیکسل های فیلتر شده را از آن به بعد در آن ذخیره کنیم را در رجیستر r5 میریزیم.
5. مقدار 15 را در رجیستر r1 به عنوان شمارنده سطر ها میریزیم.
6. لیبل حلقه سطر ها را قرار میدهیم.
7. مقدار 15 را در رجیستر r2 به عنوان شمارنده ستون ها میریزیم.
8. لیبل حلقه ستون ها را قرار میدهیم.
9. در ابتدای حلقه ستون ها مقدار 0 را در رجیستر r3 قرار میدهیم.(این رجیستر حاصل عملیات کرنل(پیکسل فیلتر شده) را در خود قرار میدهد.)
10. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) با آفست 17- را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 12 ماتریس کرنل است)
11. از آنجایی که وزن این پیکسل در کرنل 1 است، بنابراین فقط با مقدار r3 جمع شده و در r3 ذخیره میشود.
12. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) با آفست 1- را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 21 ماتریس کرنل است)
13. از آنجایی که وزن این پیکسل در کرنل 1 است، بنابراین فقط با مقدار r3 جمع شده و در r3 ذخیره میشود.

14. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 22 ماتریس کرنل است)
15. از آنجایی که وزن این پیکسل در کرنل 4- است، بنابراین ابتدا r4 را دو واحد به چپ شیفت میدهیم.
16. رجیستر r4 را قرینه میکنیم سپس با مقدار 1 جمع میکنیم تا متمم 2 آن بدست آید (قرینه شود)
17. رجیستر r4 با مقدار رجیستر r3 جمع شده و در r3 ذخیره میشود.
18. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) با آفست 1+ را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 23 ماتریس کرنل است)
19. از آنجایی که وزن این پیکسل در کرنل 1 است، بنابراین فقط با مقدار r3 جمع شده و در r3 ذخیره میشود.
20. بایت موجود در آدرس موجود در رجیستر r0 (پیکسل مرکزی) با آفست 17+ را در رجیستر r4 ذخیره میکنیم.(مقدار این بایت مطابق با درایه 32 ماتریس کرنل است)
21. از آنجایی که وزن این پیکسل در کرنل 1 است، بنابراین فقط با مقدار r3 جمع شده و در r3 ذخیره میشود.
22. مقدار رجیستر r3 (رجیستر حاصل اعمال کرنل) را بر مقدار 4 تقسیم میکنیم.
23. مقدار رجیستر r3 (رجیستر حاصل اعمال کرنل) را با مقدار 64 جمع میکنیم.
24. مقدار رجیستر r3 را با عدد 100 مقایسه می کنیم، اگر کوچکتر یا مساوی بود مقدار آن را 0 میکنیم در غیر اینصورت مقدار آن را حداکثر مقدار یک بایت یعنی 255 میکنیم.
25. حاصل رجیستر r3 را در آدرس موجود در رجیستر r5 (آدرس ذخیره پیکسل فیلتر شده در حافظه)
26. رجیستر r5 (آدرس ذخیره پیکسل فیلتر شده در حافظه) را یکی اضافه میکنیم.
27. رجیستر r0 را یکی اضافه میکنیم.(برای رفتن به پیکسل بعدی)
28. رجیستر r2 (شمارنده ستون) را یکی کم میکنیم.
29. رجیستر r2 (شمارنده ستون) را با مقدار 0 مقایسه میکنیم.
30. اگر برابر نبود به 9 بازمیگردیم.
31. به آخر سطر رسیده ایم بنابراین رجیستر r0 را دوتا اضافه میکنیم تا به آدرس پیکسل اصلی در سطر بعد، پس از طی کردن پیکسل های پد شویم.
32. رجیستر r1 (شمارنده سطر) را یکی کم میکنیم.
33. رجیستر r1 (شمارنده سطر) را با مقدار 0 مقایسه میکنیم.
34. اگر برابر نبود به 7 بازمیگردیم.
35. پایان

عملکرد کد:

در کد اسمبلی نوشته شده دو بخش موجود است :

بخش مربوط به تنظیم رجیستر های پردازنده stm32f407VGT برای تنظیم برخی از پورت های GPIO و همچنین بخش مربوط به عملیات محاسباتی فیلترینگ و اعمال کرنل ها.

بخش مربوط به محاسبات فیلترینگ به تفصیل در قسمت قبلی توضیح داده شد.

حال به بررسی تنظیمات رجیستری میپردازیم.

همانطور که در صورت پروژه خواسته شده است، با فعال شدن پین 0 پورت A عملیات فیلترینگ شروع میشود و با پایان یافتن عملیات پین 1 پورت B را فعال میکنیم.

بنابراین باید تنظیمات رجیستری برای پورت ها و پین های مربوطه انجام دهیم. در تمام خطوط کد سعی شده که با کامنت گذاری عملکرد کد روشن باشد.

ابتدا در قطعه کد زیر، کلاک باس مربوط به پورت های A,B را فعال میکنیم.

```
;********** Activating GPIOA, GPIOB Clock ***************************************
MOV32   r0, #AHB1ENR          ; r0 <- GPIO Enable Clock Register Address(AHB1ENR)
LDR     r1, [r0]              ; r1 <- AHB1ENR Register Value
MOV32   r2, #0x00000003       ; r2 <- 0b00000000000000000000000000000011
ORR     r1, r1, r2            ; r1 <- r1 OR r2 (AHB1ENR | 0b11)
STR     r1, [r0]              ; AHB1ENR Register <- r1
;********************************************************************************
```

سپس رجیستر های MODER, OSPEEDR, PUPDR در PORTA را برای تنظیم کردن PIN0 آن تنظیم میکنیم.

```
;********** Configuring GPIOA.PIN0 as input 2MHz Without Pullup/Pulldown *********
MOV32   r0, #GPIOA_MODER      ; r0 <- GPIOA MODER Register Address
LDR     r1, [r0]              ; r1 <- GPIOA_MODER Register Value
MOV32   r2, #0x00000003       ; r2 <- 0b00000000000000000000000000000011
MVN     r2, r2;               ; r2 <- ~r2
AND     r1, r1, r2            ; r1 <- r1 AND r2 (GPIOA_MODER & ~0b11)
STR     r1, [r0]              ; GPIOA_MODER Register <- r1

MOV32   r0, #GPIOA_OSPEEDR     ; r0 <- GPIOA OSPEEDR Register Address
LDR     r1, [r0]              ; r1 <- GPIOA_OSPEEDR Register Value
MOV32   r2, #0x00000003       ; r2 <- 0b00000000000000000000000000000011
MVN     r2, r2;               ; r2 <- ~r2
AND     r1, r1, r2            ; r1 <- r1 AND r2 (GPIOA_OSPEEDR & ~0b11)
STR     r1, [r0]              ; GPIOA_OSPEEDR Register <- r1

MOV32   r0, #GPIOA_PUPDR      ; r0 <- GPIOA PUPDR Register Address
LDR     r1, [r0]              ; r1 <- GPIOA_PUPDR Register Value
MOV32   r2, #0x00000003       ; r2 <- 0b00000000000000000000000000000011
MVN     r2, r2;               ; r2 <- ~r2
AND     r1, r1, r2            ; r1 <- r1 AND r2 (GPIOA_PUPDR & ~0b11)
STR     r1, [r0]              ; GPIOA_PUPDR Register <- r1
;********************************************************************************
```

به همین ترتیب سپس رجیستر های MODER, OTYPER, OSPEEDR, PUPDR در PORTB را برای تنظیم کردن PIN1 آن تنظیم میکنیم.

```
;********** Configuring GPIOB.PORT1 as output 2MHz Push Pull Without Pullup/Pulldown *********
MOV32   r0, #GPIOB_MODER    ; r0 <- GPIOB MODER Register Address
LDR     r1, [r0]            ; r1 <- GPIOB_MODER Register Value
MOV32   r2, #0x00000004     ; r2 <- 0b00000000000000000000000000000100
ORR     r1, r1, r2          ; r1 <- r1 OR r2 (GPIOB_MODER | 0b100)
STR     r1, [r0]            ; GPIOB_MODER Register <- r1

MOV32   r0, #GPIOB_OTYPER   ; r0 <- GPIOB OTYPER Register Address
LDR     r1, [r0]            ; r1 <- GPIOB_OTYPER Register Value
MOV32   r2, #0x00000002     ; r2 <- 0b00000000000000000000000000000010
MVN     r2, r2;             ; r2 <- ~r2
AND     r1, r1, r2          ; r1 <- r1 AND r2 (GPIOB_OTYPER & ~0b10)
STR     r1, [r0]            ; GPIOB_OTYPER Register <- r1

MOV32   r0, #GPIOB_OSPEEDR  ; r0 <- GPIOB_OSPEEDR Register Address
LDR     r1, [r0]            ; r1 <- GPIOB_OSPEEDR Register Value
MOV32   r2, #0x0000000C     ; r2 <- 0b00000000000000000000000000001100
MVN     r2, r2;             ; r2 <- ~r2
AND     r1, r1, r2          ; r1 <- r1 AND r2 (GPIOB_OSPEEDR & ~0b1100)
STR     r1, [r0]            ; GPIOB_OSPEEDR Register <- r1

MOV32   r0, #GPIOB_PUPDR    ; r0 <- GPIOB_PUPDR Register Address
LDR     r1, [r0]            ; r1 <- GPIOB_PUPDR Register Value
MOV32   r2, #0x0000000C     ; r2 <- 0b00000000000000000000000000001100
MVN     r2, r2;             ; r2 <- ~r2
AND     r1, r1, r2          ; r1 <- r1 AND r2 (GPIOB_PUPDR & ~0b1100)
STR     r1, [r0]            ; GPIOB_PUPDR Register <- r1
;*******************************************************************************
```

پس از تنظیم رجیستر های اولیه با استفاده از قطعه کد زیر بوسلیه یک حلقه منتظر فعال شدن پایه 0 پورت A میمانیم.

```
        ;****** Waiting for PORTA.0 to Activate ***********************
PA0Loop
        MOV32   r0, #GPIOA_IDR      ; r0 <- GPIOA_IDR Register Address
        LDR     r1, [r0]            ; r1 <- GPIOA_IDR Register Value
        MOV32   r2, #0x00000001     ; r2 <- 0b00000000000000000000000000000001
        AND     r1, r1, r2          ; r1 <- r1 AND r2 (GPIOA_IDR & 0b10)
        CMP     r1, #1
        BNE     PA0Loop
        ;*************************************************************
```

(نکته : از آنجایی که برای بررسی کد نوشته شده از شبیه سازی (simulator) استفاده می شود، چون پین 0 پورت A ورودی تعیین شده است و باید بصورت خارجی (با استفاده از کلید یا روشی دیگر) آن را فعال کنیم (رجیستر IDR، ReadOnly است و تنها با اعمال خارجی بر پین مربوطه مقدار میگیرد). در شبیه سازی راهی برای تست آن وجود ندارد. بنابراین در شبیه سازی راهی برای خروج از حلقه بالاوجود ندارد بنابراین باید کامنت شود.)

پس از آن به بدنه اصلی کد یعنی محاسبات کرنل ها میرسیم که توضیحات کامل آنها در قسمت قبل آمده است.

و پس از اتمام عملیات مربوط به کرنل ها پین 1 پورت B را فعال میکنیم.

```
;****** Finished => PORTB.1 = 1 ***********************************
            MOV32   r0, #GPIOB_ODR      ; r0 <- GPIOB ODR Register Address
            LDR     r1, [r0]            ; r1 <- GPIOB_ODR Register Value
            MOV32   r2, #0x00000002     ; r2 <- 0b00000000000000000000000000000010
            ORR     r1, r1, r2          ; r1 <- r1 OR r2 (GPIOB_ODR | 0b10)
            STR     r1, [r0]            ; GPIOB_ODR Register <- r1
;*******************************************************************
```

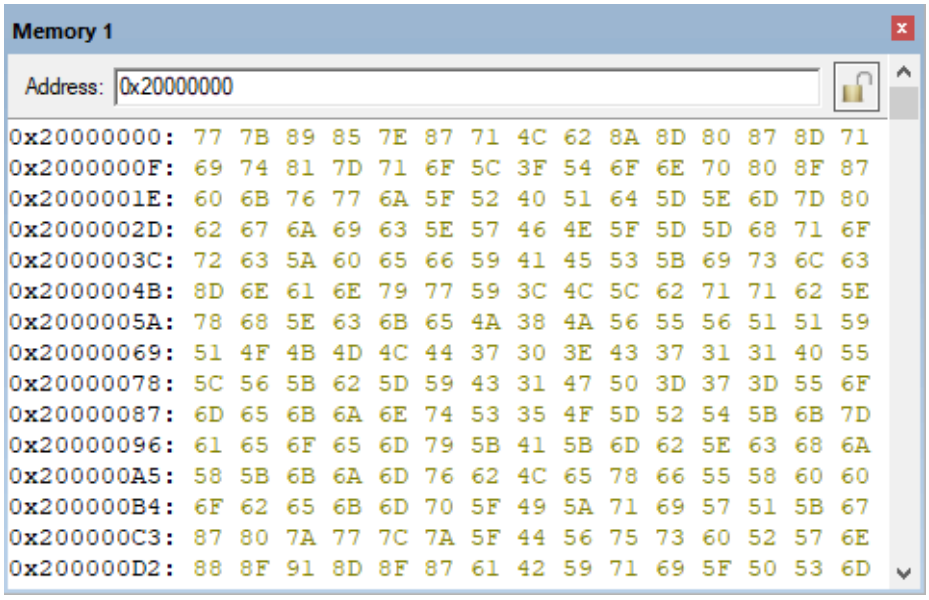
نتایج خروجی:

نتایج خروجی شامل تغییرات رجیستر های GPIOA, GPIOB و AHB1ENR مربوط به بخش RCC, GPIO در بخش system viewer و همچنین پیکسل های خروجی عملیات فیلتیرینگ پس از ذخیره در حافظه SRAM از آدرس شروع SRAM به بعد یعنی آدرس 0x20000000 حافظه است.

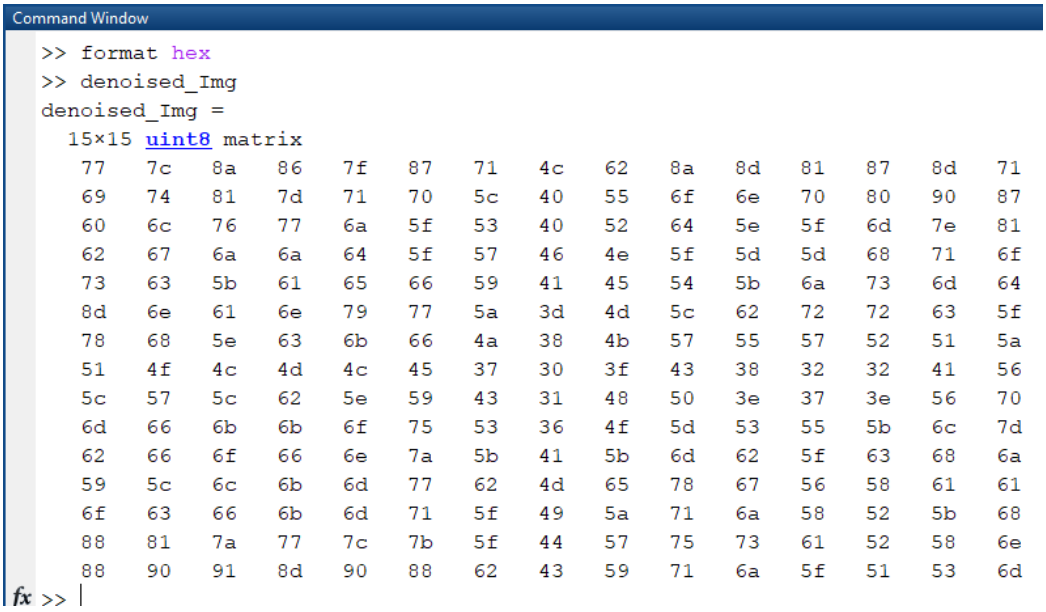
با اجرای کد و رفتن به قسمت system viewer میتوان رجیستر ها را مشاهده کرد و برای مشاهده حافظه با رفتن به قسمت memory window و تایپ آدرس حافظه 0x20000000 اطلاعات ذخیره شده را نیز مشاهده کرد.

در ادامه تصویر بلوک حافظه که در آن پیکسل های فیلتر شده را قرار داده ایم و همچنین پیکسل های مربوطه در شبیه سازی متلب که اسکریپت های آن موجود است را قرار داده و علاوه بر آن مقایسه ای در نتایج خواهیم داشت.

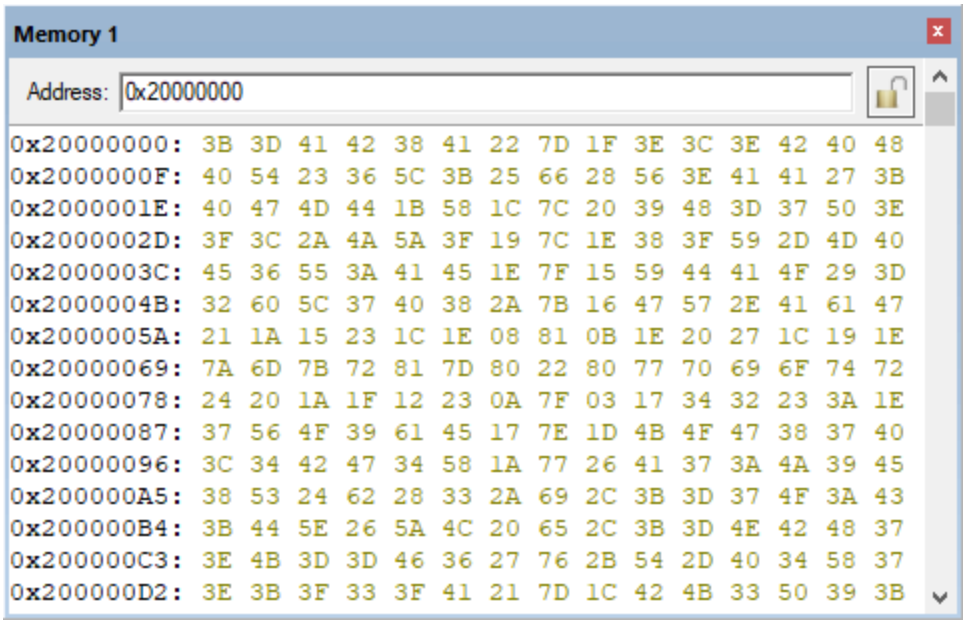
نتیجه کرنل فیلتر گاوسی در حافظه پردازنده :



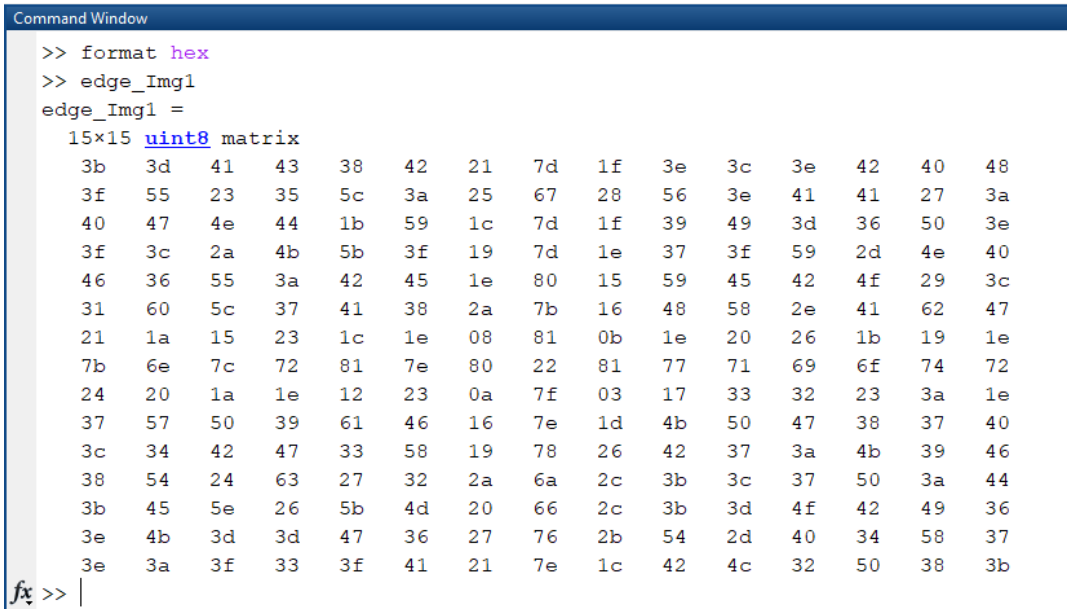
نتیجه شبیه سازی در متلب :



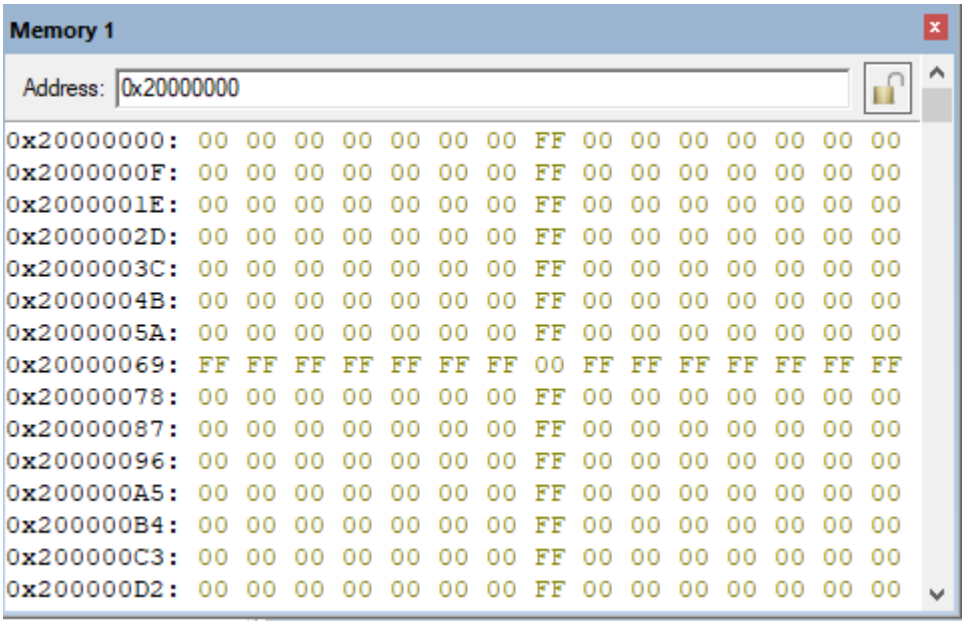
نتیجه کرنل تشخیص لبه در حافظه پردازنده پیش از اعمال حد آستانه :



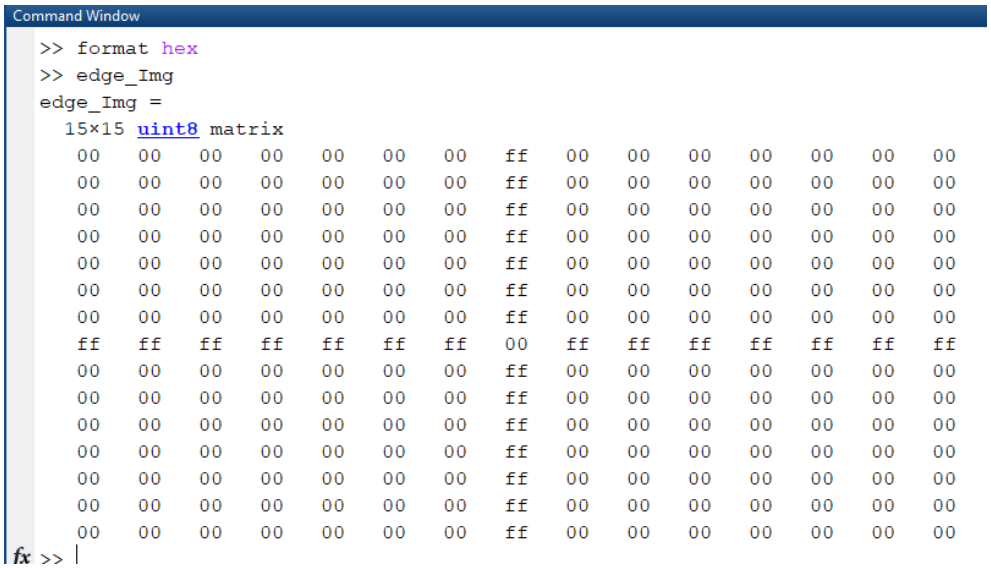
نتیجه شبیه سازی کرنل تشخیص لبه پیش از اعمال حد آستانه در متلب :



نتیجه کرنل تشخیص لبه در حافظه پردازنده پس از اعمال حد آستانه :



نتیجه شبیه سازی کرنل تشخیص لبه پس از اعمال حد آستانه در متلب :



تصاویر مربوط به شبیه سازی در متلب با استفاده از نتایج اجرای اسکریپت های آماده بدست آمده اند.

همچنین برای مقایسه بهتر با مقادیر ذخیره شده در حافظه حاصل کد اسمبلی format در متلب را بصورت hex تبدیل کرده ایم.

نکته : با مقایسه مقادیر شبیه سازی متلب و مقادیر بدست آمده در حافظه پردازنده متوجه میشویم که برخی مقادیر به اندازه 1 واحد با یکدیگر اختلاف دارند. دلیل این موضوع آن است که در دستور العمل تقسیم در پردازنده بخش اعشاری عدد به کلی حذف میشود. اما در متلب پس از تقسیم عدد بسته به مقدار اعشارش گرد میشود. (اگر اعشار بزرگتر از 0.5 باشد به یک واحد بزرگتر گرد شده در غیر اینصورت جزء صحیح عدد نتیجه میشود.)