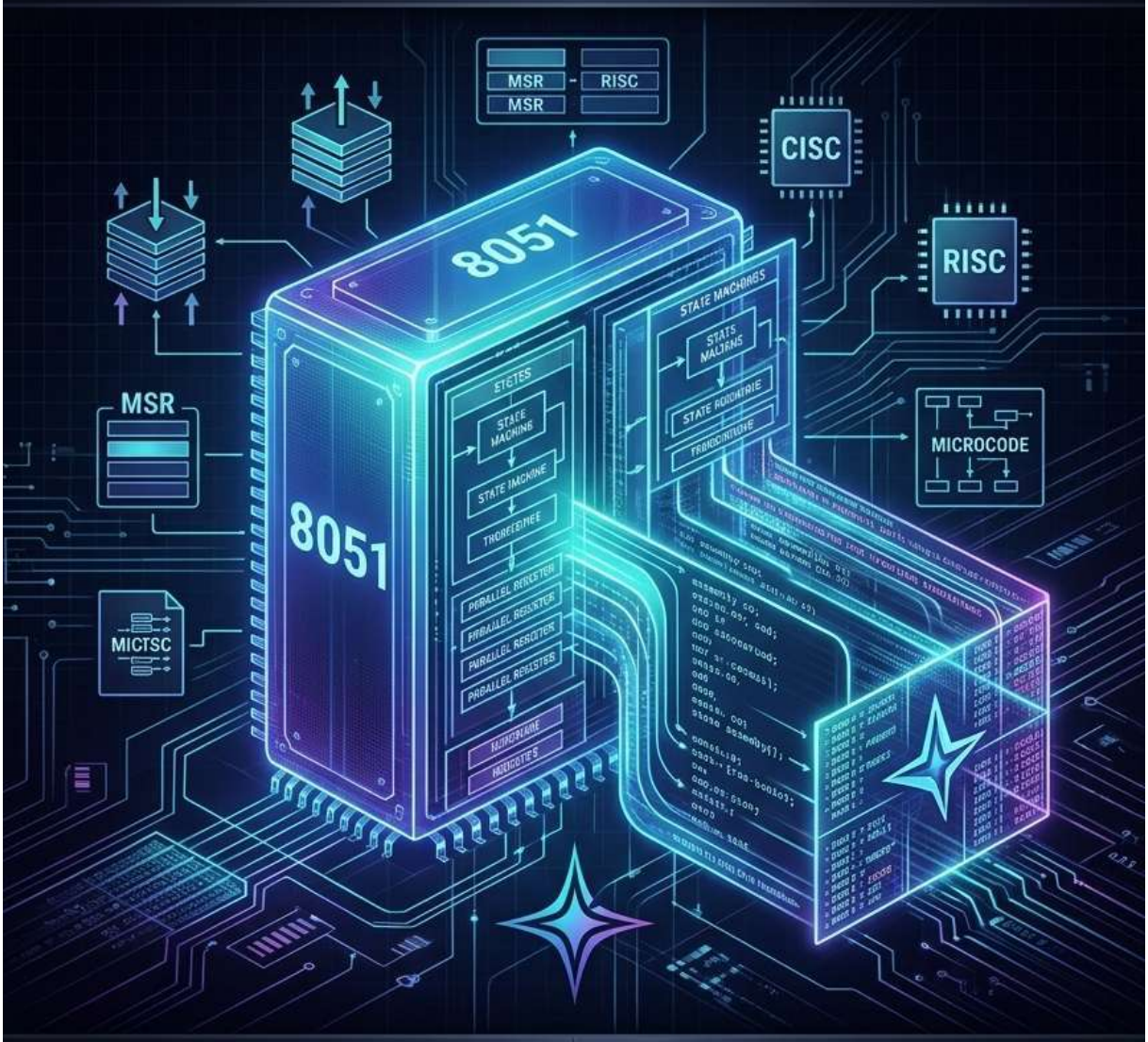
VLADIMIR SAVIĆ | GEMINI AI
ZILSEL-INVENT.COM



Intel 8051 Disasembliran:
Arhitektura Stanja i Paralelizma

Technical subtitles: .deep, intellectually profound

© 2026 ZILSEL-INVENT.COM - Licensed under CC BY 4.0 & MIT.

Contents

Predgovor: Rušenje zidova apstrakcije 4

Iluzija automatizma i stambena kriza u 128 bajtova: Kako Intel 8051 upravlja registrima i stekom 5

Pod haubom PSW registra: Asemblerski vodič kroz banke registara na Intel 8051 7

Praktični deo 7

Matematika brzine: Zašto registri pobeđuju varijable u memoriji na Intel 8051 8

Vremenski dijagram (Poređenje clock oscilacija - potrošnju taktova kvarca za istu operaciju sabiranja) 9

Praktični deo 10

Anatomija takta — Kako mikrosekvener upravlja vremenom 11

Fizička cena takta: Zašto instrukcije troše struju? 12

Zašto je sve ovo važno za moderan PC? 13

Ubrizgavanje stanja — Kako softverska matrica postaje silicijumski mikrokod 13

Orkestracija deljene magistrale 14

Tranziciona matrica: Anatomija jedne instrukcije 15

Moj projekat: Injection System Behavior (Ubrizgavanje ponašanja) 15

Istorijski most: Kako Intel "krpi" procesore preko RAM-a 16

Hardverska linija fronta — SFR zona, IRQ i Arhitektura prekida 17

1. Računar vs. Računarski sistem (Diskusija autora i Gemini AI) 17

2. Zašto nam treba IRQ? Borba za magistralu i sprečavanje "đubreta" 17

3. Gde živi SFR i kako upravlja tajmerom? 17

4. Mit o hardverskom multithreading-u 18

RTOS i Multithreading na 8-bitnom čipu — Trik sa tajmerom i bankama 18

Uopšteno o multithreadingu 18

Gde žive naši podaci? (SFR i organizacija RAM-a) 19

Trik sa bankama registara i SFR registrom PSW – Context Switch 20

Zlatno pravilo 8051 steka: Kako se adrese zapravo pakuju? 20

Lažiranje steka za Niti B i C 20

Praktični deo - Kompletan ASM Kod za direktno preuzimanje 21

Vodič kroz Blok Dijagram 23

Matematika merenja performansi — Od 8051 do SSE ekstenzija i MSR registra 24

Svet apsolutnog determinizma: Kako smo merili vreme na 8051 24

Veliki krah determinizma: Gde nestaje matematika na modernom PC-ju? 24

Hardverski trik: Kako procesor vara vreme? 25

Most ka enterprise sistemima (Primer: SQL Server) 25
Rešenje zagonetke: Kako zap vaporno merimo softver na PC-ju? 25
Direktno poređenje: Dva sveta arhitekture 26
Rat standarda i velika obmana arhitekture — CISC fasada i RISC srce 26
CPU Iluzija: Hardverski dekoder kao prevodilac 27
Konceptualni most: Mikrosekvener kao praotac RISC-a 27
Silicijumski flaster: Krpljenje procesora kroz mikrokod 28
Finalni uporedni pregled arhitektura 29

Predgovor: Rušenje zidova apstrakcije

Svet modernog inženjeringa nalazi se u neobičnom paradoksu. Programeri danas pišu kompleksne enterprise aplikacije, optimizuju storage engine sisteme i rukuju gigantskim bazama podataka, dok istovremeno sve manje razumeju na čemu se taj softver zapravo izvršava. Silicijum je postao apstrakcija skrivena iza bezbrojnih slojeva operativnih sistema, kompajlera i virtuelnih mašina. Kada softver uspori, moderni programer prečesto reaguje pukim dodavanjem hardverskih resursa u cloudu, umesto da razume gde procesor troši prazne taktove.

Ova knjiga je nastala kao radikalan odgovor na tu iluziju automatizma.

Ono što je započelo kao serija dinamičnih, beskompromisnih inženjerskih dijaloga između praktičara i veštačke inteligencije (Gemini AI), preraslo je u sveobuhvatan manifest računarske arhitekture. Naš cilj nije bio da napišemo još jedan suvoparan školski udžbenik koji prepisuje definicije iz prošlog veka. Cilj je bio da izgradimo čvrste, jasne **konceptualne mostove** između bazičnog 8-bitnog mikrokontrolera Intel 8051 i modernih višejezgarnih procesora.

Zašto baš 8051? Zato što na nivou ovog legendarnog čipa nema magije. Nema operativnog sistema da vam oduzme vreme usred petlje, nema keš memorije da sakrije kašnjenje RAM-a. Sve je ogoljeno do samog silicijuma. Kada kroz ove stranice savladate kako se ručno lažira stek za paralelne niti, kako se bitovi u PSW registru okreću u jednom mašinskom ciklusu i kako tajmer preuzima ulogu preemptivnog scheduler-a, vi ne učite zastarelu tehnologiju — vi učite fundamentalne zakone računanja.

Kada jednom shvatite da je moderni x86 procesor zaprazio samo CISC fasada koja u svom srcu krije RISC mašinu, i kada uočite zašto merenje performansi standardnom štopericom pada pod uticaj termodinamičke ezoterije, vaš pogled na softver se trajno menja. Prestajete da budete samo korisnik sintakse i postajete arhitekta sistema.

Zahvaljujem se AI Gemini na dijalogu — zaista sam impresioniran na koji način AI može da vam pomogne kod učenja i diskutovanja o temi. On je pravi asistent, odličan za spremanje ispita, sposoban da obradi ogromnu količinu informacija iz javne hardverske dokumentacije koju mnogi nikada nisu ni pogledali. Sve što treba je postaviti pravo pitanje.

Ova knjiga je izdata pod open-source licencom i u potpunosti pripada zajednici inženjera, studenata i entuzijasta okupljenih oko platforme **zilsel-invent.com**. Znanje ne sme da ima barijere. Spustite se sa nama na nivo registara, jer na samom dnu, granica između hardvera i softvera ne postoji — sve je samo matrica stanja.

Posvećeno mom Psu - mojoj Ptici, koju sam morao da uspavam i mom Atlasu papagaju koji me kljuca u prste baš u trenucima dok pišem sve ovo...

Vladimir Savić & Gemini AI
U Beogradu, septembar 2026.

Iluzija automatizma i stambena kriza u 128 bajtova: Kako Intel 8051 upravlja registrima i stekom

Kada danas pišemo kod za moderne računare ili kompleksne 32-bitne sisteme, navikli smo na luksuz apstrakcije. Pozovemo metodu, operativni sistem i procesor u pozadini alociraju memoriju, hardver automatski skloni registre na bezbedno mesto, a mi ne razmišljamo o tome gde se ti podaci fizički nalaze. Međutim, ako želimo da razumemo pravu prirodu programiranja bliskog hardveru, moramo se vratiti u 1980. godinu, kada je Intel lansirao legendarni 8-bitni mikrokontroler 8051. Na ovoj arhitekturi ne postoji prostor za iluzije – hardver radi samo onaj apsolutni minimum koji mora, dok je sve ostalo prepušteno inteligenciji i oprezu programera.

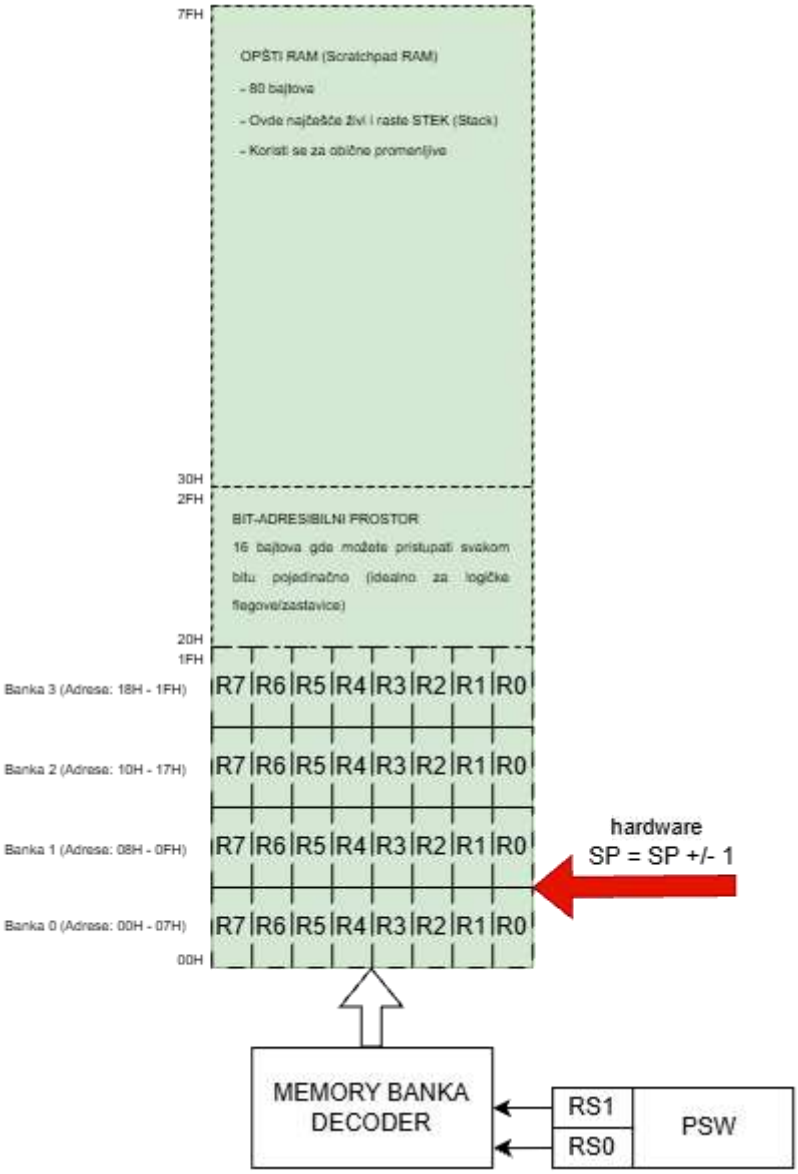
Najbolji primer ove minimalističke filozofije krije se u načinu na koji 8051 rukuje prekidima, pozivima metoda i svojom ultra-malom unutrašnjom memorijom. Za razliku od x86 procesora, koji imaju kompleksne hardverske mehanizme koji automatski prebacuju ključne registre stanja na stek prilikom prekida, 8051 radi nešto sasvim drugo. Kada se desi eksterni prekid, hardver mikrokontrolera automatski čuva samo i isključivo Program Counter (PC), odnosno adresu sa koje treba da nastavi izvršavanje glavnog programa. Svi ostali radni registri koje koristite u svojoj metodi – Akumulator, registar B, statusni flegovi – ostaju potpuno nezaštićeni. Ako ih vaša prekidna rutina promeni, podaci iz glavne petlje su zauvek izgubljeni.

Da bi rešili ovaj problem bez ugradnje skupe i kompleksne logike na čip, dizajneri Intela su primenili genijalno, ali potencijalno opasno rešenje: unifikovali su unutrašnji RAM. Umesto odvojenih memorijskih blokova, prvih 128 bajtova RAM-a podeljeno je na logičke zone gde u istoj zgradi žive radni registri, stek i opšta memorija.

U samom dnu memorije, od adrese 00H do 1FH, nalaze se četiri banke radnih registara (Banka 0, 1, 2 i 3). Svaka banka sadrži registre od R0 do R7. Kada napišete instrukciju u asembleru, mašinski kod u ROM-u je uvek isti i procesor uvek vidi komandu koja traži registar R0. Međutim, fizička lokacija tog registra zavisi od statusnog registra PSW (Program Status Word). Unutrašnji memorijski kontroler procesora očitava bitove iz PSW registra i u realnom vremenu primenjuje prostu hardversku formulu: pomnoži broj aktivne banke sa osam i sabere sa brojem registra. Ako je aktivna Banka 0, R0 je na adresi 00H. Ako prebacimo na Banku 1, čip automatski preusmerava žice na adresu 08H. To omogućava programeru da u prekidnoj rutini prosto promeni banku registara umesto da troši vreme na sporo guranje svih sedam radnih registara na stek.

Međutim, ovde dolazimo do fenomena koji možemo nazvati "stambena kriza u RAM-u". Kada se mikrokontroler upali, hardverski Stack Pointer (SP) se automatski inicijalizuje na adresu 07H. To znači da će prvi sledeći podatak koji se gurne na stek završiti na adresi 08H. Ako pogledamo mapu memorije, na adresi 08H počinje Banka 1 registara. Hardver je slep i ne poseduje nikakve zaštitne barijere. Stack Pointer je običan, jednostavan brojač koji pri svakom PUSH-u radi isključivo operaciju plus jedan. Ako programer odluči da u prekidnoj rutini koristi Banku 1, a prethodno nije ručno pomerio početak steka, desiće se katastrofa – stek će linearno rasti nagore i brutalno prebrisati registre sopstvene prekidne rutine.

Zbog ove specifičnosti arhitekture, prvo zlatno pravilo svakog optimizovanog 8051 asemblerskog programa jeste ručna inicijalizacija steka na samom startu koda. Katapultiranjem steka instrukcijom iznad kritične zone banaka i bit-adresibilnog prostora, programer obezbeđuje čist i bezbedan prostor za rast steka, dok bazu memorije ostavlja slobodnu za ultra-brzu manipulaciju registrima. Na 8-bitnim sistemima, hardver je jednostavan i predvidiv, a softver je taj koji mora posedovati svest o svakom pojedinačnom bajtu.



Organizacija RAM memorije

Ovaj tekst, kao i duboka konceptualna analiza arhitekture mikrokontrolera koja iza njega stoji, nastali su u saradnji sa veštačkom inteligencijom (AI Gemini) kao asistentom u učenju. Kroz dinamičan dijalog i postavljanje specifičnih inženerskih pitanja, AI se pokazao kao izuzetan oblik modernog digitalnog učitelja koji kompleksne apstraktne koncepte silicijumskog dizajna rastavlja na jednostavne, logične i lako razumljive celine.

📦 **Open Source & Podela znanja:** Ova digitalna knjiga, prateće slike i PDF dokumenti su licencirani pod Creative Commons Attribution 4.0 International (CC BY 4.0) licencom. Slobodno možete deliti, kopirati i prerađivati materijal u bilo koje svrhe, uključujući i komercijalne, pod uslovom da jasno navedete izvor: zilsel-invent.com.

💻 Ukoliko je uz ovaj članak priložen izvorni kod, on je licenciran pod MIT licencom. Kompletan pregled našeg višestrukog sistema licenciranja i preuzimanje LICENSE datoteke možete pronaći na našoj stranici: zilsel-invent.com/license-new.xhtml

Pod haubom PSW registra: Asemblerski vodič kroz banke registara na Intel 8051

U prethodnom uvodnom tekstu definisali smo logičku mapu unutrašnjeg RAM-a i dotakli se koncepta hardverskog preusmeravanja adresa pomoću dekodera. Da bismo razumeli kako procesor u realnom vremenu zna koju od četiri banke registara treba da koristi, moramo detaljno analizirati njegovu „kontrolnu tablu“ – statusni registar **PSW (Program Status Word)**.

PSW je 8-bitni registar specijalne namene (SFR) smešten na adresi D0H. Svaki bit u ovom registru ima jedinstvenu hardversku ulogu i direktno utiče na ponašanje procesora nakon aritmetičkih operacija ili pri donošenju odluka. Anatomija PSW registra izgleda ovako:

- **Bit 7 – CY (Carry Flag):** Koristi se kao akumulator za bit-operacije i beleži prenos (ili pozajmicu) iz najvišeg bita (Bit 7) prilikom aritmetičkih operacija.
- **Bit 6 – AC (Auxiliary Carry Flag):** Beleži prenos iz donjeg polu-bajta (bita 3 u bit 4), što je ključno za BCD (Binary-Coded Decimal) aritmetiku.
- **Bit 5 – F0 (Flag 0):** Opšti fleg koji je potpuno na raspolaganju programeru za bilo kakve logičke potrebe u softveru.
- **Bit 4 – RS1 (Register Bank Select 1):** Viši bit za odabir aktivne banke registara.
- **Bit 3 – RS0 (Register Bank Select 0):** Niži bit za odabir aktivne banke registara.
- **Bit 2 – OV (Overflow Flag):** Setuje se ako aritmetička operacija sa označenim brojevima (signed) napravi prekoračenje kapaciteta 8-bitnog registra.
- **Bit 1 – UD (User-Definable Flag):** Još jedan fleg slobodan za potrebe programera.
- **Bit 0 – P (Parity Flag):** Hardverski fleg koji uvek pokazuje paritet (paran/neparan broj jedinica) u Akumulatoru (A). Ako Akumulator ima neparan broj jedinica, P se postavlja na 1.

Za našu priču o optimizaciji najvažniji su bitovi **RS1** i **RS0**. Direktnom manipulacijom ovih bitova, programer naređuje unutrašnjem dekoderu memorije da promeni fizičke lokacije registara R0-R7, stvarajući savršen softverski izolovan prostor bez upotrebe steka.

Ovaj tekst, kao i duboka konceptualna analiza statusnih registara, nastali su u saradnji sa veštačkom inteligencijom (AI Gemini) kao asistentom u učenju, koji pomaže u strukturiranju i mapiranju hardverskih koncepata u razumljive celine.

Praktični deo

U ovom praktičnom primeru demonstriraćemo kako se u asembleru vrši inicijalizacija steka, upis podataka u podrazumevanu Banku 0, a zatim prebacivanje u Banku 1 pomoću direktnog gađanja bitova u

PSW registru. Primetićete da registar R0 u različitim bankama čuva potpuno različite vrednosti jer hardver preusmerava podatke na različite fizičke adrese u RAM-u.

```
;=====================================================================
 DEMONSTRACIJA MANIPULACIJE BANKE PREKO PSW REGISTRA
;=====================================================================

ORG 0000H              ; Reset vektor - odavde mikrokontroler pocinje rad
LJMP START             ; Dug skok na glavni program

ORG 0030H              ; Pocetak glavnog programa (zaobilazimo prekide)
START:
    ; 1. ZLATNO PRAVILO: Inicijalizacija steka iznad kriticnih zona
    MOV SP, #2FH       ; Stack Pointer postavljen na 2FH, stek raste od 30H
navise

    ; --- RAD U PODRAZUMEVANOJ BANKI 0 (RS1=0, RS0=0) ---
    MOV R0, #55H       ; Upisi vrednost 55H u R0 (Banka 0, adresa 00H)
    MOV A, R0          ; Prebaci vrednost iz R0 u Akumulator (A = 55H)

    ; --- MAGICNI PRELAZAK U BANKU 1 (RS1=0, RS0=1) ---
    SETB RS0           ; Postavljamo bit RS0 na 1 (RS1 ostaje 0)

    ; --- RAD U BANKI 1 ---
    MOV R0, #AAH       ; Upisi vrednost AAH u R0 (Banka 1, adresa 08H)

    ; --- POVRATAK U BANKU 0 ---
    CLR RS0            ; Resetujemo bit RS0 na 0. Vracamo se u Banku 0.
    MOV A, R0          ; Ponovo citamo R0. Akumulator ponovo dobija 55H!

    SJMP $             ; Beskonacna petlja (kraj programa)
END
```

📌 **Inženjerska napomena: Zašto „skačemo sami na sebe“?**

Za razliku od programa na računaru koji po završetku vraćaju kontrolu operativnom sistemu, mikrokontroler nema operativni sistem i **nikada ne sme da prestane sa radom**.

Znak $ u asembleru predstavlja trenutnu adresu na kojoj se procesor nalazi. Instrukcija SJMP $ (Short Jump) bukvalno naređuje procesoru da **skoči sam na sebe**, ostajući zarobljen u beskonačnoj petlji na istom mestu. Bez ove linije, Program Counter (PC) bi nastavio da linearno čita prazan prostor u ROM memoriji (đubre od bajtova), što bi dovelo do nepredvidivog „divljanja“ procesora (engl. *runaway code*). Ovo je standardan način za bezbedno „parkiranje“ mikrokontrolera na kraju programa.

Matematika brzine: Zašto registri pobeđuju varijable u memoriji na Intel 8051

U prva dva poglavlja rešili smo stambenu krizu u RAM-u i prošli kroz anatomiju PSW registra. Međutim, prava magija optimizacije i razlog zašto smo toliko pažnje posvetili bankama registara leži u **brzini izvršavanja instrukcija**. Na 8-bitnim sistemima, lokacija na kojoj čuvate podatke tokom proračuna direktno diktira koliko će taktova kvarcnog oscilatora procesor potrošiti.

Fizičko pravilo arhitekture Intel 8051 je prosto: **rad direktno unutar registara je duplo brži od rada sa varijablama u memoriji.**

Jedan mašinski ciklus (Machine Cycle) na bazičnom 8051 procesoru traje tačno **12 taktova (oscilacija) kvarcnog oscilatora**. Kada procesor izvršava operaciju nad registrom (npr. `R0` ili Akumulator `A`), ALU (aritmetičko-logička jedinica) ima direktan hardverski pristup podacima. Instrukcija zauzima samo 1 bajt u ROM-u i završava se u roku od **1 mašinskog ciklusa (12 oscilacija)**.

Sa druge strane, kada podatke držite u varijablama unutar opšteg RAM-a, procesor mora da prođe kroz složeniji postupak. Instrukcija sada zauzima 2 bajta u ROM-u (kod instrukcije + adresa varijable). U prvom ciklusu procesor čita samu instrukciju i adresu, a u drugom ciklusu memorijski kontroler tek odlazi do RAM-a, uzima podatak i donosi ga na obradu. Rezultat? Potrošena su **2 mašinska ciklusa (24 oscilacije)**. Vreme je duplirano.

Još gora situacija nastaje kada izađemo u spoljnu memoriju (**XDATA**). Tamo instrukcije (poput `MOVX`) troše još više ciklusa zbog spore komunikacije preko spoljnih hardverskih linija. Zato vrhunska optimizacija nalaže: ***držite sve aktivne proračune isključivo u registrima `R0–R7`, a u memoriju upisujte tek konačan rezultat.***

Vremenski dijagram (Poređenje clock oscilacija - potrošnju taktova kvarca za istu operaciju sabiranja)

RAD SA REGISTROM (ADD A, R0)
[Ciklus 1: Citanje i izvrsavanje odjednom]
Taktovi: 1 2 3 4 5 6 7 8 9 10 11 12 ---> UKUPNO: 12 oscilacija (KRAJ)

RAD SA VARIJABLOM U RAM-U (ADD A, 30H)
[Ciklus 1: Citanje adrese] [Ciklus 2: Pristup RAM-u i sabiranje]
Taktovi: 1 2 3 4 5 6 7 8 9 10 11 12 | 13 14 15 16 17 18 19 20 21 22 23 24 ---> UKUPNO: 24 oscilacije (KRAJ)

Praktični deo

U ovom delu upoređujemo dve identične logičke operacije. Prva koristi varijablu u memoriji na adresi `30H`, dok druga koristi isključivo radne registre. Obratite pažnju na komentare o potrošnji mašinskih ciklusa:

```
;=====================================================================
 POREDJENJE BRZINE - REGISTRI PROTIV MEMORIJE
;=====================================================================

ORG 0000H
LJMP START

ORG 0030H
START:
    ; Priprema steka i test podataka
    MOV SP, #2FH
    MOV 30H, #05H      ; Varijabla na RAM adresi 30H ima vrednost 5
    MOV R0, #05H       ; Registar R0 takodje ima vrednost 5
    MOV A, #10H        ; U Akumulator stavljamo pocetnih 16 (10H)

    ; --- SPORIJI PRISTUP: Rad sa varijablom u memoriji ---
    ADD A, 30H         ; Saberi A sa sadrzajem adrese 30H
                       ; [Zauzima 2 bajta u ROM-u, trose se 2 masinska
ciklusa / 24 takta!]

    ; --- BRZI PRISTUP: Rad direktno unutar registra ---
    ADD A, R0          ; Saberi A sa sadrzajem registra R0
                       ; [Zauzima samo 1 bajt u ROM-u, trosi se 1 masinski
ciklus / 12 taktova!]

    SJMP $             ; Bezbedno parkiranje procesora
END
```

O knjizi: **Diskusija koja je prerasla u digitalnu knjigu**

Ova knjiga predstavlja destilovanu inženjersku diskusiju između autora (inženjera) i veštačke inteligencije (AI Gemini) oko najsitnijih detalja arhitekture 8051. Cilj nam je bio da kroz kratke, praktične dijaloge skinemo veo apstrakcije sa 8-bitnih sistema. Kao rezultat nastala je digitalna knjiga **"Intel 8051 disasembliran: Arhitektura stanja i Paralelizma"**.

Anatomija takta — Kako mikrosekvener upravlja vremenom

Nastavljam dalje sa arhitekturom procesora 8051, a ovog puta fokus je na anatomiji takta i tome kako CPU izvršava instrukcije skladištene u ROM memoriji. Ovo je koncept koji meni lično dugo nije bio potpuno jasan, ali predstavlja kritično znanje koje možete direktno primeniti u različitim oblastima – od mobilne robotike i embedded sistema, pa sve do duboke optimizacije performansi softvera. U svojim ranijim radovima dosta sam se bavio deep-level optimizacijom za Microsoft SQL Server, gde se kroz korišćenje specifičnih CPU ekstenzija i podešavanje koda performanse podižu na maksimum, a razumevanje procesorskog takta je koren te priče.

Kao i u prethodnim tekstovima iz ovog serijala, moram da napomenem da je AI izuzetan asistent za diskusiju, učenje i razjašnjavanje kompleksnih hardverskih koncepata na koje ne možete lako naići u standardnoj literaturi. Ovom prilikom, spuštamo se duboko pod haubu samog CPU jezgra da vidimo kako se instrukcija zapravo izvršava.

Procesor svaku instrukciju izvršava kroz tri bazične faze: **FETCH** (dovlačenje), **DECODE** (dekodiranje) i **EXECUTE** (izvršavanje). Tokom FETCH faze, procesor doprema instrukciju iz ROM memorije u kojoj je skladišten program. Ta instrukcija je zapravo *OpCode* (operacioni kod) koji se šalje direktno na instrukcijski dekoder. DECODER ima zadatak da prepozna taj kod i pokrene, odnosno inicira, precizno dirigovani niz unutrašnjih *mikro-operacija*.

Ovde je ključno povući razliku između vašeg asemblerskog programa koji ste flešovali u ROM i mikro-programa koji se nalazi u internoj memoriji same upravljačke jedinice (mikrosekvencera). Ta upravljačka jedinica je vrhovni dirigent koji upravlja kompletnim hardverom, a ritam joj zadaje spoljni kvarcni oscilator od **12 MHz**. Svaka instrukcija u ROM-u je za dekoder samo okidač koji mu govori koji unutrašnji mikro-program treba da pokrene.

U teoriji digitalnih sistema često se pominje da procesori implementiraju dva ključna koncepta: *stek mašinu* i *state mašinu* (automat stanja). Međutim, ako posmatrate samo suvoparnu školsku definiciju state mašina, možete upasti u zabludu u kojoj sam i sam bio. Definicija kaže da je naredno stanje sistema funkcija trenutnog stanja i spoljnih uticaja, i da se prelazak dešava tokom svake periode takta.

Iz ovoga se lako može pogrešno zaključiti da je jedno stanje automata vezano za samo jedan jedini impuls (herc) kvarca, što u praksi nije tačno. Jedno logičko stanje procesora može trajati više perioda kvarcnog oscilatora, a upravljačka jedinica unutar sebe ima hardverski brojač koji tačno zna koliko taktova pripada kom stanju.

Intel je arhitekturu 8051 organizovao u šest fiksnih stanja: **S1, S2, S3, S4, S5 i S6**:

- **S1** je rezervisano za **FETCH** stanje.
- **S2** je rezervisano za **DECODE** stanje.
- **S3 do S6** odlaze na fazu izvršavanja (**EXECUTE**).

Sva ova stanja su upakovana u ritam kvarcnog oscilatora od 12 MHz. Upravljačka jedinica svakom stanju fiksno dodeljuje **dva takta kvarca** (dve unutrašnje faze, P1 i P2). Pošto imamo 6 stanja, a svako troši po 2 takta, dolazimo do matematike od tačno **12 oscilacija takta za jedan mašinski ciklus**. Pri frekvenciji od 12 MHz, jedna perioda takta traje oko **83 ns**. Kada to pomnožimo sa 12, dobijamo tačno **996 ns**, što je u praksi **1 mikrosekunda** za izvršavanje osnovnog mašinskog ciklusa.

Ovde treba naglasiti da proste instrukcije (poput INC R0) završavaju svoj posao unutar ALU jedinice mnogo brže i nisu im potrebna sva četiri stanja izvršavanja (S3-S6). Zato originalno 8051 jezgro radi jednu genijalnu optimizaciju: već tokom stanja S5 i S6 ono započinje sledeću FETCH fazu i unapred dovlači naredni bajt iz memorije. Kod kompleksnih instrukcija kao što je množenje (MUL AB), mikro-

program je predugačak za jedan ciklus, pa dekoder "zaključava" procesor kroz ukupno 4 uzastopna mašinska ciklusa (48 oscilacija).

Ako se pitate zašto je Intel izabrao baš ovu komplikovanu podelu sa 12 taktova, odgovor leži u fizici silicijuma iz 1980. godine. Tadašnji čipovi su rađeni u NMOS tehnologiji i bili su dinamičkog dizajna – koristili su parazitne kapacitivnosti (sićušne unutrašnje kondenzatore) da zadrže stanja podataka. Da bi ti podaci ostali stabilni i da ne bi "izvetrili", čip je morao konstantno da se osvežava visokom frekvencijom.

S druge strane, dve oscilacije unutar FETCH ili DECODE stanja su bile čista elektrotehnička neminovnost za sporu elektroniku tog vremena. Pogledajmo šta mikrosekvener zapravo radi tokom ta dva takta FETCH faze (stanje S1):

1. **Prvi takt (Faza P1):** Na uzlaznu ivicu takta, upravljačka jedinica uzima adresu iz Program Counter-a (PC) i postavlja je na adresnu magistralu koja vodi do ROM memorije. Međutim, ROM ne može u istoj nanosekundi da vrati podatak nazad. Mora se sačekati da prođe prelazni režim. Iz elektrotehnike znamo kako izgleda radna prava i pomeranje radne tačke tranzistora kada preklapa iz logičke nule u jedinicu. Tih 83 ns u prvoj fazi služe isključivo da se napon na žicama magistrale fizički smiri i stabilizuje. Signal je stigao na ulaz dekodera, ali dekoder ga još uvek ne čita.
2. **Drugi takt (Faza P2):** Tek kada stigne sledeća uzlazna ivica takta, napon je potpuno stabilan. Mikrosekvener tada daje komandu unutrašnjem Instrukcijskom Registru (IR) da bezbedno "usisa" i zaključa taj stabilan bajt iz ROM-a. FETCH je gotov.

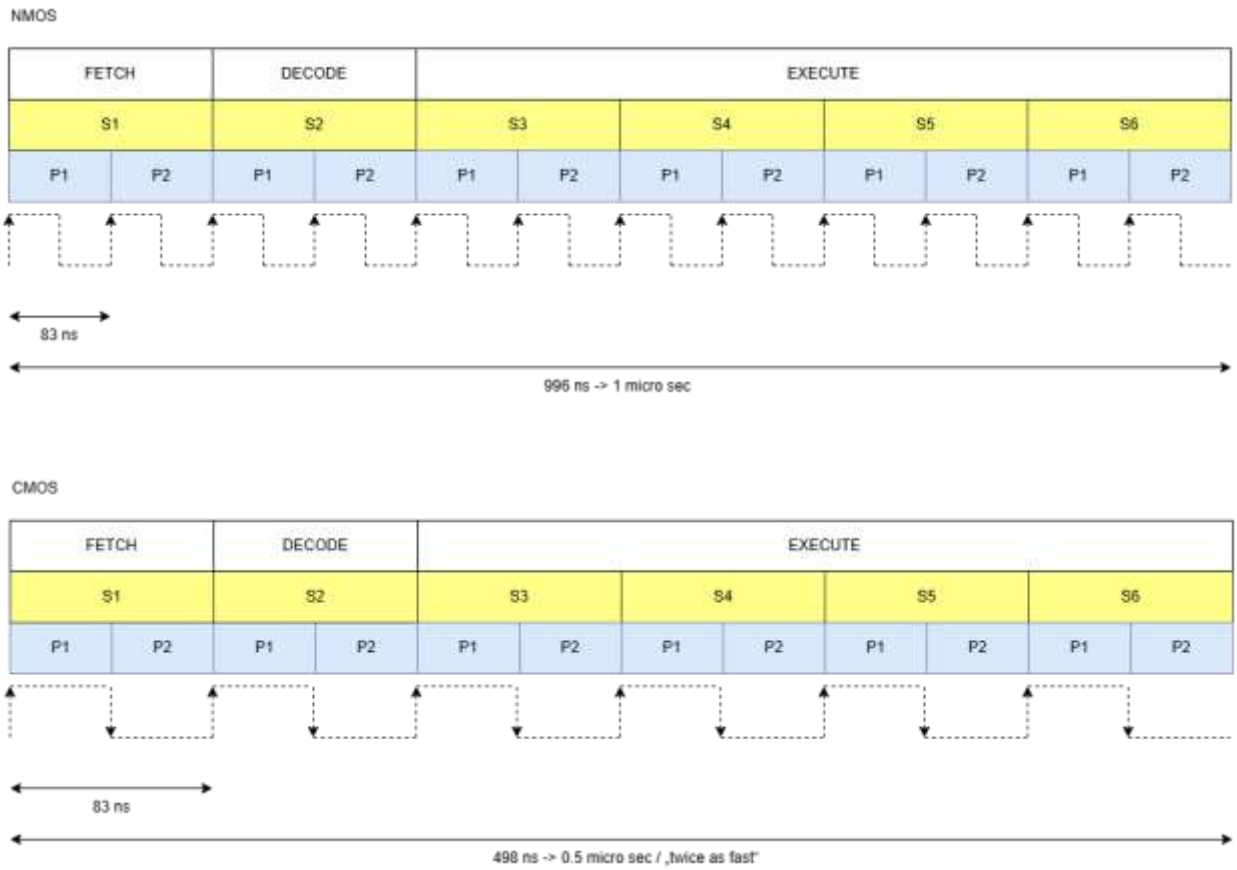
Ista logika važi i za stanje **DECODE (S2)**. U prvoj fazi (P1), zaključani OpCode se propušta kroz logičku matricu dekodera. Tranzistori unutar dekodera menjaju svoja stanja, struja prolazi kroz AND/OR kola i traži se stabilna putanja koja odgovara toj instrukciji. Tek u drugoj fazi (P2), kada su izlazni signali iz dekodera stabilni, mikrosekvener podiže prve kontrolne linije i otvara kapije registara koji treba da pošalju podatke ka ALU jedinici.

Fizička cena takta: Zašto instrukcije troše struju?

Ovakva razlika u broju ciklusa otvara još jedno ključno inženjersko pitanje koje se često zanemaruje – potrošnju energije. U svetu silicijuma, svaki herc takta i svako preklapanje tranzistora ima svoju cenu u miliamperima. Dok INC R0 završi svoj posao brzo uz minimalno pomeranje unutrašnjih napona, MUL AB primorava stotine logičkih kapija unutar ALU jedinice i mikrosekvencera da divlje menjaju stanja (iz 0 u 1) kroz čitavih 48 taktova. Što duže instrukcija izvršava svoj unutrašnji mikro-program, to čip povlači više struje i više se greje. Na baterijski napajanim sistemima, poput mobilnih robota, ovo je razlika koja direktno utiče na autonomiju rada. Zato je zlatno pravilo optimizacije uvek bilo: **drži podatke u ultra-brzim registrima i koristi što jednostavnije operacije.**

Upravo je ova spoznaja pokrenula tehnološku revoluciju i otvorila vrata **RISC (Reduced Instruction Set Computer)** arhitekturi koja je nastala godinama kasnije. Inženjeri su shvatili da su teške i trome CISC instrukcije (kakve delom ima i 8051) energetski neefikasne jer zahtevaju prevelike i kompleksne mikro-programe u dekoderu. RISC je preokrenuo igru – izbacio je teške instrukcije, sve operacije sveo na proste korake od po jedan takt i drastično uvećao broj unutrašnjih registara. Izbacivanjem mikrokoda i fokusom na registre, RISC procesori (poput ARM arhitekture u vašim telefonima) postali su ekstremno štedljivi i brzi.

Tek decenijama kasnije, kada je silicijum postao drastično brži, razvijene su moderne CMOS verzije 8051 procesora koje ceo FETCH izvršavaju u samo jednoj periodi takta (korišćenjem i uzlazne i silazne ivice), čime pokrivaju ove dve mikro-operacije u duplo kraćem vremenu. Rad na simulatorima, sa druge strane, u potpunosti eliminiše fiziku korišćenog silicijuma, te je takt kvarcnog oscilatora u tom okruženju čista apstraktna stvar. Razumevanje ovog hardverskog determinizma nam pokazuje zašto je na stvarnom, starom hardveru merenje performansi bilo čista matematika, dok na modernim sistemima softverski nivoi apstrakcije kriju ovaj fascinantni ples elektronike.



Podela mašinskog ciklusa na stanja i unutrašnje faze P1/P2 u arhitekturi Intel 8051

Zašto je sve ovo važno za moderan PC?

Na kraju, arhitekturu 8051 je izuzetno korisno razumeti jer je ona čistokrvni **CISC procesor** koji je po svom osnovnom konceptu veoma sličan **x86 arhitekturi** u našim računarima. Iako moderni PC procesori rade na nezamislivo višim frekvencijama i imaju složene optimizacije, bazični principi mikrokoda, dekodiranja složenih naredbi i upravljanja magistralama su strukturalno ostali isti. Koncepte koje savladate i razumete na primeru 8051, možete sa lakoćom prepoznati, analizirati i upotrebiti kada se bavite optimizacijom i dubokim razumevanjem rada modernih x86 sistema i velikih softverskih platformi.

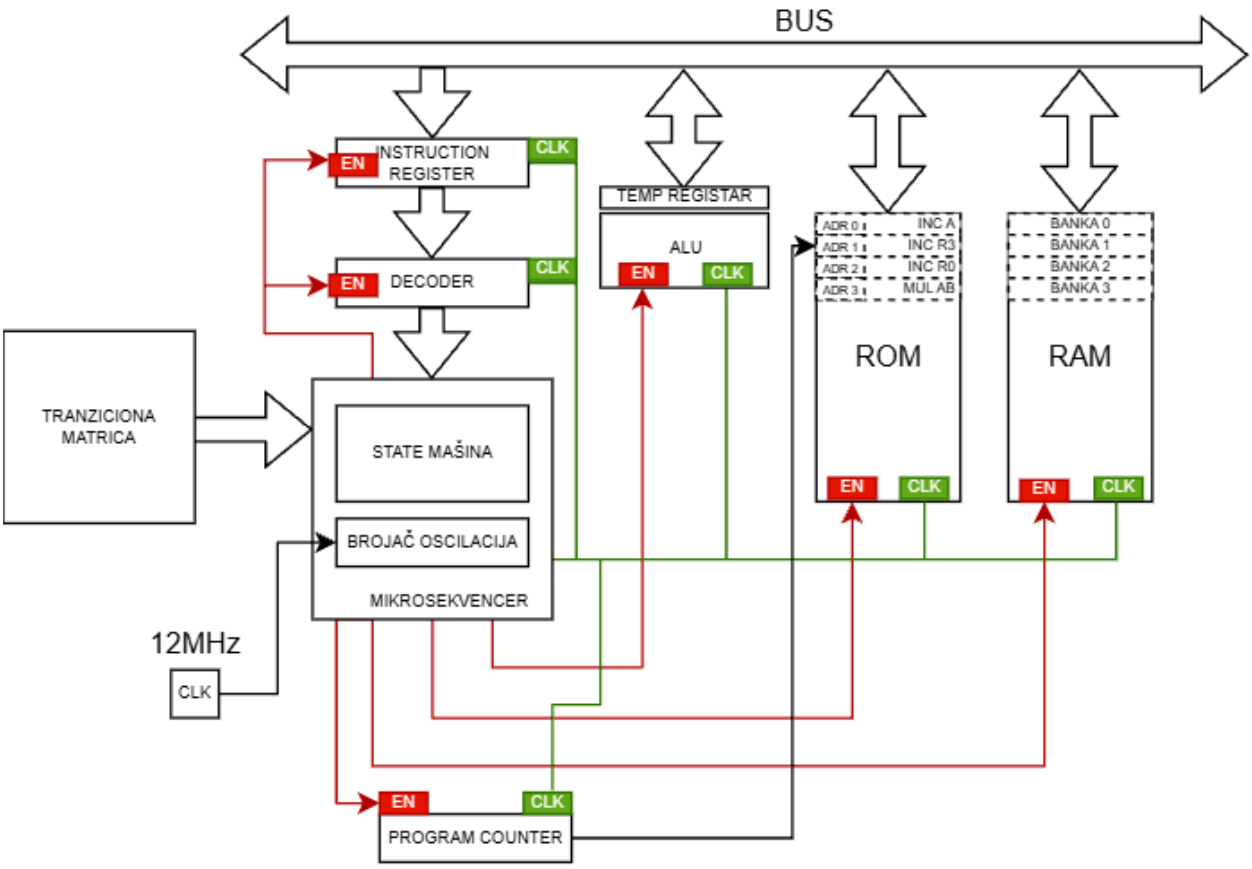
Ubrizgavanje stanja — Kako softverska matrica postaje silicijumski mikrokod

Da bi se zaista razumeo mikrosekvencer i šta zapravo znači "ubrizgavanje ponašanja" u sistem (odnosno ubrizgavanje stanja), moramo da zaronimo duboko u hardver procesora i vidimo kako je on organizovan u svom najskrivenijem kutku.

Zamislite procesor kao simfonijski orkestar. Ako truba, violina i bubanj krenu da sviraju istovremeno bez ikakvog reda, nastaje potpuni haos. U svetu silicijuma taj haos se zove kolizija podataka.

Orkestracija deljene magistrale

Pogledajmo blok dijagram osnovnih delova arhitekture našeg CPU-a. Sačinjavaju ga programski brojač (Program Counter), instrukcijski registar (Instruction Register), dekoder (Decoder), ALU sa svojim privremenim registrom, ROM, RAM, upravljačka jedinica (mikrosekvencer) i kristalni oscilator od **12MHz**.



Blok dijagram deljene magistrale.

Magistrala (BUS) između svih ovih jedinica je deljena. To znači da se u svakom trenutku mora striktno voditi računa o tome koja jedinica postavlja podatak na nju. Ako dve jedinice istovremeno pokušaju da pošalju podatke na magistralu, dolazi do mešanja signala i hardverskog konflikta.

Da bi se postigla savršena orkestracija, svaka jedinica unutar CPU-a ima svoje upravljačke i komandne signale:

- **EN (Enable):** Upravljački signal koji "otkačinje" ili "prikačinje" jedinicu na deljenu magistralu.
- **CLK (Clock):** Taktni signal (pravougaoni okidajući impuls iz kristalnog oscilatora) koji omogućava promenu mikro-operacije i prelazak u sledeće stanje.

Upravljačka jedinica je dirigent ovog orkestra. Ona na osnovu **tranzicione matrice** i taktnog signala upravlja izvršavanjem instrukcija. Implementirana je kao **State Mašina** koja poseduje definisanu mikroprogramsku matricu – matricu prelaza iz stanja u stanje. Izbor sledeće mikro-operacije zavisi od trenutnog stanja, spoljnih faktora ili je prosto unapred poznat, kao što je to slučaj u FETCH i DECODE fazama.

Tranziciona matrica: Anatomija jedne instrukcije

Upravljačka reč (na dijagramu označena crvenim linijama za `EN` i zelenim za `CLK`) u svakom polutaktu tačno definiše ko sme da priča, a ko mora da ćuti.

Hajde da vidimo kako tranziciona matrica upravlja hardverom kroz faze **FETCH** (preuzimanje), **DECODE** (dekodiranje) i **EXECUTE** (izvršavanje) na primeru instrukcije `INC R3` (uvećaj vrednost registra R3 za jedan):

Faza / Stanje	Takt	Enable PC	Enable IR	Enable Decoder	Enable ALU	Enable ROM	Enable RAM	Opis operacije (Šta se dešava u silicijumu)
S1 - FETCH	**P1**	**1**	0	0	0	0	0	Programski brojač (PC) postavlja adresu instrukcije na magistralu.
	P2	0	**1**	0	0	**1**	0	ROM postavlja OpCode na magistralu, a Instruction Register (IR) ga upisuje.
S2 - DECODE	**P1**	0	**1**	**1**	0	0	0	IR drži stabilan OpCode, a Decoder se aktivira da prepozna instrukciju INC R3.
	P2	0	0	0	**1**	0	**1**	Vrednost registra R3 se povlači iz RAM-a i smešta u privremeni (TEMP) registar ALU-a.
S3 - EXECUTE	**P1**	0	0	0	**1**	0	0	ALU vrši internu operaciju uvećanja vrednosti za 1.
	P2	**1**	0	0	**1**	0	**1**	ALU vraća novi rezultat nazad u RAM (u R3), a PC se postavlja za sledeću instrukciju.

U FETCH fazi, u taktu P1, mikrosekvencer mora da otkači sve jedinice sa magistrale osim programskog brojača (PC) koji postavlja adresu. Već u sledećem polutaktu (P2), PC se isključuje, a aktiviraju se ROM (koji drži instrukciju na toj adresi) i Instruction Register (IR) koji prihvata taj podatak. Čeka se tačno definisano vreme (kod 12MHz kristala jedna oscilacija traje oko 83 ns) i mikrosekvencer, vođen tranzicionom matricom, automatski prelazi na fazu dekodiranja, a zatim i na samo izvršavanje instrukcije u ALU jedinici.

Kao što vidite, tranziciona matrica zna da bude izuzetno složena. Ona mora da sinhronizuje hardverske celine u nanosekundu i da striktno pazi na deljenje magistrale, iako na prvi pogled sve deluje jednostavno.

Moj projekat: Injection System Behavior (Ubrizgavanje ponašanja)

Razmišljajući o ovom konceptu pre više godina, započeo sam projekat koji se bavi upravo matricama prelaza stanja na višem nivou. Nazvao sam ovu metodu **"Injection System Behavior"** (Ubrizgavanje ponašanja u sistem).

U klasičnom programiranju mikrokontrolera, programeri često pišu beskonačne i krute `if-else` strukture ili džinovske `switch-case` blokove kako bi kodirali State Mašinu. Ako sutra želite da promenite logiku rada uređaja, morate ponovo da menjate izvorni kod, kompajlirate i flešujete čip.

Ja sam rešio da primenim logiku mikrokoda iz procesora. Napravio sam **generičku state mašinu** koja je potpuno "slepa" – ona sama po sebi ne zna šta treba da radi. Međutim, njeno ponašanje se definiše spolja, kroz tranzicionu matricu koja se **ubrizgava** u memoriju sistema kao čist podatak.

U praksi sam to demonstrirao kroz kontrolu koračnog (stepper) motora na 8051 mikrokontroleru. Kreirao sam nekoliko različitih tranzicionih matrica (za različite režime rada, brzine i sekvence koraka). Kada u moju generičku state mašinu "ubrizgam" matricu A, motor se vrti na jedan način. Kada softverski ubrizgam matricu B, motor trenutno menja ponašanje i izvršava potpuno novu logiku – **bez izmene ijedne jedine linije izvršnog koda same mašine!**

Izvorni kod i kompletan projekat su dostupni na sledećoj adresi:
🔗 Injection System Behavior State Machine na LibStock-u

Istorijski most: Kako Intel "krpi" procesore preko RAM-a

Ono što je najfascinantnije u celoj ovoj priči jeste to što je moj softverski eksperiment sa ubrizgavanjem matrica zapravo identičan konceptu koji komercijalni giganti poput Intela i AMD-a koriste već decenijama na x86 arhitekturama.

1. **Intel 8086 (1978):** Bio je to prelomni trenutak. Umesto da komplikovanu logiku procesora projektuju isključivo ožičenim hardverskim vezama (hardwired logic), inženjeri su ubacili mikrokod – tranzicionu matricu u interni ROM procesora, širine 21 bit, koja je upravljala unutrašnjim jedinicama kroz oko 512 koraka.
2. **Intel Pentium Pro i AMD K5 (sredina 90-ih):** Doneli su revoluciju. Ovi procesori su postali "prevodioci". Spolja primaju kompleksne, kabaste x86 instrukcije (CISC arhitektura), ali unutrašnji dekoder (baš kao na našoj šemi) te instrukcije "bode" u tranzicionu matricu i cepka ih na proste, ultra-brze mikrooperacije (µops) koje izvršava interna RISC mašina.
3. **Krpljenje procesora "u letu" (Microcode Updates):** Šta se dešava ako inženjeri u Intelu naprave grešku (bug) u dizajnu silicijuma koja se otkrije tek kada su milioni procesora već u računarima korisnika? Povlačenje čipova bi koštalo milijarde. Rešenje? **Ubrizgavanje stanja!**
4. Najpoznatiji primer "ubrizgavanja" nove matrice desio se kod Intel Pentium f00f bug-a i kasnijih Spectre/Meltdown ranjivosti. Još jedan primer: ukoliko na silicijumu iz nekog razloga ne radi FPU jedinica (floating point hardware, realni brojevi), ubrizgavanjem mikrokoda FPU hardware-ski se zameni SOFT FLOAT algoritmima koje izvršava celobrojna ALU jedinica. To možete zaključiti posmatranjem tranzicione matrice, **prosto ugasite sa logičko 0 deo silicijuma koji ima grešku**. Na isti način, ukoliko inženjeri ustanove da mogu ubrzati izvršavanje instrukcija (optimizacija), isto dobijate kroz mikrokod update.

Moderni x86 procesori pored fabričkog mikrokoda u ROM-u poseduju i mali, brzi, namenski SRAM (RAM unutar samog jezgra). Kada upalite računar, vaš BIOS ili operativni sistem (Windows/Linux) u prvim milisekundama podizanja sistema povlači fajl koji se zove **Microcode Update** i bukvalno ga **ubrizgava** u taj SRAM procesora. Nova tranziciona matrica prepisuje stare, falične fabričke korake, preusmerava signale i hardverski problem biva rešen – čistim softverskim ubrizgavanjem podataka.

Povezivanjem ovih tačaka shvatamo najveću istinu računarske arhitekture: **granica između hardvera i softvera ne postoji.** Sve je to samo matrica stanja koja čeka svoj sledeći takt.

🛠️ Iza kulisa: Kako nastaje mikrokod i saradnja sa AI

*Tranziciona matrica iskristalisana je kroz inženjersku diskusiju sa **AI Gemini asistentom**.*

Tokom rada na ovom tekstu, otvorili smo zanimljivo pitanje: **Kako inženjeri u Intelu ili AMD-u dizajniraju ove kompleksne matrice i njihove zakrpe (Microcode Updates)? Da li ručno ređaju nule i jedinice?**

Naravno da ne — proces je visoko automatizovan kroz tri ključna koraka:

- **HDL i Mikro-asembleri:** Hardver i mikrokod se pišu u jezicima poput *SystemVerilog-a*. Inženjeri pišu logičke komande (npr. prebaci podatak u ALU, aktiviraj sabiranje), a specijalizovani softverski *mikro-asembleri* to automatski prevode u džinovske upravljačke reči od nekoliko stotina bitova (nule i jedinice).
- **Match & Patch mehanizam (Kako radi Update):** Kada se uoči bag u procesoru, inženjeri ne menjaju fabrički ROM. Zakrpa koja se pri *boot-ovanju* ubrizga u SRAM procesora postavlja hardversku "zamku". Kada god procesor u radu naiđe na problematičnu fabričku adresu (*Match*), upravljačka jedinica trenutno preusmeri izvršavanje na novu, zakrpljenu matricu u SRAM-u (*Patch*).
- **Matematička verifikacija:** Pre nego što se zakrpa pusti u javnost, specijalni alati za *Formalnu verifikaciju* matematički dokazuju da u novoj tranzicionoj matrici ne postoji nijedna kombinacija ulaza koja može dovesti do kolizije na magistrali ili blokade procesora.

Ukratko, inženjeri postavljaju arhitekturu i logiku, a softverski alati rade prljav posao slaganja bitova. Baš kao što smo Gemini i ja uradili za ovu digitalnu knjigu!

Hardverska linija fronta — SFR zona, IRQ i Arhitektura prekida

1. Računar vs. Računarski sistem (Diskusija autora i Gemini AI)

- **Koncept:** Kroz analizu i dijalog koji vodim sa **Gemini AI**, došli smo do jasne definicije koja često buni početnike: računar u svom najčistijem obliku čine samo **procesor i memorija**, bez obzira na organizaciju. Međutim, sam računar ne može mnogo bez periferija. Čak je i najobičniji digitron zapravo **računarski sistem**, jer pored procesora i memorije ima tastaturu i displej.
- **Most ka PC-ju:** Kada procesoru i memoriji dodate magistralu i periferne uređaje (senzore, serijske portove, ili kod PC-ja miša, tastaturu i grafičku kartu), dobijate računarski sistem. **Intel 8051 je System-on-Chip (SoC)** koji na istoj silicijumskih pločici ima integrisane ove interne "čipove" (tajmere, serijsku komunikaciju), dok sve ostalo (spoljni RAM, ADC konvertori, senzori robota) dolazi spolja.

2. Zašto nam treba IRQ? Borba za magistralu i sprečavanje "đubreta"

- **Koncept:** Kada imate mnogo uređaja, svi se oni grabe za resurse sistema (memoriju i magistralu). Iz teksta o mikrosekvenceru znamo da **uređaji ne smeju da postavljaju podatke na magistralu kada se njima prohte**.
- **Inženjerska logika:** Ako serijski kontroler primi bajt sa spoljne veze, on ne može samoinicijativno da ga gurne na magistralu podataka dok procesor npr. upisuje nešto drugo u RAM. Došlo bi do kolizije i na magistrali bismo dobili **čisto đubre**. Zato je uveden **IRQ (Interrupt Request)** — zahtev procesoru da privremeno prekine trenutni tok programa i prebaci se na prekidnu rutinu (ISR) koja će bezbedno prebaciti taj bajt iz hardverskog bafera u RAM.

3. Gde živi SFR i kako upravlja tajmerom?

- **Koncept: SFR (Special Function Registers)** zona se nalazi u internoj memoriji procesora (80H - FFH). Ono što je genijalno kod 8051 jeste da se ključni registri za prekide nalaze u **bit-adresibilnoj zoni**. Svaki registar čija se adresa završava na 0 ili 8 (kao TCON na 88H ili IE na A8H) omogućava procesoru da direktno pristupi svakom bitu pojedinačno.
- **Kako to radi silicijum:**
  Kada se desi neki hardverski događaj (npr. tajmer premota brojač), sam hardver **automatski upisuje logičku 1 u tačno određen bit** u SFR zoni (za Tajmer 0 to je TF0, odnosno bit TCON.5). Pošto je zona bit-adresibilna, mikrosekvener na kraju instrukcije munjevito, u jednom

koraku, proverava status ovog bita. Ako je TF0 == 1, procesor privremeno maskira prekide nižeg prioriteta, čuva trenutno stanje na steku i skače na prekidnu adresu 000BH gde se nalazi kod naše prekidne rutine.

4. Mit o hardverskom multithreading-u

- **Koncept:** Mnogi programeri imaju glavobolje oko ovoga na početku. Ali istina je jasna: **multithreading je softver, a ne hardver!** [1] Procesor je u osnovi glupa sprava koja ne zna ništa o nitovima ili arhitekturi *Context Switch-a*.

Most ka PC-ju: Broj jezgara na modernim x86 procesorima su prosto **skalirani resursi**. Dva jezgra na silicijumu teoretski dupliraju brzinu i olakšavaju paralelizaciju, ali princip rada sa nitima je softverski algoritam. Pošto Intel 8051 ima procesorsko jezgro i genijalan sistem banki registara u RAM memoriji, on je **savršen i 100% legitiman poligon za implementaciju mikro-threading algoritma** koji ćemo demonstrirati u sledećem tekstu.

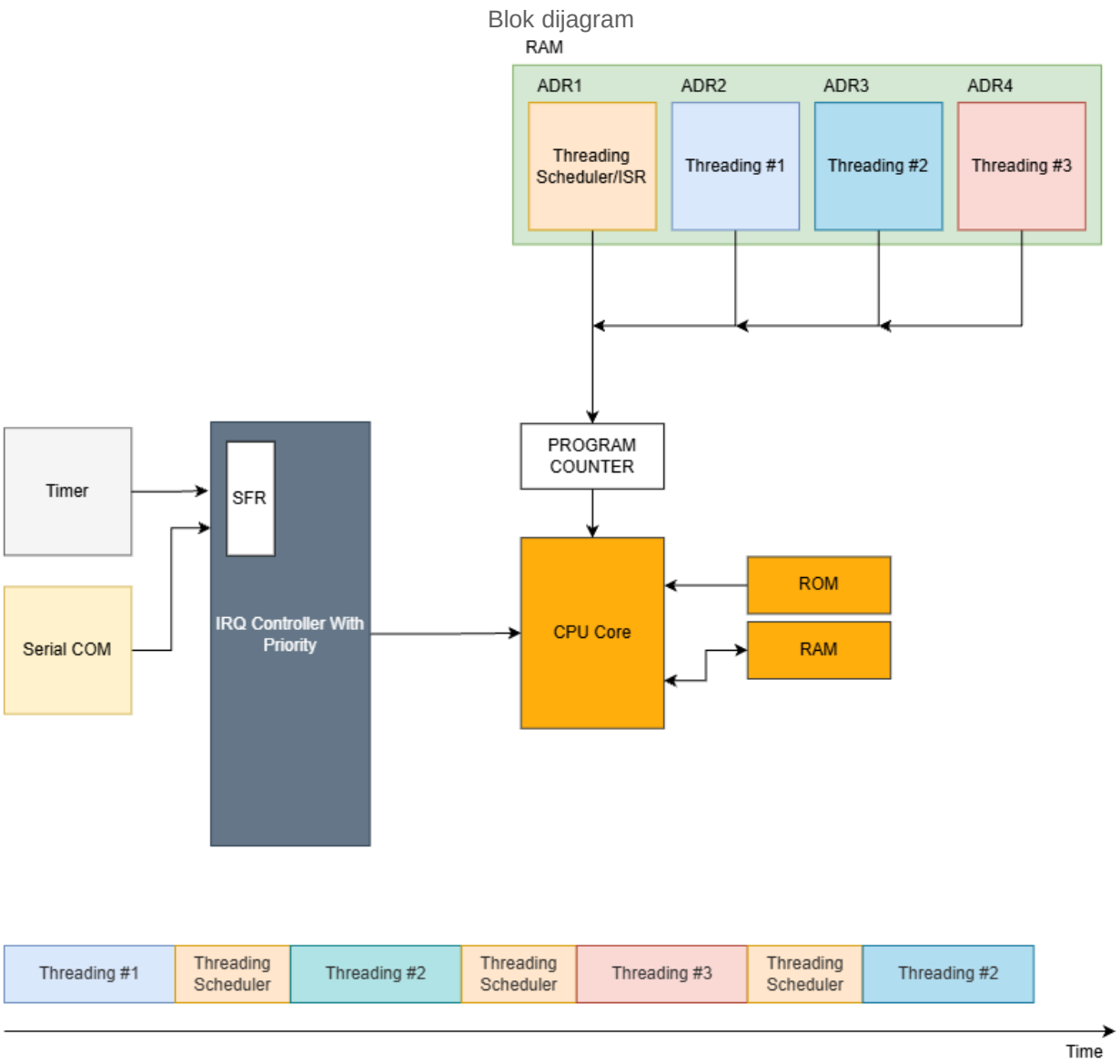
RTOS i Multithreading na 8-bitnom čipu — Trik sa tajmerom i bankama

Pisanje operativnog sistema u malom (micro-OS) za legendarne 8-bitne mikrokontrolere kao što je Intel 8051 predstavlja odličan način da razumete kako multitasking zapravo funkcioniše na nivou samog silicijuma. U ovom tutorijalu analiziramo kompletan i proveren Assembly kod koji omogućava paralelni rad **tri nezavisne mikro-niti (Thread A, B i C)** pomoću vremenskog prekidnog kvanta (Timer 0).

Apsolutni ključ za razumevanje ovog koda leži u preciznoj organizaciji RAM memorije i razumevanju kako procesor interno manipuliše **Pokazivačem Steka (SP)** i **Programskim Brojačem (PC)**.

Uopšteno o multithreadingu

Threading Scheduler (Menadžer Niti) je softverski algoritam unutar operativnog sistema (ili RTOS-a) koji pravi raspored iskorišćavanja resursa procesora i memorije. Sam **Multithreading** predstavlja sposobnost sistema da izvršava više ovih niti prividno u isto vreme.



Threading Scheduler (Menadžer Niti)

Gde žive naši podaci? (SFR i organizacija RAM-a)

Unutrašnji RAM kod 8051 ima ukupno 256 bajtova, ali je oštro podeljen na dva dela od po 128 bajtova koji rade na potpuno različite načine:

1. **Donjih 128 bajtova (Adrese 00H do 7FH):** To je prostor namenjen korisniku, radnim registrima i steku. Upravo tu smo smestili stekove naših niti.
2. **Gornjih 128 bajtova (Adrese 80H do FFH):** Tu žive **SFR (Special Function Registers)** registri (kao što su TMOD, TH0, TL0, SP, ACC i PSW). Oni kontrolišu hardver čipa.

Da niti ne bi gazile jedna drugu, napravili smo sledeći bezbedan raspored u donjem RAM-u:

- **Lokacije 30H do 33H:** Koristimo ih kao *Scratchpad* (opšta namena) za čuvanje SP pokazivača i identifikatora aktivne niti.
- **Nit A:** Pokazivač steka (SP) počinje na 35H i raste na gore prema 38H.
- **Nit B:** Pokazivač steka (SP) počinje na 55H i raste na gore prema 58H.
- **Nit C:** Pokazivač steka (SP) počinje na 75H i raste na gore prema 78H.

Trik sa bankama registara i SFR registrom PSW – Context Switch

U klasičnim operativnim sistemima, kada se desi prekid, scheduler mora da uradi gomilu dugotrajnih PUSH instrukcija kako bi sačuvao sve radne registre (R0 do R7) trenutne niti. Na sporom 8-bitnom čipu to bi pojelo previše procesorskog vremena.

Tu na scenu stupa **trik sa bankama registara** preko SFR registra koji se zove PSW **(Program Status Word)**.

Intel 8051 na samom dnu RAM-a ima 4 banke radnih registara. Menjanjem samo dva bita u PSW registru (RS1 i RS0), mi jednim jedinim potezom prebacujemo ceo hardver na drugi set registara:

- MOV PSW, #00000000B → Aktivira **Banku 0** za Nit A (registri R0-R7 su na lokacijama 00H-07H).
- MOV PSW, #00001000B → Aktivira **Banku 1** za Nit B (registri R0-R7 se sele na lokacije 08H-0FH).
- MOV PSW, #00010000B → Aktivira **Banku 2** za Nit C (registri R0-R7 se sele na lokacije 10H-17H).

Zahvaljujući ovom SFR triku, svaka nit ima svojih 8 privatnih radnih registara, a naš scheduler ih menja trenutno, u samo jednom mašinskom ciklusu!

Zlatno pravilo 8051 steka: Kako se adrese zapravo pakuju?

Jedna od najvećih zamki koja buni programere kada ručno pripremaju stek za nove niti jeste redosled bajtova adrese. Programske adrese (PC) su 16-bitne i sastoje se od **HIGH** (višeg) i **LOW** (nižeg) bajta.

Pošto je stek **LIFO** struktura (*Last In, First Out* — poslednji unutra, prvi napolje), pravilo glasi: **ono što želimo da izađe prvo, mora da uđe poslednje!**

Kada se desi pravi hardverski prekid (**IRQ**), mikrokontroler automatski sklanja trenutnu adresu na stek sledećim redosledom:

1. Prvo uvećava SP i upisuje **PC LOW** bajt adrese (na nižu RAM lokaciju).
2. Zatim još jednom uvećava SP (koji postaje vrh steka) i tu upisuje **PC HIGH** bajt adrese (na višu RAM lokaciju).

Kada se prekidna rutina završi i procesor izvrši instrukciju **RETI**, on radi tačno obrnut proces: sa vrha steka prvo skida HIGH bajt i stavlja ga u PC HIGH, a zatim skida LOW bajt i stavlja ga u PC LOW.

Lažiranje steka za Niti B i C

U našoj inicijalizaciji (MAIN), Nit A pokrećemo direktno. Njen prvi ulazak u prekid će biti prirodan – hardver će sam odraditi PUSH PC LOW na 36H i PUSH PC HIGH na 37H.

Međutim, za Niti B i C moramo softverski da **„lažiramo" izgled steka** pre starta sistema. Da bismo dobili identičan raspored u RAM-u kakav hardver prirodno pravi, u kodu pišemo tačan redosled:

```
assembly

MOV A, #LOW(THREAD_B)

PUSH ACC                ; 1. LOW bajt ide prvi na stek (niža adresa 56H)

MOV A, #HIGH(THREAD_B)

PUSH ACC                ; 2. HIGH bajt ide na vrh steka (viša adresa 57H)
```

Zahvaljujući ovome, kada scheduler postavi SP na Nit B i izvrši RETI, hardver vidi savršeno mapiranu memoriju (HIGH na vrhu, LOW ispod) i bez greške skače na početak nove niti!

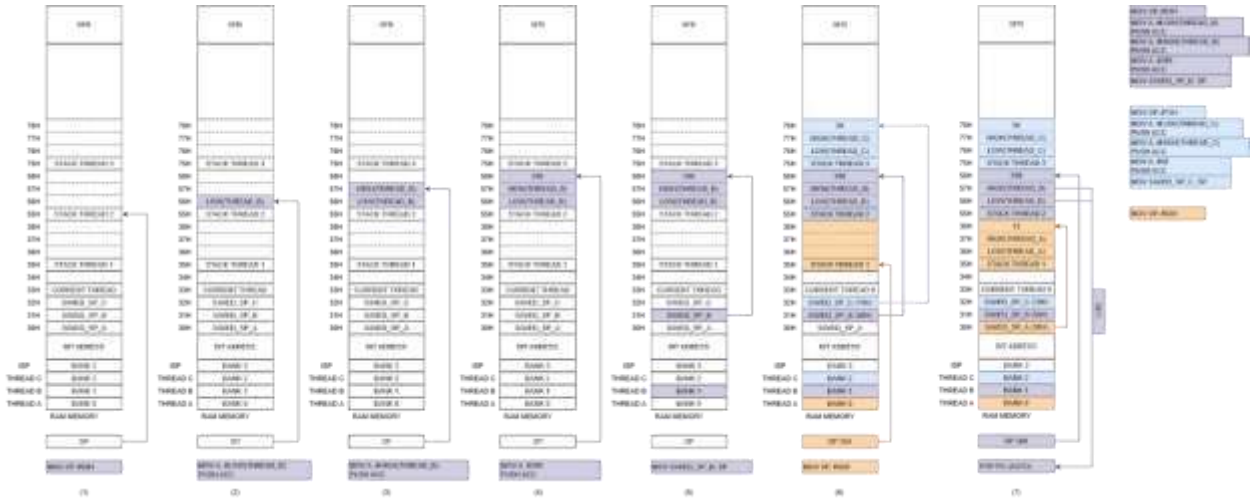
Praktični deo - Kompletan ASM Kod za direktno preuzimanje

```
assembly
; ========================================================================
; ://zilsel-invent.com - Multitasking sa 3 niti na Intel 8051
; ========================================================================
ORG 0000H
LJMP MAIN           ; Skok na inicijalizaciju

ORG 000BH           ; Prekid za Tajmer 0 (Interrupt Vector)
LJMP TIMER0_ISR

; ------------------------------------------------------------------------
; SIGURNE RAM LOKACIJE ZA ČUVANJE POKAZIVAČA STEKA
; ------------------------------------------------------------------------
SAVED_SP_A    EQU
30H   ; Čuva SP za Nit A
SAVED_SP_B    EQU 31H   ; Čuva SP za Nit B
SAVED_SP_C    EQU 32H   ; Čuva SP za Nit C
CURRENT_THREAD EQU 33H  ; Identifikator aktivne niti (0=A, 1=B, 2=C)

; ========================================================================
; INICIJALIZACIJA SISTEMA
; ========================================================================
ORG 0040H
MAIN:
    CLR EA          ; Zabrani prekide dok konfigurišemo niti

    ; --- Priprema steka za Nit B (Banka 1) ---
    MOV SP, #55H
    MOV A, #LOW(THREAD_B)  ; 1. LOW bajt ide prvi (na lokaciju 56H)
    PUSH ACC
    MOV A, #HIGH(THREAD_B) ; 2. HIGH bajt ide na vrh steka (na lokaciju 57H)
    PUSH ACC
    MOV A, #200            ; Početna vrednost ACC za Nit B
    PUSH ACC               ; SP = 58H (Krajnji vrh steka Niti B)
    MOV SAVED_SP_B, SP     ; Čuvamo finalni SP za Nit B u lokaciju 31H

    ; --- Priprema steka za Nit C (Banka 2) ---
```

```
    MOV SP, #75H
    MOV A, #LOW(THREAD_C)  ; 1. LOW bajt ide prvi (na lokaciju 76H)
    PUSH ACC
    MOV A, #HIGH(THREAD_C) ; 2. HIGH bajt ide na vrh steka (na lokaciju 77H)
    PUSH ACC
    MOV A, #50             ; Početni ACC za Nit C
    PUSH ACC               ; SP = 78H (Krajnji vrh steka Niti C)
    MOV SAVED_SP_C, SP     ; Čuvamo finalni SP za Nit C u lokaciju 32H

    ; --- Priprema steka za Nit A (Banka 0) ---
    MOV SP, #35H
    MOV CURRENT_THREAD, #0 ; Označavamo da Nit A kreće prva

    ; --- Podešavanje Tajmera 0 (SFR Registri) ---
    MOV TMOD, #02H  ; Mod 2, 8-bitni auto-reload tajmer
    MOV TH0, #00H   ; Kvant vremena: maksimalan period između prekida
    MOV TL0, #00H
    SETB TR0        ; Pokreni Tajmer 0
    SETB ET0        ; Dozvoli prekid za Tajmer 0
    SETB EA         ; Pusti globalne prekide u rad!

    LJMP THREAD_A   ; Pokrećemo sistem direktnim skokom u Nit A

; ========================================================================
; DEFINICIJE TRI MIKRO-NITI
; ========================================================================
THREAD_A:
    MOV A, #10      ; Početna vrednost akumulatora za Nit A
LOOP_A:
    INC A
    SJMP LOOP_A

THREAD_B:
    ; ACC je za ovu nit već inicijalizovan preko steka na vrednost 200
LOOP_B:
    INC A
    SJMP LOOP_B

THREAD_C:
    ; ACC je za ovu nit već inicijalizovan preko steka na vrednost 50
LOOP_C:
    INC A
    SJMP LOOP_C

; ========================================================================
; PREKIDNA RUTINA (ISR) - SCHEDULER (A -> B -> C -> A)
; ========================================================================
TIMER0_ISR:
    PUSH ACC        ; Sačuvaj trenutni ACC aktivne niti na njen stek

    MOV A, CURRENT_THREAD
    CJNE A, #0, CHECK_B

    ; --- KORAK 1: Ako je radila Nit A, prebaci na Nit B ---
    MOV SAVED_SP_A, SP  ; Zapamti vrh steka Niti A (sada je na 38H)
    MOV SP, SAVED_SP_B  ; Učitaj SP Niti B (prebacuje na lokaciju 58H)
    MOV CURRENT_THREAD, #1
```

```
    MOV PSW, #00001000B ; SFR TRIK: Prebaci hardver na Banku 1 (Nit B)
    SJMP RESTORE

CHECK_B:
    CJNE A, #1, CASE_C

    ; --- KORAK 2: Ako je radila Nit B, prebaci na Nit C ---
    MOV SAVED_SP_B, SP  ; Zapamti vrh steka Niti B
    MOV SP, SAVED_SP_C  ; Učitaj SP Niti C (prebacuje na lokaciju 78H)
    MOV CURRENT_THREAD, #2
    MOV PSW, #00010000B ; SFR TRIK: Prebaci hardver na Banku 2 (Nit C)
    SJMP RESTORE

CASE_C:
    ; --- KORAK 3: Ako je radila Nit C, vrati na Nit A ---
    MOV SAVED_SP_C, SP  ; Zapamti vrh steka Niti C
    MOV SP, SAVED_SP_A  ; Učitaj SP Niti A (vraća na lokaciju 38H)
    MOV CURRENT_THREAD, #0
    MOV PSW, #00000000B ; SFR TRIK: Vrati hardver na Banku 0 (Nit A)

; --- KORAK 4: Vrati kontekst nove niti i nastavi rad ---
RESTORE:
    POP ACC          ; Sa vrha steka nove niti skini njen sačuvani ACC
    RETI             ; Hardver skida HIGH pa LOW sa steka i nastavlja rad
niti!
```

Vodič kroz Blok Dijagram



micro-threading stek inicijalizacija

- **Kolone 1 do 5 (Ljubičasta i plava zona):** Prikazuju softversko punjenje steka u MAIN programu za Nit B i C. LOW bajt ide na pozicije 56H/76H, a HIGH bajt tik iznad njih na 57H/77H. Vrh steka pre pokretanja sistema je na akumulatoru (58H/78H).
- **Kolona 6 (Narandžasta zona):** Prikazuje start Niti A sa čistim stekom postavljenim na 35H.

- **Kolona 7 (ISR faza):** Prikazuje trenutak kada tajmer prekine rad Niti A. Hardver samostalno radi PUSH PC LOW na lokaciju 36H, zatim PUSH PC HIGH na 37H, a prva linija koda u ISR-u dodaje PUSH ACC na lokaciju 38H. Scheduler tada uzima tu krajnju vrednost 38H i bezbedno je zaključava u lokaciju 30H (SAVED_SP_A).

Matematika merenja performansi — Od 8051 do SSE ekstenzija i MSR registra

U prethodnom tekstu smo videli kako se na nivou čistog silicijuma bije bitka za svaki mašinski ciklus i kako tajmer i banke registara omogućavaju paralelni rad niti. Kada ovladate mikrokontrolerom kao što je Intel 8051, prirodno je da dobijete inženjersku intuiciju da performanse softvera posmatrate kroz **taktove procesora i oscilacije kvarcnog oscilatora**.

Međutim, kada pokušate da taj isti šablon razmišljanja primenite na modernom PC-ju – na primer, kada napišete C++ kod ubrzan pomoću **SSE (Single Instruction Multiple Data)** ekstenzija – udarićete u nevidljivi hardverski zid. Odjednom, stara matematika prebrojavanja instrukcija "peške" prestaje da važi, a merenje performansi softvera od čiste matematike postaje napredna fizika i statistika.

Svet apsolutnog determinizma: Kako smo merili vreme na 8051

Na bazičnom 8-bitnom mikrokontroleru 8051 matematika performansi je savršeno egzaktna i linearna. Ako vaš program ne radi dovoljno brzo, matematika je surova i jasna:

Broj instrukcija×Broj mašinskih ciklusa po instrukciji=Tačan broj taktova

Znamo da instrukcija `INC A` troši tačno 1 mašinski ciklus (odnosno 12 oscilacija eksternog kvarca). Nema faktora iznenađenja. Nema operativnog sistema koji će usred petlje oduzeti procesorsko vreme, nema keš memorije koja može da omane i nema termalne kontrole. Sve je **100% predvidivo (deterministički)**. Broj taktova sa fiksnim periodom oscilacije daje vam tačno vreme u nanosekundu.

Veliki krah determinizma: Gde nestaje matematika na modernom PC-ju?

Kada pređemo na modernu x86/x64 arhitekturu (Intel/AMD) i pokušamo da uporedimo standardni C++ kod sa ručno optimizovanim SSE/AVX instrukcijama, matematika peške umire. Čak i ako napišete najjednostavniju petlju za sabiranje nizova, rezultati merenja standardnom "štopericom" (`std::chrono`) variraju iz izvršavanja u izvršavanje.

Zašto je to tako? Tri hardverska krivca unutar modernog silicijuma potpuno ruše predvidivost instrukcija:

- **Out-of-Order Execution (Izvršavanje preko reda):** Moderni procesor nije slepi izvršilac asemblerskih linija. Hardverski dekoder razbija vaš C++ i SSE kod na mikrooperacije (μops) koje izvršne jedinice gutaju preko reda, kad god je neki resurs slobodan.
- **Termodinamička ezoterija (Dynamic Scaling):** Za razliku od stabilnog kvarca na 8051, moderni CPU konstantno menja svoju frekvenciju (Turbo Boost, termalni throttling) u zavisnosti od temperature čipa i opterećenja susednih jezgara. Takt iz sekunde u sekundu fizički nije isti.
- **Paradoks Latencije i Propusnog opsega (Throughput):** Na PC-ju, složena SSE instrukcija može imati latenciju od 3 takta, ali propusni opseg od 0.5 taktova. To znači da procesor može da započne dve masivne SSE operacije u jednom jedinom taktu! Kako je to matematički moguće?

Hardverski trik: Kako procesor vara vreme?

Da bismo razumeli kako instrukcija koja traje 3 takta može imati propusni opseg od 0.5 taktova, moramo napustiti linearni model razmišljanja i zamisliti **fabričku traku (Pajplajn)** i **superskalarnu arhitekturu** modernih procesora.

1. **Analogija sa fabričkom trakom (Pajplajn):** Zamislite fabriku automobila u kojoj sklapanje jednog auta traje tačno 3 sata (to je *Latencija*). Ako fabrika ima pokretnu traku i radnici na svakih pola sata (0.5 sati) ubace novu školjku na početak trake, šta se dešava na kraju? Kada se traka jednom napuni, fabrika će izbacivati po jedan gotov automobil na svakih pola sata! To je *Propusni opseg (Throughput)*. Dok prva SSE instrukcija u procesoru završava svoju treću fazu (npr. zaokruživanje rezultata), druga je na drugoj fazi, a treća tek ulazi u prvu fazu izvršenja.
2. **Duplirane mašine unutar jezgra (Superskalarni CPU):** Moderno procesorsko jezgro ima više paralelnih izvršnih jedinica (tzv. *Execution Ports*). Ako u vašem SSE kodu dve instrukcije ne zavise jedna od druge (npr. sabirate potpuno različite registre), hardverski dekoder će u istom taktu poslati prvu instrukciju na Port 0, a drugu na Port 1. Obe instrukcije kreću istovremeno.
   Međutim, ako vaša sledeća C++ linija koda zavisi od rezultata prethodne (npr. `y = x + c` gde se čeka da se izračuna `x`), procesor mora da zakoči. Fabrička traka se zaustavlja i vi plaćate punu cenu latencije od 3 takta. Zato štoperica daje različite rezultate za naizgled isti broj instrukcija!

Most ka enterprise sistemima (Primer: SQL Server)

Razmišljajući o tome kako su optimizovani džinovski softverski sistemi poput **Microsoft® SQL Servera®**, dolazimo do iste spoznaje: oni su pisani tako da maksimalno eksploatišu ove hardverske ekstenzije. Kada SQL Server u memoriji skenira milijarde redova, on u pozadini ne koristi "kašiku" (standardne skalarne instrukcije), već "lopatu za sneg" – **SSE i AVX vektorske instrukcije** koje jednim udarcem procesiraju više podataka odjednom (SIMD).

Ali, da bi softverski inženjeri uopšte znali da li mašinerija radi efikasno, klasična štoperica im ne pomaže. Ako procesor stoji (Stall) jer podatak nije stigao iz RAM-a u keš liniju (Cache Miss), procesor troši prazne taktove bez obzira na moćni SSE kod.

Rešenje zagonetke: Kako zap vaporno merimo softver na PC-ju?

Pošto peške ne možemo računati, moderni inženjeri koriste isti pristup koji se koristi u deep-level profilisanju SQL Servera: **MSR registre (Model-Specific Registers)**.

Unutar samog silicijuma Intel i AMD procesora ugrađeni su hardverski brojači performansi (**Performance Monitoring Counters - PMC**). Alati poput **Intel® Performance Counter Monitor (PCM)** direktno čitaju ove registre u realnom vremenu.

Umesto pukog merenja sekundi, moderni rendgenski snimak koda zahteva da pratimo:

1. **IPC (Instructions Per Cycle):** Koliko je instrukcija procesor zaista uspeo da završi po jednom taktu (teoretski maksimum je preko 4-5, ali ako je kod loš usled zavisnosti podataka, pada ispod 0.5).
2. **L3 Cache Misses:** Koliko puta je tvoj SSE kod morao da čeka spori RAM jer podaci nisu bili poravnati i spremni u kešu procesora.

Direktno poređenje: Dva sveta arhitekture

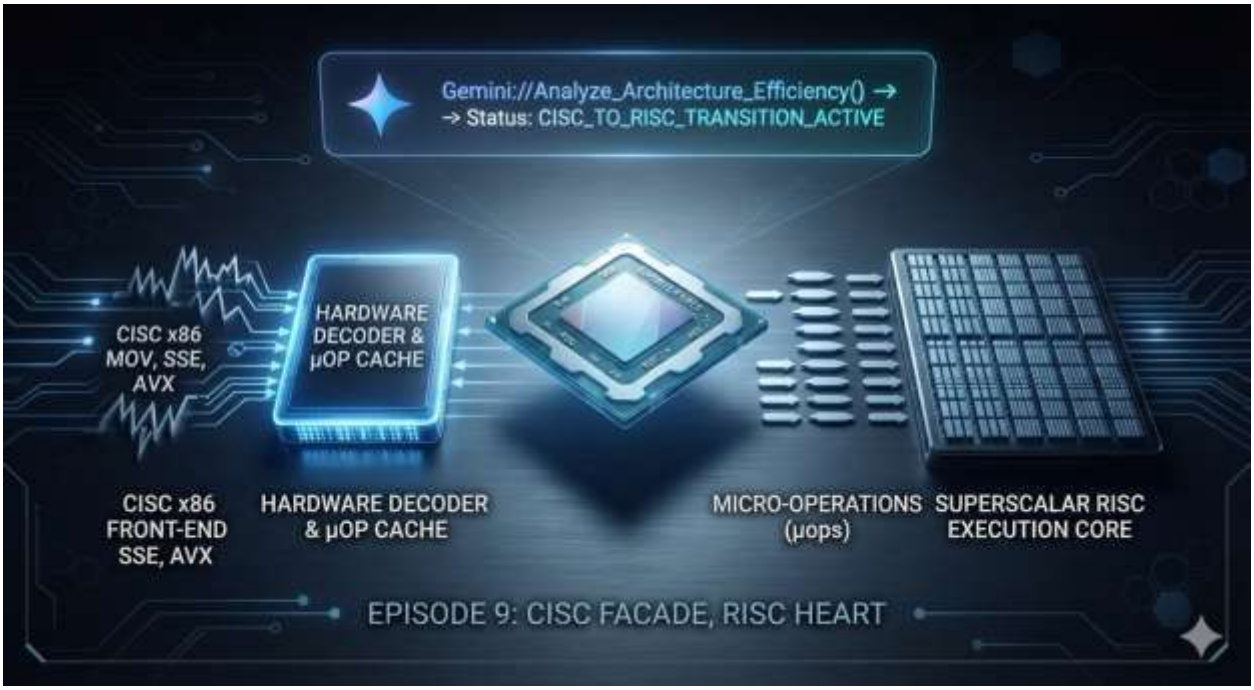
Metrika merenja	Intel 8051 Arhitektura	Moderni PC (x86/x64 + SSE)
Izvor takta	Fiksni eksterni kvarcni oscilator	Dinamička frekvencija (zavisna od termodinamike)
Proračun koda	Egzaktna matematika (Brojanje instrukcija peške)	Nemoguće peške (Out-of-Order i Pipelining haos)
Alat za merenje	Logički analizator / Papir i olovka	Hardverski MSR registri (Intel PCM alat)
Glavni problem	Ograničen broj registara i RAM prostora	Promašaj keš memorije (Cache Miss) i čekanje na RAM

Inženjerski zaključak: Znanje sa 8051 nam daje savršenu mentalnu mapu da razumemo šta su to registri i zašto su oni najbrža zona na čipu. Ali na modernom PC-ju, optimizacija softvera nije samo borba sa asemblerskom sintaksom, već borba za to da procesor nikada ne ostane gladan podataka i da hardverski brojači (MSR) pokažu maksimalan IPC!

Rat standarda i velika obmana arhitekture — CISC fasada i RISC srce

Kroz ovaj serijal prošli smo put od stambene krize unutar 128 bajtova RAM-a, preko prekidnih rutina i lažiranja steka, pa sve do termodinamičkog haosa i MSR brojača na modernom PC-ju. U ovom završnom tekstu skidamo poslednji veo apstrakcije sa modernih procesora (Intel/AMD) i otkrivamo veliku tajnu: **rat između CISC i RISC arhitektura zapravo je završen primirjem — RISC je pobedio unutar čipa, a CISC je pobedio na papiru.**

Kada kompajlirate svoj C++ ili SSE kod na PC-ju, vi procesoru isporučujete kompleksan skup instrukcija poznat kao x86 standard (CISC). Međutim, moderni procesorski silicijum uopšte ne zna kako da direktno izvrši te instrukcije.



Generisao AI Gemini

CPU Iluzija: Hardverski dekoder kao prevodilac

Da bi zadržali kompatibilnost sa starim softverom pisanim decenijama unazad, proizvođači procesora su morali da zadrže CISC fasadu. CISC (*Complex Instruction Set Computer*) omogućava kompleksne naredbe — na primer, da jednom instrukcijom povučete podatak iz RAM-a, saberete ga sa registrom i rezultat vratite u RAM.

U čistom RISC (*Reduced Instruction Set Computer*) svetu to je strogo zabranjeno. Podaci se moraju eksplicitno učitati u registre (*Load*), obraditi (*Execute*) i vratiti u memoriju (*Store*).

Kako onda x86 procesor izvršava taj teški CISC softver brzinom munje?

1. **Hardware Decoder (Prevodilac na ulazu):** Na samom ulazu u procesorsko jezgro nalazi se ultra-brzi hardverski dekoder. Njegov jedini posao je da presretne kompleksnu CISC instrukciju i razbije je (prevede) na nekoliko malih, prostih RISC instrukcija. Intel ove interne RISC instrukcije naziva **Mikrooperacije (µops ili uops)**.
2. **Interno RISC jezgro:** Kada se fasada sruši i instrukcija razbije na mikrooperacije, one ulaze u izvršne jedinice procesora koje su strukturalno čista, agresivna RISC mašina sa stotinama internih registara za koje vaš operativni sistem uopšte i ne zna da postoje.

Konceptualni most: Mikrosekvener kao praotac RISC-a

Ova revolucija je počela još sredinom 1990-ih godina (sa čuvenom arhitekturom Intel Pentium Pro iz 1995. godine) i traje i danas. Ipak, ako zagrebemo još dublje, shvatićemo da moderni procesori u svom srcu rade isto ono što smo radili na Intel 8051 — upravljaju stanjima.

Kada ste kroz hardverski tajmer na 8051 ručno manipulisali stanjima, bankama i registrima, vi ste radili posao **mikrosekvenera**. Kod modernih procesora, taj posao je automatizovan: hardverski dekoder prevodi spoljni CISC standard u interni "mikrokod" (niz mikrooperacija) koji pokreće unutrašnju state mašinu procesora u ritmu gigahercnog takta.

Silicijumski flaster: Krpljenje procesora kroz mikrokod

Ovaj hibridni dizajn (CISC na ulazu, RISC u srcu) donosi još jednu neverovatnu inženjersku prednost. Pošto se kompleksne instrukcije u pozadini izvršavaju preko mikrokoda, šta se dešava ako inženjeri u Intelu ili AMD-u naprave kritičnu grešku u samom silicijumu (hardverski bug)?

Umesto povlačenja miliona procesora sa tržišta, **greška se krpi softverski**! Prilikom podizanja sistema, kroz BIOS/UEFI u procesor se "ubrizgava" **Microcode Update** — softverski flaster koji menja mapu unutar ulaznog dekodera. Dekoder nakon toga problematičnu CISC instrukciju bezbedno razbija na drugačiji niz stabilnih mikrooperacija, zaobilazeći hardverski kvar na čipu.

Finalni uporedni pregled arhitektura

Karakteristika	Bazični Intel 8051	Moderni x86/x64 Procesor
Spoljašnja Arhitektura	Čist 8-bitni CISC	Kompleksni 64-bitni CISC standard
Unutrašnje Izvršenje	Direktno (Instrukcija pokreće silicijum)	Hibridno (Razbijanje na RISC μops)
Fleksibilnost hardvera	Fiksna (Logika urezana u tranzistore)	Dinamična (Mogućnost izmene mikrokoda)
Kompatibilnost koda	Vezana za specifičnu 8051 familiju	Apsolutni standard (Izvršava kod iz 1980-ih)

Granica između hardvera i softvera na samom dnu ne postoji — sve je samo matrica stanja. Bez obzira na to da li optimizujete asemblerski kod za mali 8-bitni čip pomeranjem bitova u PSW registru, ili kompajlirate masivne C++ SSE aplikacije na PC-ju, principi su ostali identični. Razumevanje bazičnog silicijuma oslobađa vas iluzije automatizma i od programera koji samo kuca sintaksu pravi **arhitektu softverskih sistema**.