**შესავალი**

ამ დოკუმენტში მექნება კონსპექტი რომელსაც შევადგენ Embedded Programming ვიდეო კურსიდან, კონსპექტი ცოტა ამახინჯებს თიქთქოს ამ დოკუმენტს, უფრო კონკრეტულად რომ ვთქვა ესაა დოკუმენტი სადაც დეტალურად იქნება განხილული როგორც კონტროლერის მუსაობის პრინციპი, მეხსიერების მენეჯმენტი, რეგისტრები, პერიფერიალები და სხვა ასევე პროგრამირება რის საშუალებითაც ვახდენთ კონტროლერის კონტროლს. საფუძვლიანად იქნება განმარტებული ყველა ეტაპი და არ იქნება მხოლოდ მონაკვეთები, პარარელურად თუ დაიწყებ საქმის კეთებას კარგი ბიჭი იქნები. დავიწყებ იმით რომ ეს არაა დამწყებებისთვის, რაც გულისხმობს რომ უნდა გვქონდეს საფუძველი ცოტა ცოტა მიკროკონტროლერებში და ცოტა ცოტა პროგრამირებაში. პროგრამირებას რაც შეეხება არის C ენაზე. გისურვებთ გართობას.

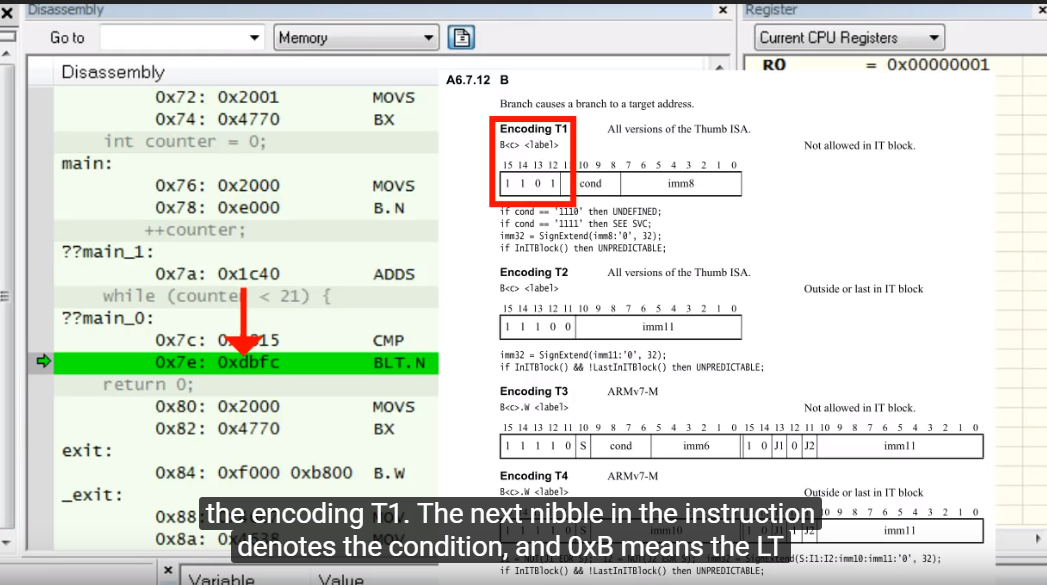
**ლექცია 1 დათვლა**

ამ ლექციაზე შევეხეთ თემას თუ როგორ ხდება დათვლა კონტროლერში, arm კონტროლერები ფანსხვავებით AVR ისგან მონაცემებს ინახავენ პორტის რეგიტრებში ყოველთვის არა მაგრამ როცა საშუალება აქვთ ოპერატიულის დაზოგვის მიზნით კოდში გამოყენებული ინფორმაციის დამახსოვრება ხშირად ხდება რეგისტრებში. არის 32 ბიტიანი სისტემა რაც ნიშნავს 32 ბიტიან რეგისტრებს. წინასწარ ვიტყვი რომ საკმაოდ რთული კონფიგურაციაა საჭირო რეგისტრებისთვის რადგანაც ისევ ავეერს რომ შევადაროთ იქ გვქონდა 8 ბიტიანი სისტემა და რეგისტრების კონფიგურაცია შედარებით მარტივი იყო. როგორც შევატყეთ arm პროცესორები შედარებით გაუმჯობესებულები არიან და გამოიყენებიან უფრო მეტი დავალების შესასრულებლად, უფრო კონკრეტულად რომ ვთქვათ აქვს უფრო მეტი რესურსი პინებთან კავშირი, თვითონ პინები. თუნდაც წყვეტა რომელიც ყველა პინზე გვხვდება. ავეერისგან განსხვავებით 8 ბიტი, პინებზე მიგვება წყვეტის ფუნქცია. ასევე შევეხეთ სიმულატორში IDE ში რეგისტრების დისასიმბლირების და ოპერატიულში არსებულ მისამართებს თუ როგორ ხდება ინფორმაციის ჩაწერა და რა ცვლილებები ხდება მათში. ძირითადად სულ ეს იყო გისურვებთ სწავლის შიმშილს.

**ლექცია 2 ნაკადების კონტროლი (დინების, რომ არ შევიდეთ შეცდომაშ ვიტყვი ციკლს)**

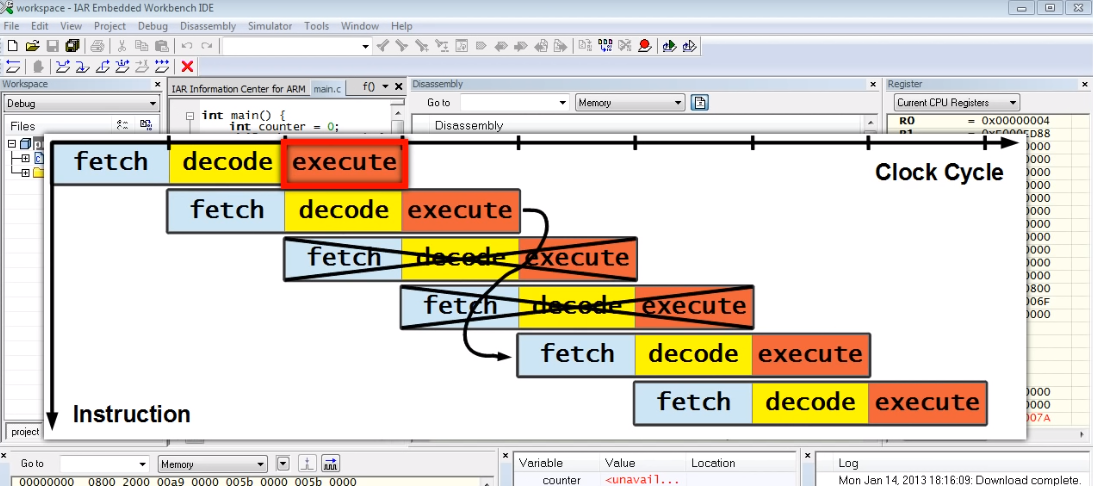
პროცესორს აქვს რეგისტრები, ზოგიერთ რეგისტრში არის უკვე ჩადებული ინფორმაცია ზოგიერთი საჭიროებს კონფიგურაციას. მზა რეგისტრებში არის უკვე გამზადებული ინსტრუქციები და თუ პროგრამ ქაუნთერი იგივე PC რეგისტრი გამოიძახებს თითოეულს ნიშნავს რომ უნდა შესრულდეს მასში არსებული ინსტრუქცია.

განვიხილოთ PC (Program Counter Register) რომელშიც იწერება მისამართი, მიმდინარე ინსტრუქციის. ახლა შევეხოთ APSR (Application Program Status Register) რეგისტრს, სადაც გვაქვს ფლაგები, N – negative, Z – zero, C – carry(ანუ Unsigned Overflow), V – overflow. როცა გვაქვს if ასემბლერისთვის CMP ანუ შედარების ბრძანება ეს ბრძანება ათავსებს Z flag ზე 1 იანს როცა შედარების დროს არის სხვაობა მაგ 1 < 2, z ფლაგზე გვექნება , ანუ ნაკლებობის პირობა არაა ჭეშმარიტი. მაგალითს აქვე დავწერ if(1 < 2), ან while(I < 21), ამ ბრძანების შემდეგ მოხდება ზემოთ რაც დავწერე ის. ციკლის როს როცა მოწმდება პირობა პროგრამა გადადის იმ მისამართზე რაც უწერია ინსტრიქციას წინასწარ უკვე არსებულ მისამართზე.



ახლა აღვწეროთ თუ რა ხდება ფოტოზე, 0x7e არის ოპერატიულის მისამართი, 0xdbfc არის ინსტრუქცია ამ მისამართზე იგივე blt.n. ენკოდირებულ ინტრუქციაში 0xdbfc ში თითოეული ნიშნავს შემდეგს: d ანუ ორობითი 1101 ნიშნავს რომ ჩვენ ვიყენებთ T1 ენკოდირებას, იგი წერია დატაშიტში. ჩავხედოთ ისევ ფოტოში და ვნახოთ encoding T1. 0xb(1011) აღნიშნავს b პირობას(B condition) არის LT condition. და ბოლო fc ში გვაქვს ენკოდირება სადაც წერია თუ როგორ შეიცვლება PC რასაც ჰქვია offset, ოფსეტი არის შეზღუდული რაოდენობის და გადავსებისას ხდება ბრუნვა როგორც 1 ბაიტში თუ 9ს ჩავწერთ რომ უდრის 0ს, 8 რვას და რვის შემდეგ რომ ხდება გადავსება. Fc იქნება -4, ამის შემდეგ ჩვენ გვეცოდინება PC ის შემდეგი ინსტრუქციის მისამართი არსებულ მისამართზე ანუ 0x7e ზე თუ არის გამოვაკლებთ 4 ს, და მივიღებთ მივიღებთ 7A-ს რაც უკვე PC ისთვის ახალი მისამართია.

ყველა თანამედროვე პროცესორი რომელიც შეიცავს arm cortex-m პროცესორს იყენებენ ინსტრუქციების მილს რომ გაზარდონ შედეგიანობა. მილი pipeline არის ასიმბლერის მსგავსი, რომელიც პროცესორში მუშაობს მრავალ ინსტრუქციაზე და სახვადასხვა დასრულების ეტაპებზე. ეს ზრდის ინსტრუქციების რაოდენობას რომელიც ეშვება მოცემულ დროს. ინსტრუქციები გამოყოფილია თანმიმდევრულად დამოუკიდებელ ეტაპებად, ასე მიიღება fetch მეხსიერება decode და execute. ფოტოზეა აღნიშნული თუ როგორ არის მიმაგრებული ინსტრუქციები ერთმანეთზე ქლოქის მუშაობის პარარელურად.



**ლექცია 3 ცვლადები & მიმთითებლები**

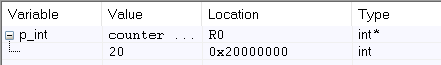
როცა მთავარ main ფუნქციაში ვწერდით ცვლადებს აღწერისას ისინი იქმნდებოდნენ რეგისტრებში. R0, R1… და ა.შ, თუ უბრალოდ აღწერილი გვაქვს და არ ვიყენებთ შეიძლება გადააწეროს კომპილერმა სხვა ინფორმაცია რეგისტრს. ანუ თუ აღვწერეთ ინტ ტიპის რაიმე ცვლადი და მივანიჭეთ 17 თუ ამ ცვლადს არსად ვიყენებთ მომავალში კიდევ თუ აღვწერთ ცვლადს შეიძლება ახალი აღწერილი ცვლადის ჩაიწეროს იმ რეგისტრში სადაც გვეწერა ცვლადი რომელსაც არ ვიყენებთ. **როდესაც გლობალურ ცვლადს ვქმნით ის იწერება ოპერატიულ მეხსიერებაში.** პროგრამაში მუშაობისას კომპილატორი ოპერატიულის ამ მისამართს ჩაწერს რეგისტრში და მისამართზე არსებულ ინფორმაციას ანუ რიცხვს ჩაწერს სხვა რეგისტრში. ASM – STR Store.   
 ARM არის ერთერთი RISC კატოგორიის ელემენტი (Reduced Instruction Set Computer), სადაც მეხსიერებაში არსებული ინფორმაციის ამოკითხვა შესაძლებელია სპეციალური ინსტრუქციებით, ხოლო დანარჩენი ინფორმაციის (data) მანიპულაცია ხდება რეგისტრებზე. საბოლოოდ შეცვლილი მოდიფიცირებული რეგისტრის მნიშვნელობა შესაძლებელია ისევ ჩაწერილი იქნას ოპერატიულ მეხსიერებაში. ეს მიდგომა განსხვავდება CISC Complex Instruction Set Computer არქიტექტურისგან. CISC გვხვდება კომპიუტერებში X86 პროცესორები, არ სჭირდება მოქმედების ობიექტს იყოს რეგისტრებში იგი შეგვიძლია მოვათავსოთ ოპერატიულში. ასე რომ უნდა დავაფასოთ დავაფასოთ მეხსიერება ოპერატიულში იგი მოითხოვს საკმაო ცოდნას რომ მოვახდინოთ ჩაწერა ან წაკითხვა სასურველ მისამართზე. Arm პროცესორებისთვის ეს ერთერთი ფუნდამეტური ცოდნის საწინდარია. მისამართებთან სამუშაოდ აუცილებელია მიმთითებლები რომლებიც მიუთითებენ მისამართზე და მასში არსებულ მონაცემზე. როგორც წესი მიმთითებლებში გვაქვს მისამართები და შეგვიძლია როგორც წვდომა მისამართზე ასევე მისამართზე არსებულ მონაცემზე.

**------ მცირე გადახვევა -----**

**int volatile \* const PortRegister; // ასე ჩაწერის დროს რომ არ დავიბნეთ უნდა წავიკითხოთ აღწერა ბოლოდან. ცვლადი PortRegister რომელიც არის კონსტანტ მიმთითებელი არის volitale და არის ინტ ტიპის.**

**------ მადლობა ყურადღებისთვის -----**

ვნახოთ როგორ ხდება ჩაწერა მისამართზე მნიშვნელობის.



p\_int არის მიმთითებელი type: int \* როგორც ფოტოზეა არის counter ცვლადის მიმთითებელი და მისი მისამართი მდებარეობს R0 რეგისტრზე. ხოლო დაბლა ხაზზე მნიშვნელობები გვაქვს თუ R0 ში რა მისამართია, ამ მისამართზე რა ტიპის ინფორმაცია გვაქვს და რა არის ეს ინფორმაცია(20). ახლა კი მთავარი რაც არის ის გავიაროთ მიმთითებლებში რაც არმში გამოგვადგბა. ჩვენ შეგვიძლია პოინტერში(ტარტოლოგია რომ არ გამოვიდეს ხან პოინტერს ვიტყვი ხან მიმთითებელს) პირდაპირ რაიმე მისამართი ჩავწეროთ. ფოტოში ვხედავთ მისამართს 0x20000000 რომელიც არის ოპერატიულის მისამართი. პირდაპირ ვერ მივანიჭებთ გვჭირდება type casting და რიცხვის დაკონკრეტება, აი ასე

int \*p;

p = (int \*) 0x20000000U; // U არის unsigned ნიშნის მიკონკრეტება

\*p = 0xDEADBEEF; //მისამართის ჩაწერა მისამართზე.

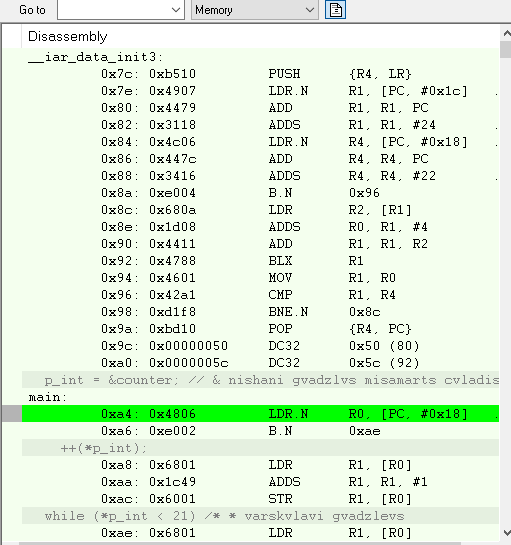
\*p = 11; //მისამართზე ჩაიწერება b (hex) ანუ 11 (dec)

ასე რომ მიმთითებლები გზას და კარს გვიხსნიან ახალ შესაძლებლობებზე ანუ მისამართებზე.

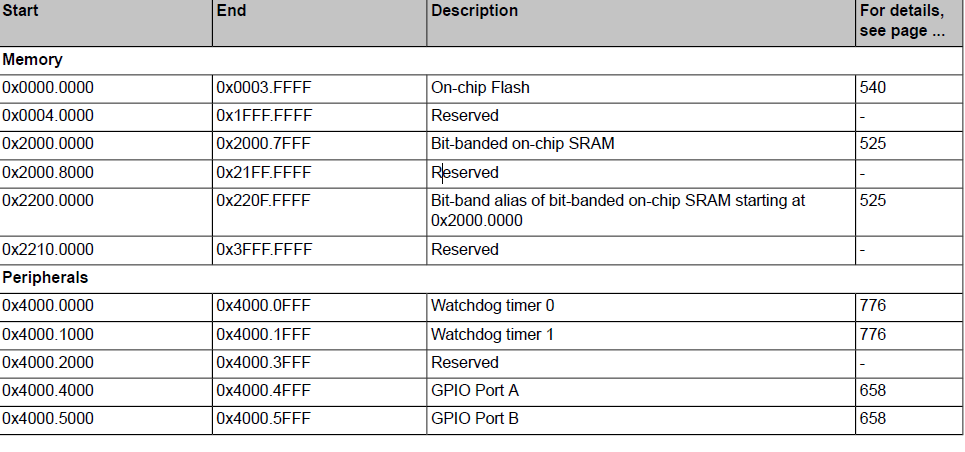
პ.ს CORTEX პროცესორის მოდელები განსხვავდებიან მისამართების შენახვის მხრივ შესაძლოა ზოგიერთმა დაკავებული მისამართის თავისუფალი ბიტები დაიკავოს ან უბრალოდ შემდეგი თავისუფალი ადგილი ნახოს ახალ მისამათზე. გამოვიყენოთ ფრთხილად და დაკვირვებით თორემ საშიშები არიან პოინტერები.

**ლექცია 4 ნათურის ციმციმი**

განვიხილოთ დამისამართება ცეპეუში:

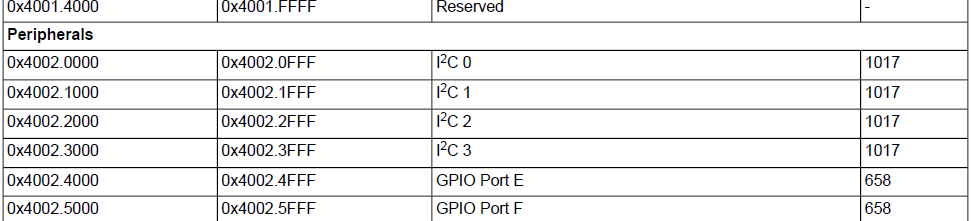


სურათზე როგორც სჩანს Low საწყისი მისამართები არიან 0 ები, შეგვიძლია ვნახოთ ინსტრუქციები. ეს არის დაკომპილირებული კოდი ჩვენი პროგრამის, რომელიც ჩაწერილია მიკროკონტროლერში. საწყისი ანუ LOW 0 -თ დაწყებული მისამართები არიან მიამაგრებული **ფლეშ მეხისერებაზე.** მისამართები რომლებიც იწყებიან 0x2 ით არიან ცვლადებისთვის, შემდეგ მოყვება 0 ები. ე.ი ჩვენ შეგვიძლია ვნახოთ მისამართები და მათზე არსებული ინფორმაცია დასაწყისიდან. 0x20000000 ამდე დებაგერს არ შეუძლია წაიკითხოს მანამდე არსებული მონაცემი რადგანაც ძირითადად ისინი არიან გამოუყენებლები. ოპერატიული მთავრდება 0x20008000 მისამართზე. 0x8000 == 32kb (32768) რაც ნიშნავს რომ კონტროლერს აქვს 32 კილობაიტი ოპერატიული მეხსირება (RAM Random Access Memory). მთლიანი მეხსიერების ნაწილის ამ კუნძულზე (20008000 -- 20000000) განთავებულია და იყენებს ოპერატიული. ახლა კი დავიწყოთ ნათურის ანთებაზე მუშაობა თან ვისწავლოთ უფრო მეტი. უნდა ვიცოდეთ რუკა ყველა სხვა კუნძულების და კონტინენტების როგორც ამ შემთხვევაში ოპერატიულის, ამისათვის გვაქვს საყვარელი დატაშიტი. დატაშიტში გვაქვს ზღვა იფორმაცია რაც arm პროცესორს აქვს არის 1000 ზე მეტი გვერდი და ამის მთლიანი სწავლა ხომ სისულელეა. საჭიროა ვიცოდეთ რასთან გვაქვს საქმე რა გვჭირდება და სწორად მოვძებნოთ ის დატაშიტში. აღნიშნულ დოკუმენტში არის მისამართების მთელი ნაკრები თუ რომელ მისამართზე რა არის მოთავსებული.

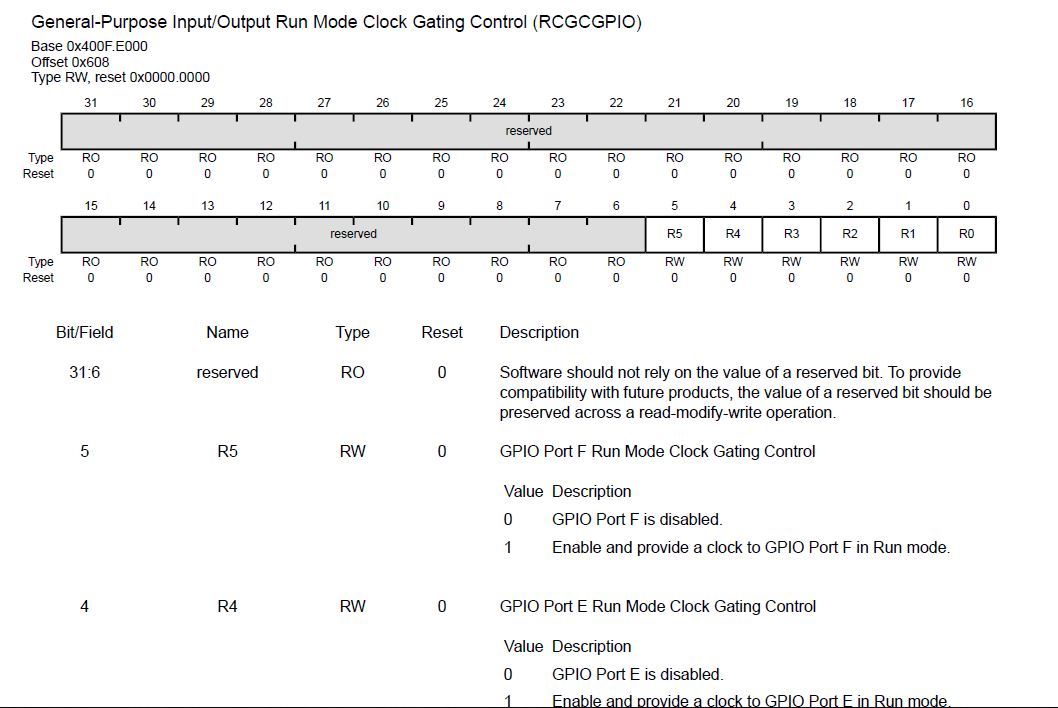


ეს არასრული ნაკრებია ფოტოზე.

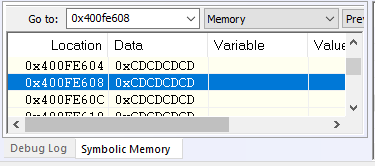
პირველი აქ რაც არის 0x00000000 დან 0x0003ffff ამდე არის On-chip Flash, 256kb იგი შეესაბამება ფლეშ მეხსიერებას. 20000000 დან 20008000 ამდე როგორც უკვე ვიცით არის Bit-banded on-chip SRAM (Static Random Access Memory). Peripherals ეს ბლოკი ძალზედ საინტერესოა მივაქციოთ ყურადღება GPIO (General Purpose Input Output) რადგანაც ჩვენ უნდა შევეხოთ პორტებს და ავანათოთ ნათურა.



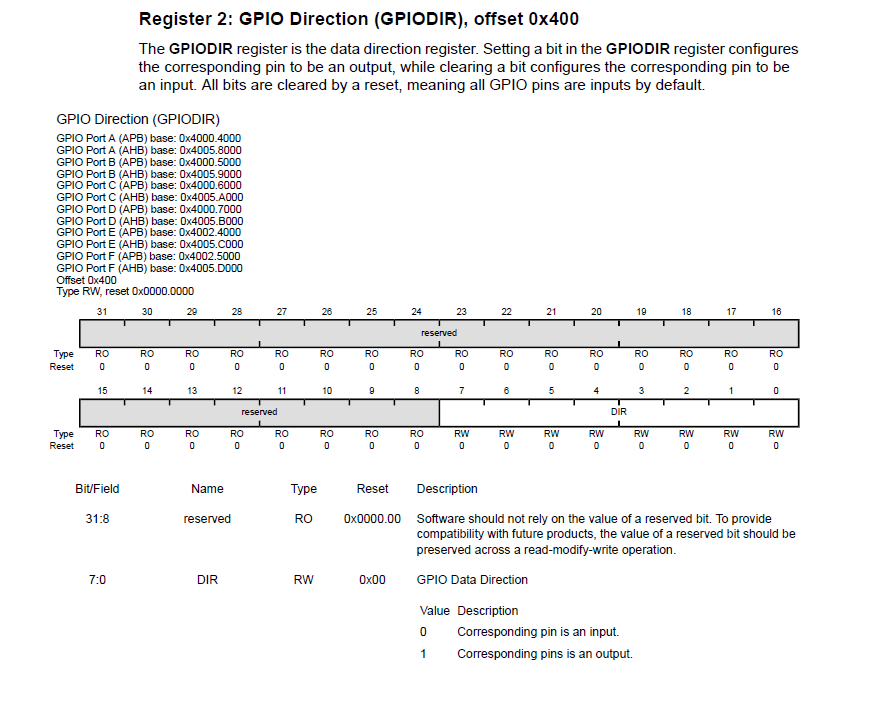
ეს კიდევ ნაწილი მეხსიერების მისამართების. ჩვენ გვჭირდება Port F რადგანაც ამ პორტზეა ჩამაგრებული მიკროკონტროლერის პლატაზე RGB ნათურა. შეგვიძლია სხვა პორტი ავიღოთ და სხვა ნათუდა მივამაგროთ სხვა პორტს. ფოტოზე მისი საწყისი მისამართია 0x40025000, საწყისად დებაგერით თუ ვნახავთ მეხსიერებას ეს მისამართი იქნება ცარიელი რადგანაც ვზოგავთ ენერგიას. ახლა კი ვნახოთ თუ როგორ უნდა ჩავრთოთ GPIO-F ბლოკი, ეს ნისნავს იმას რომ უნდა დავბრუნდეთ დატაშიტზე და მოვძებნოთ შემდეგი დასახელება clock gating (რომელიც იქნება GPIO - სთვის).



რეგისტრი გვიჩვენებს ბიტების ბლოკს, რომელიც ყოველთვის იწყება 0 დან. პორტის ტიპები მითითებულია სურათზე RO Read Only, RW Read Write, WO Write Only. როგორც აღწერაში გვაქვს Clock Gating Control გვაქვს 5 პინზე. ამ ბიტით შეგვიძლია გავააქტიუროდ CGC, ანუ ეს არის ის რეგისტრი რომელსაც ვეძედით. დაგვჭირდება base address რომელიც წერია ზემოთ მარცხენა კუთხეში 0x400fe000 დამატებითი Offset მისამართი 0x608 გვჭირდება რომ დავამაოთ GPIO მისამართის საწყის მისამართს და მოვხვდებით ამ რეგისტრის მისამართზე. აი ასეთი სახე ენქება:



იმისათვის რომ ვაკონტროლოთ ეს რეგისტრი გადავედით მის მისამართზე. Data-ში უნდა ჩავწეროთ 5 (მეექვსე, რომელიც ააქტიურებს CGC-ს) ბიტზე 1 ანუ უნდა ჩავწეროთ ყველა ბიტში 0 გარდა მეექვსე ბიტისა 00000000000000000000000100000 იგივეა რაც hex ებში 20. ასე რომ ჩავწეროთ მასში 0x00000020, რაც გააღვიძებს ჩვენს რეგისტრს. იმისათვის რომ ავანათოთ ნატურა უკვე რეგისტრის პინის ბიტებზე უნდა ჩაავწეროთ 1 ები გვაქვს RGB ნათურა. ისევ ერთხელ მოგვიწევს დატაშიტში ჩახედვა რომ განვახორციელოთ შეტანა გატანა პინზე.



ეს არის მიმართულების მისამართები ჩვენ გვინდა F პორტი რომლის 1,2,3 ბიტები მართავენ RGB - ის. Offset ვიცით რაც არის უკვე ასე რომ გადავიდეთ 0x40025400 მისამართზე (0x40025000, პორტის მიმართულების საწყისი მისამართი მეხსიერებაში და 400 offset ) და ჩავწეროთ შემდეგნაირად 1110 ბიტებში რომ სამივეზე გავიტანოთ ინფორმაცია ეს რიცხვი თექვსმეტობითში არის E. იმისათვის რომ გავხადოთ პინები Digital Output ები უნდა ჩამოვიდეთ GPIO-F ბლოკის 0x4002551C მისამართზე და მასშიც ჩავწეროთ e სიმბოლო. საბოლოოდ რომ ვაკონტროლოთ ნათურა, უნდა გავაკეთოთ ეს GPIO-F პორტზე რომელიც მდებარეობს 0x400253fc მისამართზე და ამ მისამართზე რომელ ბიტსაც გავააქტიურებთ ის ფერი აინთება ანუ 1,2,3 რომელსაც ჩავწერთ იმის შესაბამის ფერს აანთებს, ვცადოთ 2 რომლის ( bin 10 ანუ ფერი ნომერი 1) hex არის 2, აინთება წითელი ნათურა, (100) ანუ მეორე ნათურა hex 4 ს თუ ჩავწერთ ლურჯი და (1000) მესამე hex 8 ჩაწერისას მწვანე. ახლა კი იგივე უნდა გავაკეთოთ პოინტერებით რომ ვწეროთ მეხსიერებების მისამართებზე სასურველი მნიშვნელობები. კოდს ჩავამატაბ დაბლა გვერდზე.

int main()

{

\*((unsigned int \*) 0x400fe608U) = 0x20U; // GPIO-F block on

\*((unsigned int \*) 0x40025400U) = 0x0eU; // GPIO-F pin direction

\*((unsigned int \*) 0x4002551cU) = 0x0eU; // digital func Register

while (1)

{

\*((unsigned int \*) 0x400253fcU) = 0x02U; // GPIO data register red on

int counter = 0;

while(counter < 1000000) // dayovneba ^\_^

{

counter ++;

}

\*((unsigned int \*) 0x400253fcU) = 0x00U; // GPIO data register off

counter = 0;

while(counter < 1000000) //dayovneba ^\_^

{

counter ++;

}

}

return 0;

}

**ლექცია 5 Preprocessor & Volatile**

წინა ლექციაზე დავწერეთ კოდი რომელიც შეიძლება საშინლად არ გამოიყურება თუმცა გამარტივება და გალამაზება არ აწყენდა. პრეპროცესორი პროგრამირებაში არის კოდის ის მონაკვეთი რომელსაც კომპილერი არ გადის, დაკომპილირებამდე იცის რომ ეს ინსტრუმენტი გვაქვს როგორც არის ხოლმე #include ბიბლიოთეკები. გვაქვს ასევე დეფინიცია #define რომელსაც ჰქვია macro ფუნქციები, საშუალებას გვაძლევენ ფართო კოდები შევამოკლოდ სასურველი სახელი დავარქვათ და ამ სახელის გამოყენებისას იგულისხმება კოდში ის ნაწილი რასაც დავარქვით.

#define RCGCGPIO (\*((unsigned int \*) 0x400fe608U)) //clock gating control

RCGCGPIO ამ სახელს სადაც გამოვიყენებთ პროგრამაში იგულისხმება ის რაც მის შემდეგ წერია დეფინაციისას. ამ დროს არ იზრდება შრომა კონტროლერის იგივე რესურსს ხარჯავს. თუ ვნახავთ lm4f120h5qr.h ბიბლიოთეკას მასშიც ზუსტად მაკრო ფუნქციებით აწყობილი კოდია უკვე გამზადებული. ჩვენს დაწერილ დეფინიციასა და ბიბლიოთეკის დეფინიციას შორის განსხვავება არის განმსაზღვრელი სიტყვა volatile (არამდგრადი), რომელიც ასე წერია

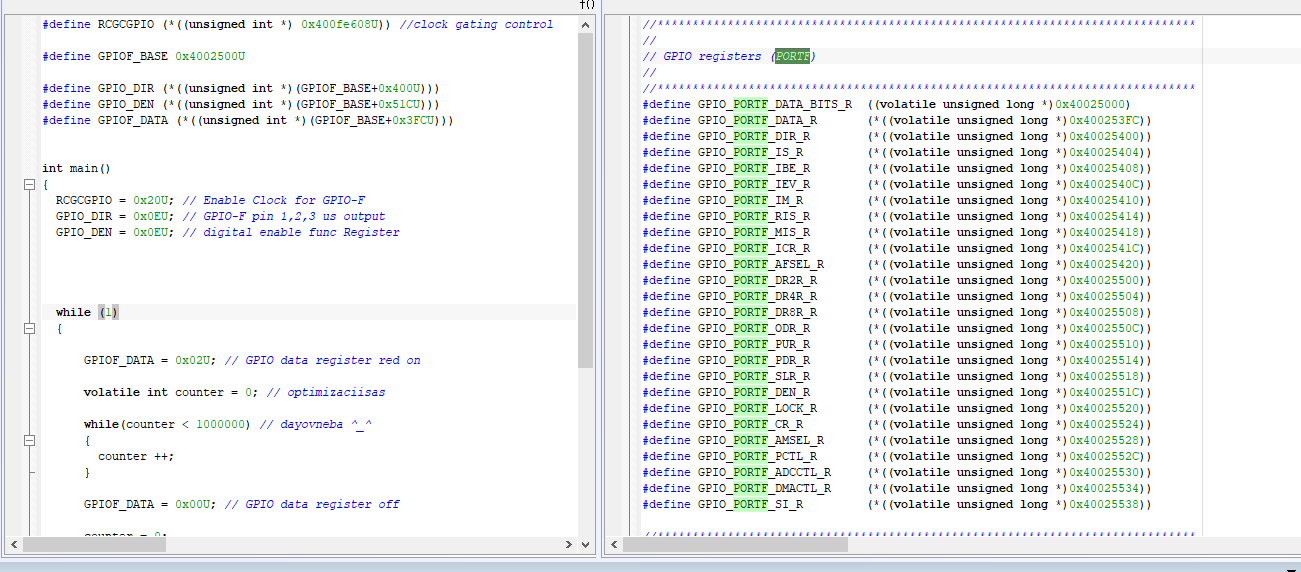
#define GPIO\_PORTA\_DATA\_BITS\_R ((volatile unsigned long \*)0x40004000)

Volatile განმსაზღვრელი კომპილატორს ამცნობებს რომ შეიძლება მისი მნიშვნელობა შეიცვალოს სპონტანურად. როცა მას ვიყენებთ ჩვენ ვეუბნებით კომპილერს რომ ობიექტი შეიძლება შეიცვალოს მაშინაც კი როცა არ მივმართავთ მას პროგრამიდან. მაგალითად ღილაკს თუ მივანიჭეთ ამ ცვლადს, პირდაპირ ღილაკის ჩართვა გამორთვით იცვლება მისი მნიშვნელობა. შესაბამისად რეგისტრის პორტები არიან volatile ები. კომპილერი აკეთებს კოდის ოპტიმიზაციას ცვლადებს ათავსებს რეგისტრებზე ინახავს მათ და ა.შ volatile განმსაზღვრელის შემთხვევაში კი ამ ცვლადზე აღარ აკეთებს ოპტიმიზაციას, უბრალოდ კითხულობს და წერს ამ ცვლადში ინფოს. როცა უბრალოდ ცვლადის აღწერისას ოპტიმიზაცია შეიძლება მოახდინოს ერთხელ ჩაწეროს მხლოდ მნიშვნელობა ან არ გაიაროს მისი აღწერა მოკლედ რანაირი ოპტიმიზაციაა ვერ მივხვდი :/ თუკი კოდის ნაწილს ისე არ შეასრულებს როგორც გვიწერია, გამოდის ოპტიმიზაცია უნდა ვისწავლოთ იგი ხომ კოდს ამსუბუქებს მაგრამ გვიღირს ასეთი შემსუბუქება? უნდა ვიცოდეთ როგორ ამსუბუქებს კოდს რომ შემთხვევით ოპტიმიზაციის ცოცხს საჭირო ნაწილი არ გავაყოლოთ. ისევ volatile განმსაზღვრელზე ვიტყვი, კომპილერი ყოველ ჯერზე ამოწმებს კითხულობს ან წერს ამ ცვლადს და ისინი როგორც წესი გამოიყენებიან გარე მოწყობილობებთან კავშირისთვის, GPIO პორტებზე, წყვეტის დროს, კომუნიკაციის და სხვა პეროფერიული მოწყობილობის დროს. ოპტიმიზაციის მაგალითიც განვიხილოთ წყვეტა არის ტიპი რომელიც კოდისგან დამოუკიდებლად აწყვეტინებს ცეპეუს მუშაობას და თავის საქმეზე გადმოჰყავს შემდეგ ისევ აგრძელებეს ცეპეუ. ანუ მოვლენის დაჭერისთვის კარგი რამეა წყვეტა, წყვეტას ველოდებით რომრლიმე წყვეტის პინზე და შემდეგ როცა ხდება რაიმე მოვლენა ვასრულებთ რაიმე დავალებას. ახლა წარმოვიდგინოთ ოპტიმიზაციის დროს როგორ იქნება. პროგრამაში წყვეტას თუ ვიყენებთ მისი ფუნქციის გამოძახება არ ხდება კოდიდან უბრალოდ გვაქვს ფუნქცია რომელიც წყვეტის დროს ეშვება როგორც უკვე ვთქვი პროცესორისგან დამოუკიდებლად. ოპტიმიზირებული კომპილერი შეიძლება მოიქცეს შემდეგნაირად: ნახოს რომ ამ ფუნქციას არაფერი იყენებს და არ ხდება მისი გამოძახება კოდში ამიტომ შეიძლება წაშალოს იგი, ამისთვის ვიყენებთ volatile ცვლადს რომ არ წაშალოს და აუცილებლად ნახოს იგი კომპილერმა. შეიძლება არც შექმნას ცვლადი რომელსაც კოდი არ იყენებს ან უბრალოდ არაფერს ცვლის, ან დაყოვნების ლუპები თუ გვაქვს ამისათვის ან უნდა გავთიშოთ ოპტიმიზაცია ან გამოვიყენოთ volatile რომ მსგავსი პრობლემა არ გვქონდეს.

ახლა დავუბრუნდეთ ლექციას და უკვე დამატებულ ბიბლიოთეკის კოდით დავწეროთ ჩვენი დაწერილი პროგრამა. ჩვენ გავაკეთეთ განმარტებები მისამართების რათა უფრო გასაგები ყოფილიყო კოდი და გამარტივებული.

#define GPIOF\_DATA (\*((unsigned int \*)(GPIOF\_BASE+0x3FCU)))

არსებობს უკვე მზა ბიბლიოთეკები რომლებიშიც უკე გაკეთებულია განსაზღვრება თითოეული მისამართის და როცა გვჭირდებიან ისინი უბრალოდ შეგვიძლია ხშირი და საჭირო განმარტებები ვისწავლოთ ან თვითონ .h ფაილში ვნახოთ თუ რომელ მისამართს რომელი განმარტება აქვს.



**ლექცია 5 Bit-wise Operators**

ორობითი ოპერატორები

c = a | b; // OR

c = a&b; // AND

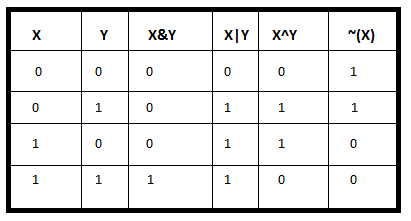
c = a ^ b; // exclusive OR

c = ~b; // inversion, NOT

c = b >> 1; // right-shift

c = b << 1; // left-shift

თითოეული ადარებს თითოეულ ბიტს როგორც ფოტოზეა დაბლა ნაჩვენები.



ახლა ვნახოთ მაგალითები:

**11001100 & 11110000 == 11000000**, თითეულ ბიტის ინდექსით იღებს ანუ პირველს პირველ ადარებს და თუ ორივე 1 მაშინ წერს 1-ს.

**11001100 | 11110000 == 11111100**, ან ელემენტის მსაგვსად ადარებს და თუ რომელიმე მაინც იქნება 1 მაშინ წერს მათივე ინდექსის შესაბამისად 1-ს.

**11001100 ^ 11110000 == 00111100**, ამ შემთხვევაში მხოლოდ ერთერთი ბიტის 1 იანად ყოფნის შემთხვევაში წერს პასუხში 1-ს.

**~ 11110000 == 00111100**, არა ელემენტი კი მხოლოდ მაშინ წერს 1 როცა ბიტის მნიშვნელობა არის 0.

**11001100 >> 1 == 01100110**; მარჯვნივ გადააქვს ბიტები და თავში უმატებს 0-ს. იგივე მნიშვნელობის ათობით ჩანაწერი რომ გავყოთ 2 ზე იგივეა, 11001100 არის 204 >> 1 არის 01100110 102, რადგანაც ყოველი ბიტი რა ინდექსიც არის იქ იწერება 2 ის იმდენი ხარისხი რა ინდექსზეც არის რიცხვი. ანუ 1000000 არის 2 ის მერვე ხარისხი თუ ერთიანს მარჯვნივ წავანაცვლებთ გამოდის რომ იქნება 7 ის ხარისხი ანუ 2 ხარისხად 7 თუ არის 128 მერვე ხარისხში რომ ავიყვანოთ უნდა გავამრავლოთ 128, 2 ზე და მივიღებთ 256-ს რაც ნიშნავს იმას რომ თუ ბიტებს გადავიტანთ 1 ით მარჯვნივ უნდა გავყოთ 2 ზე რომ მივიღოთ დაბალი ხარისხი 2 ის მერვე ხარისხი არის 256, რომ გავყოთ 2 ზე მივიღებთ 2 ის მეშვიდე ხარისხს ანუ 128-ს. თუ 2 ბიტით წავანაცვლებთ მარჯვნივ ე.ი უნდა გავყოთ 2 ზე და კიდევ 2 ზე ანუ 4 ზე.

**11001100 << 3 == 01100000**; ამ შემთხვევაში მარცხნიდან დაამატებს 0 იანებს ბოლო სამ ბიტს და წინა სამ ბიტს გააქრობს.

სულ ესაა ორობითი ოპერატორები. აღსანიშნავია ნიშნიანი ანუ უარყოფითი რიცხვის არსებობის დროს რა ხდება, განსაკუთრებული არაფელი შედეგი ჩვეულებრივია უბრალოდ საინტერესოა როგორ ხდება მაგ 1024 ორობითშ არის

კოდი:

#define LED\_BLUE (1U << 1) // 0 ებს ბოლოში დაამატებს 1 სა მარცხნიდან ერთი ბიტით წაწევს ანუ მარცხნიდან დაემატება 0 და მარჯვენა პირველი ბიტი გაქრება

#define LED\_RED(1U << 2) // 0 ებს ბოლოში დაამატებს 1 და მარცხნიდან 2 ბიტით წაწევს ანუ მარცხნიდან დაემატება 00 და მარჯვენა პირველი ორი ბიტი გაქრება

#define LED\_GREEN (1U << 3) // 0 ებს ბოლოში დაამატებს 3 და მარცხნიდან 3 ბიტით წაწევს ანუ მარცხნიდან დაემატება 0 და მარჯვენა პირველი სამი ბიტი გაქრება

დღეისათვის სულ ეს იყო აბა წარმატებული თამაში.