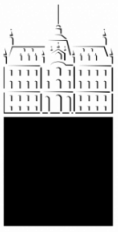


Univerza *v Ljubljani*
Fakulteta *za računalništvo*
in informatiko

Vanja Stojanović

Zbirni jezik na podlagi 8-bitne arhitekture Von Neumannovega modela

Seminarska naloga

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentorja: viš. pred. dr. Robert Rozman, asist. Žiga Pušnik

Ljubljana, 2021

Izjavljam, da je seminarska naloga v celoti moje delo in si do njega pridržujem vse avtorske pravice. Vse uporabljene literature so korektno navedene pod viri.

Vanja Stojanović

Kazalo

Povzetek

S raznimi teoretičnimi podlagami, je mogoče sestaviti neko računalniško arhitekturo, specifično v tem delu, arhitekturo danes pogosto uporabljenega Von Neumannovega modela. Računalnikovo delovanje je možno zmeraj simulirati in tudi izvajati tako rečeno ročno, vendar je znanost v tem kako tak proces avtomatizirati s pomočjo nekega jezika iz ukazov. Računalniki na tem nivoju uporabljajo logične napetostne nivoje, ki predstavljajo digitalne informacije v binarnem številskem sestavu. Ta se uporablja, ker je zanj najlažje definirati dva stabilna stanja, s katerimi lahko operiramo, te dva označujemo z 0 in 1. Računalnik s ničlami in enicami znajo "v naravi" le seštevanje, vse ostale matematične operacije so izpeljane iz te, tudi bolj kompleksne operacije. Poleg pomenske zmogljivosti, arhitektura potrebuje tudi neko logično podlago, ki definira kdaj se katera operacija izvede, to imenujemo kontrolna logika. Kontrolna logika računalnik dobesedno kontrolira s raznimi kontrolnimi signali, ki določajo točno to, zaporedje izvedb operacij. Zaporedje je pa časovno odvisno, kar pomeni da moramo mi, kot programerji računalniku povedati, kdaj izvesti katere operacije, to naredimo v obliki programa, ki je sestavljen iz zbirnega jezika. Po nivoju jezikov v računalniški znanosti, je ta zelo blizu strojne kode oz. naravnega jezika arhitekture. Zbirni jezik sestavljajo ukazi, ki opravljajo razne semantične operacije v *t. i.* paketih strojne kode. Kakšni so te ukazi in kaj počno je popolnoma odvisno od arhitekture in njenega ustanovitelja.

Ključne besede: 8-bitna računalniška arhitektura, zbirni jezik, digitalna tehnika, CPE-ja

Abstract

With different theoretical fundamentals, it is completely possible to build some sort of computer architecture, specifically in this case, an architecture most commonly used today, Von Neumann's computer model. It is always possible to simulate how the computer is working and executing something manually, but the science is in automating this process with the help of some language made out of commands. On this level the computers use logical voltage levels, to represent digital information in the binary system. The binary system is used for its simplicity to define two stable states with which we can operate, we denote these states with 0s and 1s. The computer in its nature only knows how to add with the two states, every other mathematical operation is derived from that, so are more complex operations. Other than some meaningful capabilities, the architecture needs some logical base, which defines when an operation should execute, that base is named the control unit. It, in the literary meaning, controls the computer via many control signals, which define just that, the sequence of operation executions. This sequence is, however, time-dependent, which means that we, as programmers have to instruct the computer when and which operations to execute. We do that in the form of a program, which is composed out of the assembly language. According to the language hierarchy in computer science, it is very close to machine code *i. e.* the natural language of the architecture. Assembly is made up of different commands, that perform various semantical operations in so-called batches of machine code. The format and definitions of these commands are limited only by the architecture itself and its creator.

Keywords: 8-bit computer architecture, assembly, digital technology, CPU

Uvod

Svet računalništva se skozi čas vse hitreje razvija, tudi v sedanjosti ni opažanj, da bi se le ta ustavil ali upočasnil. Vseskozi z razvojem aplikacij in strojnih oz. računalniških sistemov, strmimo k visokem nivoju abstrakcije, da program za nas naredi čim več sam od sebe, medtem se lahko mi, kot računalničarji osredotočimo na pomembnejše dele, kot so tok programi ali pa algoritmi ali pa predstavitev podatkov ipd. Redko kdaj se obrnemo nazaj in se vprašamo, ali program oz. stroj za nas vse te procese opravlja res tako efektivno, kot bi jih lahko, ali jih zgolj in samo opravlja, ne glede na to kako potratni so te. Take procese najlažje optimiziramo tako, da jih računalniku definiramo v njemu najbolj prijaznem jeziku, v katerem je potrebno specificirati vsak sleherni korak postopka. Ta jezik se imenuje zbirni jezik, to je tehnologija, ki ohranja velik kontakt s strojno kodo a hkrati je človeško berljiva do takega nivoja, da se jo lahko sistematično učimo.

Zaradi tega, ker je zbirni jezik tako blizu s dano strojno kodo sistema, je ta velikokrat specifičen za arhitekturo in realizacijo stojne opreme, na kateri program pišemo. Tako efektivno dobimo orodje za optimizacijo, a zgubimo generalizacijo takega jezika na več različnih sistemih, saj ima vsak sistem, ki si je po zgradbi drugačen, tudi drugačno strojno kodo in posledično drugačen zbirni jezik.

To seminarsko delo je namenjeno teoretičnemu dokazovanju tega, da je zbirni jezik mogoče ustvariti tudi za svojo arhitekturo in, da je njegovo razumevanje tesno povezano s poznavanjem strojne opreme in arhitekture, na kateri delujemo.

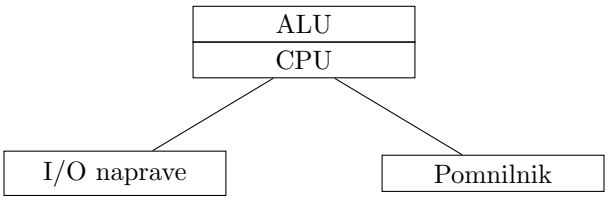
1 Koncept dane arhitekturne podlage

Računalnike lahko gradimo na mnogo načinov, od raznih konceptualnih shem, do izvedb z razno tehnologijo. Računalnik v sledeči teoriji temelji na teoretičnem konceptu iz navedene literature *8-bitna računalniška enota*.

1.1 Arhitektura

1.1.1 Konceptualni in logični nivo

Računalnik, na podlagi katerega, bomo napisali zbirni jezik, je bil modeliran po **Von Neumannu**. To je znan računalniški model, ki predvideva 4 funkcionalne sestavne enote, ter njihove naloge. Te so sledeče;

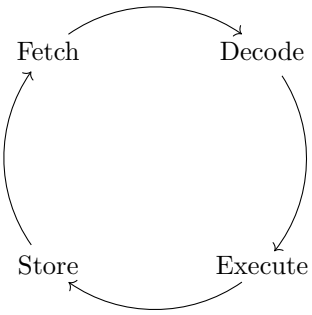


Slika 1.1: Von Neumannov model

Kot prikazuje Slika 1.1, so naše enote ALU, CPU, I/O naprave in pomnilnik. I/O naprave, so Vhodno/Izhodne naprave (*angl.* Input/Output devices) in te služijo nabiranju podatkov od uporabnika oz. iz okolja samega stroja, ter prikazovanju teh podatkov v okolju razumljivem formatu. ALU in CPU so možgani našega stroja, Aritmetično logična enota (*angl.* Arithmetic Logical Unit) zna izvajati samo aritmetične operacije, na katerih so zgrajene bolj kompleksne operacije, medtem ko pa Centralna Procesna enota (*angl.* Central Processing Unit) zgolj procesira podatke iz I/O naprav, jih pravilno usmerja naprej in poskrbi za harmonizacijo vseh enot. Zadnja enota je pomnilnik, v njem so shranjeni programi, ki so le skupki ukazov in njihovih pripadajočih argumentov.

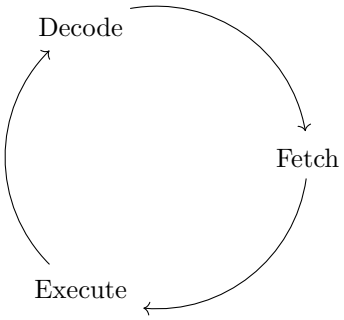
1.1.1.1 Osvnovna izvedba programa

Program, ki je shranjen v pomnilniku Von Neumannovega modela, bi se izvajal v ciklu, ki ga imenujemo ukazni cikel (*angl.* Instruction cycle) ali tudi Fetch-Decode-Execute-Store cikel. CPU po tem ciklu, prvo pridobi podatke oz. ukaz iz glavnega pomnilnika, ta ukaz s pomočjo logike dekodira, to pomeni, da ga prevede v sebi razumljivo strojno kodo (t. j. binarna koda 1 in 0). Nato ukaz s pripadajočimi argumenti izvede in na koncu rezultat tudi shrani nazaj v pomnilnik (Zadnji korak je v nekaterih arhitekturah opcijski).



Slika 1.2: Ukazni cikel

Ta cikel, se konstantno ponavlja in efektivno dobimo delujoč računalnik Von Neumannovega modela. Bolj poenostavljen ukazni cikel, bi se glasil zgolj Fetch-Decode-Execute;



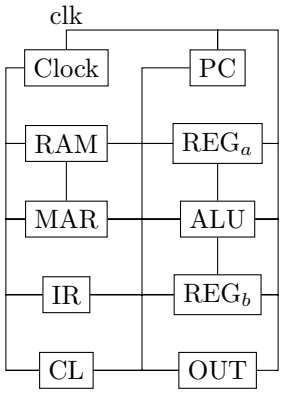
Slika 1.3: Poenostavljen ukazni cikel

1.1.2 Sestava in izvedba

Arhitektura, katera bo podlaga za zbirni jezik, je 8-bit Von Neumannov model računalnika, to v praksi pomeni, da ima vse registre, tako generalne, kot specializirane 8-bitne, prav tako je glavno vodilo 8-bitno.

Računalnik predvideva 10 sestavnih *modulov* oz. komponent, ki skupaj v harmoniji opravljajo operacije računalnika. Te so sledeči;

- Urin modul
- Programski števec (*angl.* Program Counter (PC))
- Glavni pomnilnik (*angl.* Random Access Memory (RAM))
- Pomnilniško-naslovni register (*angl.* Memory Address Register (MAR))
- Ukazni register (*angl.* Instructions Register (IR))
- Generalna registra A in B (*angl.* Registers A and B (REG_a/REG_b))
- Aritmetično-Logična enota (*angl.* Arithmetic Logic Unit (ALU))
- Izhodna enota (*angl.* Output unit (OUT))
- Kontrolna logika (*angl.* Control Logic (CL))



Slika 1.4: Konceptualna shema računalnika

Slika 1.4 nam nakaže konceptualno shemo računalnika oz. njegovo logično izvedbo na abstraktnem nivoju sestavnih modulov, ter njihovo medsebojno povezanost.

1.1.2.1 Povezava sestavnih modulov in Von Neumannovega modela

Vse module računalnika lahko uvrstimo med sestavne enote Von Neumannovega modela, kot je prikazano na Sliki 1.1. Clock zagotavlja urin pulz, ki sinhronizira vse module in s tem celoten računalnik. Programski števec sledi pomnilniškemu prostoru, in nam pove kje v programu se nahajamo. Ukazni register hrani trenutni ukaz, ki se izvaja v programu, tega s pomočjo kontrolne logike dekodiramo, hkrati pa kontrolna logika skrbi za pravilno zaporedje in operiranje samega računalnika. Do sedaj našteti elementi oz. moduli spadajo pod enoto CPU-ja, ALU je enota zase in pod njo bi lahko vzeli oba splošna registra A in B, ter sam ALU modul. Vhodno/Izhodne sisteme pa predstavljamo mi, ki moramo računalniku ročno vnašati podatke in pa izhodni modul, k prikazuje rezultate operacij v desetiškem sestavu.

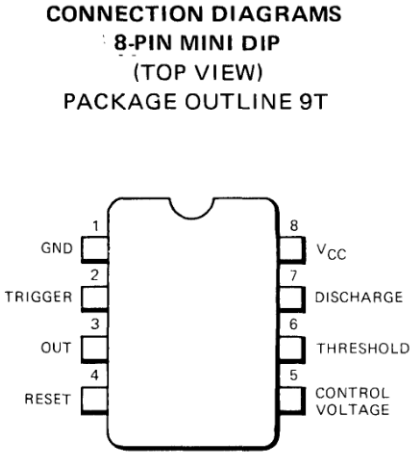
1.1.3 Moduli računalnika

Moduli so sestavljeni iz 74xx serije integriranih vezij, tehnološke družine HC in LS, ki vsebuje CMOS oziroma TTL izvedbo čipa. Razlog za mešanje teh tehnologij, sledi zgolj iz tega, da je LS zastarel in ponekod potrebni deli niso bili dobavljivi, tu te nadomesti HC. Izvedba je postavljena na prototipnih ploščicah (*angl.* Breadboards), kar je omogočala enostavnost in praktičnost pri sestavljanju modulov, ter njihovo medsebojno povezovanje. Računalnik je napajan s **5V** električne napetosti.

Opomba (Prototipne ploščice). Prototipne ploščice so daleč od optimalne podlage za elektroniko, uporabljene so bile zgolj za dokaz danega koncepta. Če bi se zadeva naredila, tako rečeno bolj profesionalno, bi predlagal standardno tiskano vezje.

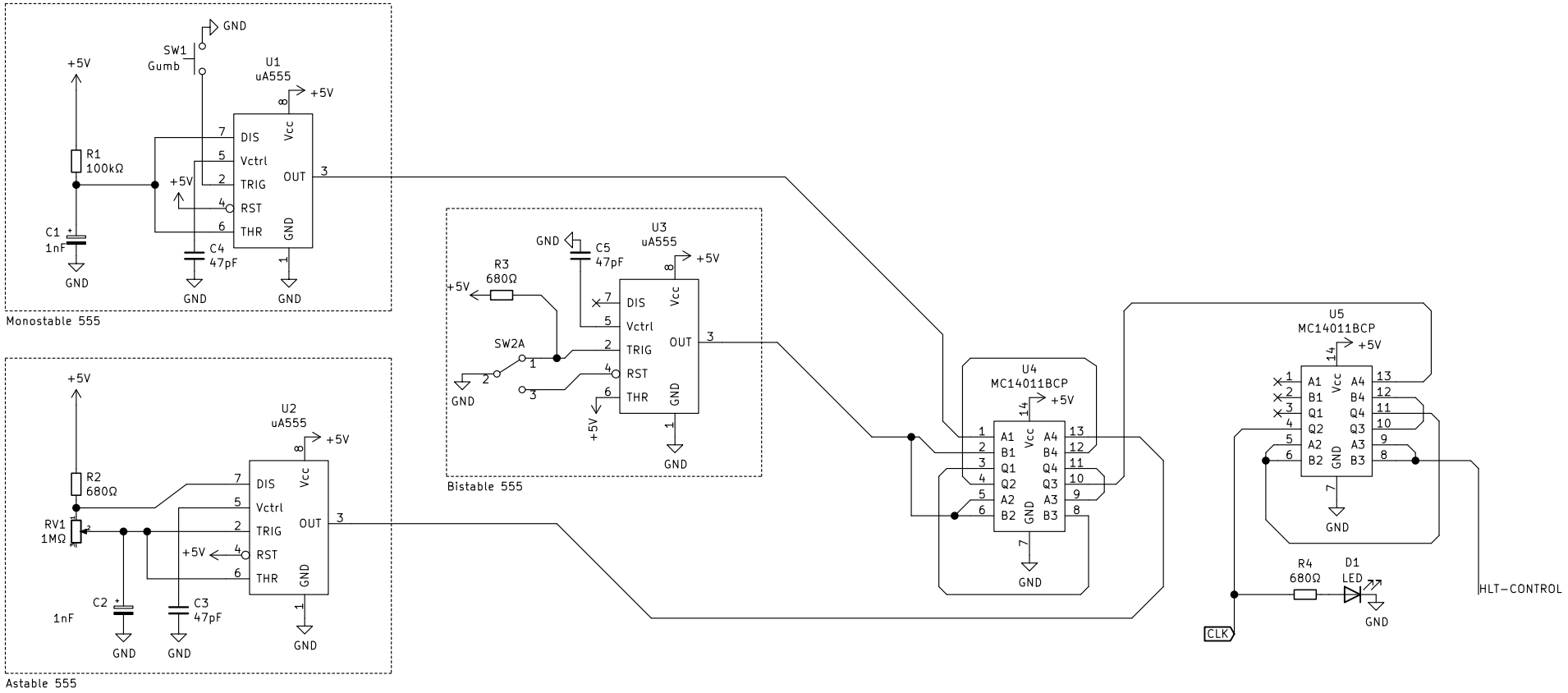
1.1.3.1 Urin modul

Urin modul je sestavljen iz treh vzporedno vezanih 555-timer čipov, ki so čipovja za časovnik. Vsak od njih je vezan v eno izmed najpogostejših osnovnih vezav tega čipa, te so **monostabilna**, **bistabilna** in **astabilna** vezava.



Slika 1.5: Diagram povezav 555-timer čipa

Shema vezja:



Clock je zmožen samodejnega generiranja pulza ali generiranja pulza na pritisk gumba SW1, med načini preklapljamo z SW1A.

Vanja Stojanović

Sheet: /
File: Clock – schematic.sch

Title: CLOCK Pulse Generator

Size: A4	Date: 2020-12-04	**Rev: 1**
KiCad E.D.A. kicad (5.1.8)-1		Id: 1/1

Čip na podlagi napetostnih nivojev na svojih vhodih, generira časovno natančne signale na izhodih, katere lahko s ostalimi pasivnimi členi po lastni potrebi spreminjamo (t. j. *moduliramo*). Izhodni urin pulz je neodvisen od ostalih modulov in operira s svojim taktom, povezan je le s HLT signalom, ki ko je prižgan ustavi delovanje ure in efektivno celotne računalniške enote.

1.1.3.2 Modul splošnega registra A in B

Splošna registra A in B sta sestavljena iz dveh komponent, samega 8-bitnega registra in kontrole pristopa do glavnega vodila.

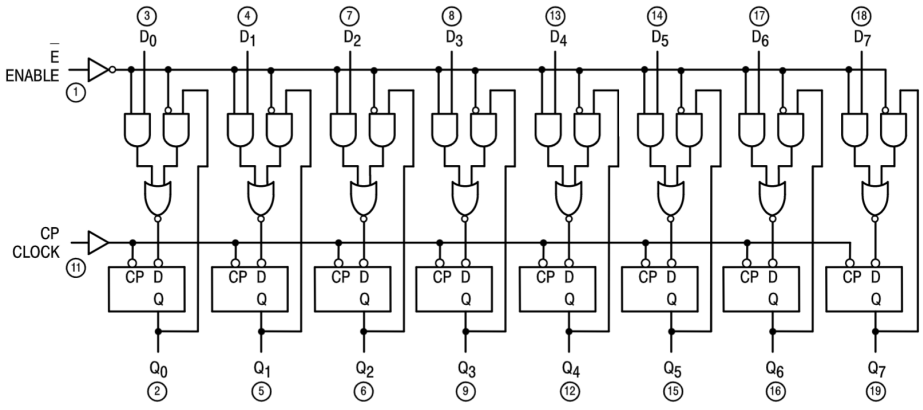
8-bitni register sestavljata čipa 74LS377 in 74LS245. Čipovje 74LS377 je 8-bitni D-tip Flip-Flop register s enable signalom. Deluje na principu D pomnilniške celice, saj je sestavljen iz 8 takih.

FUNCTION TABLE
(EACH FLIP-FLOP)

INPUTS			OUTPUTS	
$\overline{G}$	CLOCK	DATA	Q	$\overline{Q}$
H	X	X	Q_0	$\overline{Q}_0$
L	↑	H	H	L
L	↑	L	L	H
X	L	X	Q_0	$\overline{Q}_0$

Slika 1.6: Resničnostna tabela D-flip flopa 74LS377 čipa

8 takih celic skupaj tvori 8-bitni register, ki je sinhroniziran s skupnim clock pulzom. Čip je omejen tudi s svojim lastnim ENABLE signalom, ki, ko je povezan na logično "1" ne dopušča spreminjanje vsebine registra ("zaklene" vsebino).



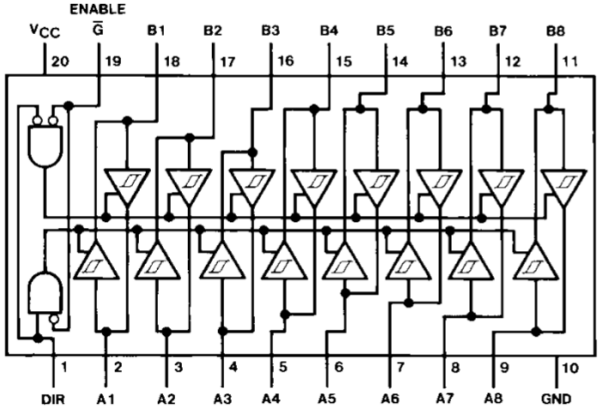
Slika 1.7: Logični diagram 74LS377 čipa

Drugi sestavni del je čip 74LS245, ta je oktalni transiver za dostop do glavnega vodila, s 3-stanjskim izhodom, s možnostjo izmenjave smeri prevajanja. Deluje na podlagi 3-stanjskega buffer sistema, ki je glede na kontrolni signal v prevodnem stanju, ali pa v *visoko impedančnem* (Z) stanju;

A	E	Q_0
0	0	Z
0	1	0
1	0	Z
1	1	1

Slika 1.8: Resničnostna tabela 3-stanjskega bufferja

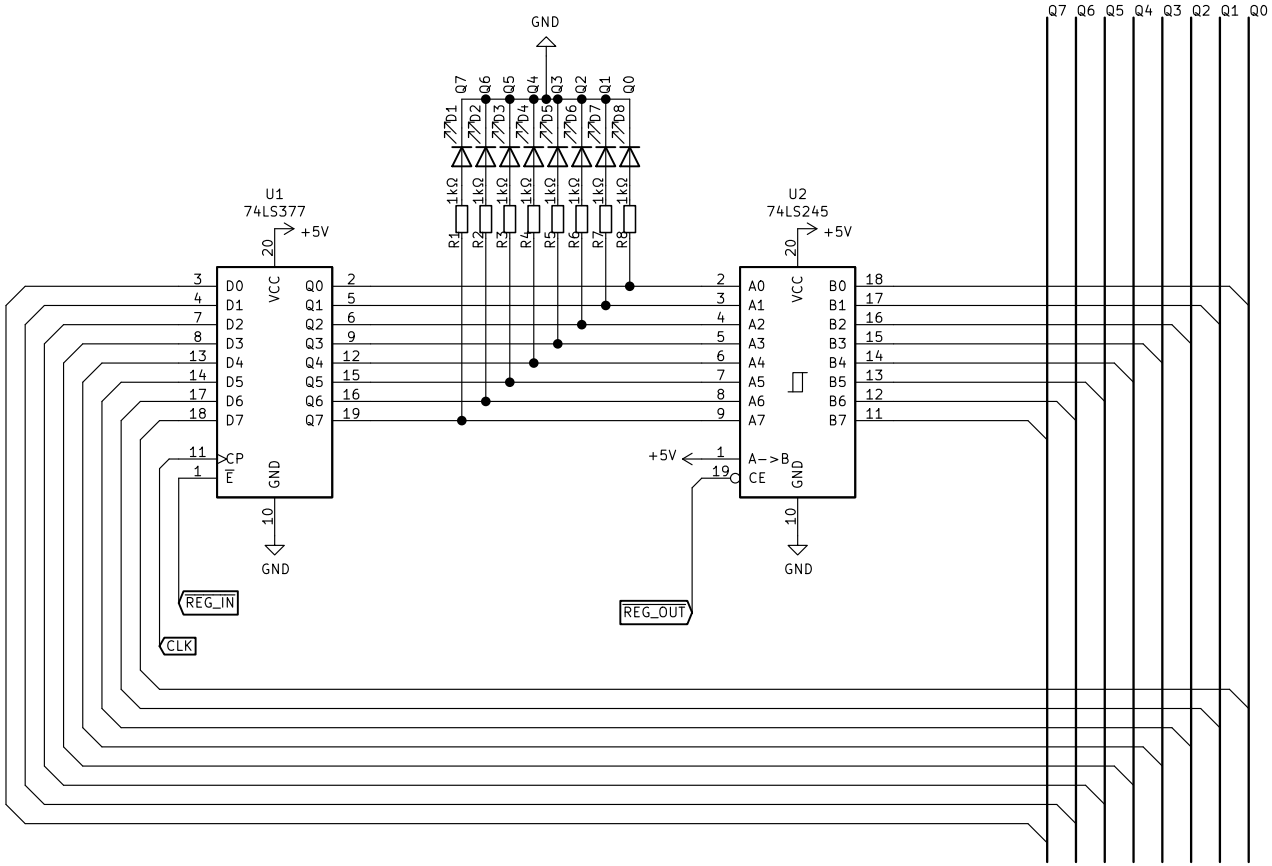
Zgornja tabela (Slika 1.8) prikazuje vhodna in izhodna stanja takega elementa, A je podatkovni vhod, E kontrolni signal in Q_0 izhodni signal.



Slika 1.9: Logični diagram 74LS245 čipa

Pini A_1 do A_8 in B_1 do B_8 so vhodi, ki so izmenljivi, to pomeni, da lahko pretok toka usmerimo iz A $\rightarrow$ B ali obratno B $\rightarrow$ A. Pin $\overline{G}$ je enable pin 3-stanjskih bufferjev v čipu. Tako efektivno dobimo kontrolo dostopa do glavnega vodila s strani registrov, *t. j.* kontroliramo, kateri modul podatke pošilja na vodilo in kateri jih iz njega bere.

Shema vezja:



8-Bit Register z D-flip flop pomnilniškimi celicami, z vhodom iz glavnega vodila in izhodom preko vodilnega transiverja na glavno vodilo.

Vanja Stojanović, Vegova Ljubljana

Sheet: /
File: Register - schematic.sch

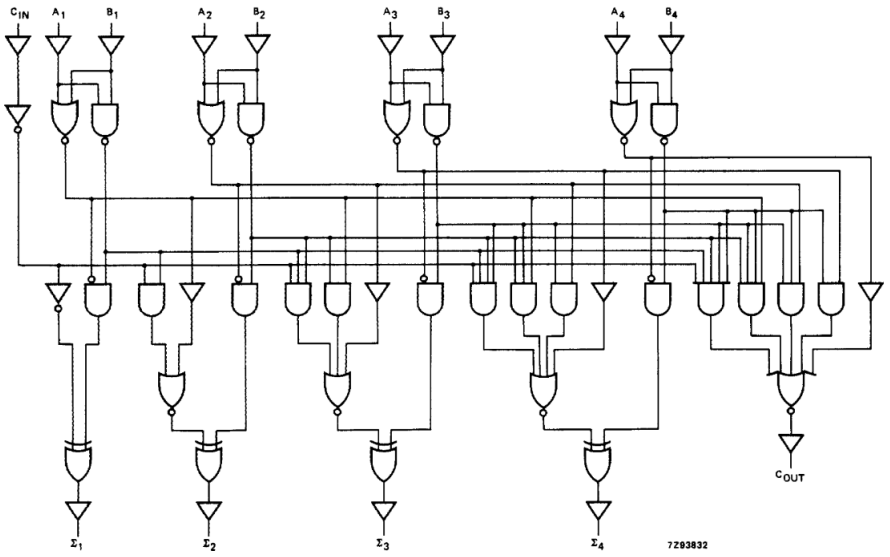
Title: 8-Bit D-Flip Flop Register

Size: A4	Date: 2021-01-10	**Rev: 5**
KiCad E.D.A. kicad (5.1.8)-1		Id: 1/1

1.1.3.3 ALU modul

ALU oz. Aritmetično-Logična Enota je modul, ki je sposoben izvajati matematično aritmetične operacije nad vsebinami iz registrov A in B. Med operacije sodita **seštevanje** in **odštevanje**. Vsebini registrov A in B sta neposredno povezani z ALU, kar pomeni, da ta pasivno vedno računa $A+B$ oz. $A-B$ in točno v tem vrstnem redu, nikoli $B+A$ oz. $B-A$. Med operacijami, sam ALU preklopi s pomočjo logičnega vezja, ki mu pomaga opraviti **eniški** in **dvojiški komplement** nad vsebino registra **B**. Tega aktiviramo s kontrolnim signalom SUBTRACT.

Modul je sestavljen iz 3 delov oz. komponent. Samega 8-bitnega polnega seštevalnika (ta je sestavljen iz dveh 74HC283 4-bitnih polnih seštevalnikov s *carry bitom*), tako kot register ima tudi ALU kontrolo dostopa do glavnega vodila in logično vezje za opravljanje eniškega in dvojiškega komplementa.

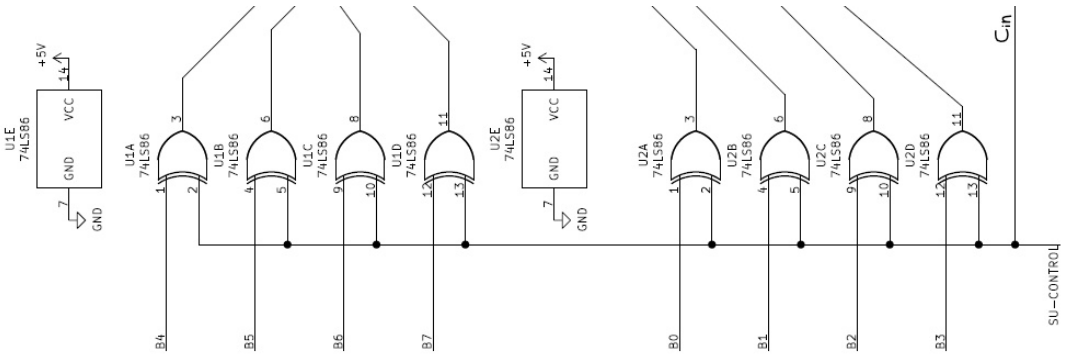


Slika 1.10: Logični diagram 74HC283 čipa

Slika 1.10 prikazuje logični diagram enega izmed 74HC283 integriranih vezij. Ta vzame, kot vhodni podatek 8-bitov informacij, razdeljenih v A in B vhode, kot dodaten vhodni podatek sprejme tudi C_{in}, kar je carry bit. Izhodi pa so seštevki bitov in dodaten carry bit C_{out}. Sistem 8-bitnega seštevalnika je enostavno sestaviti s 2 takima čipoma, iz Registra A izhodne od Q_0 do Q_3 linije povežemo s ujemajočimi A vhodi na seštevalniku 1 in izhode Q_4 do Q_7 na ujemajoče vhode seštevalnika 2. Isto naredimo s registrom B, le da izhode povežemo na B vhode seštevalnikov. Hkrati pa C_{out} bit 1. seštevalnika povežemo s vhodom C_{in} 2. seštevalnika, tako lahko ALU prenaša carry bit iz enega seštevalnika na drugega.

Dostop do glavnega vodila je realiziran z istim integriranim vezjem, kot pri registrih (*t. j.* 74LS245).

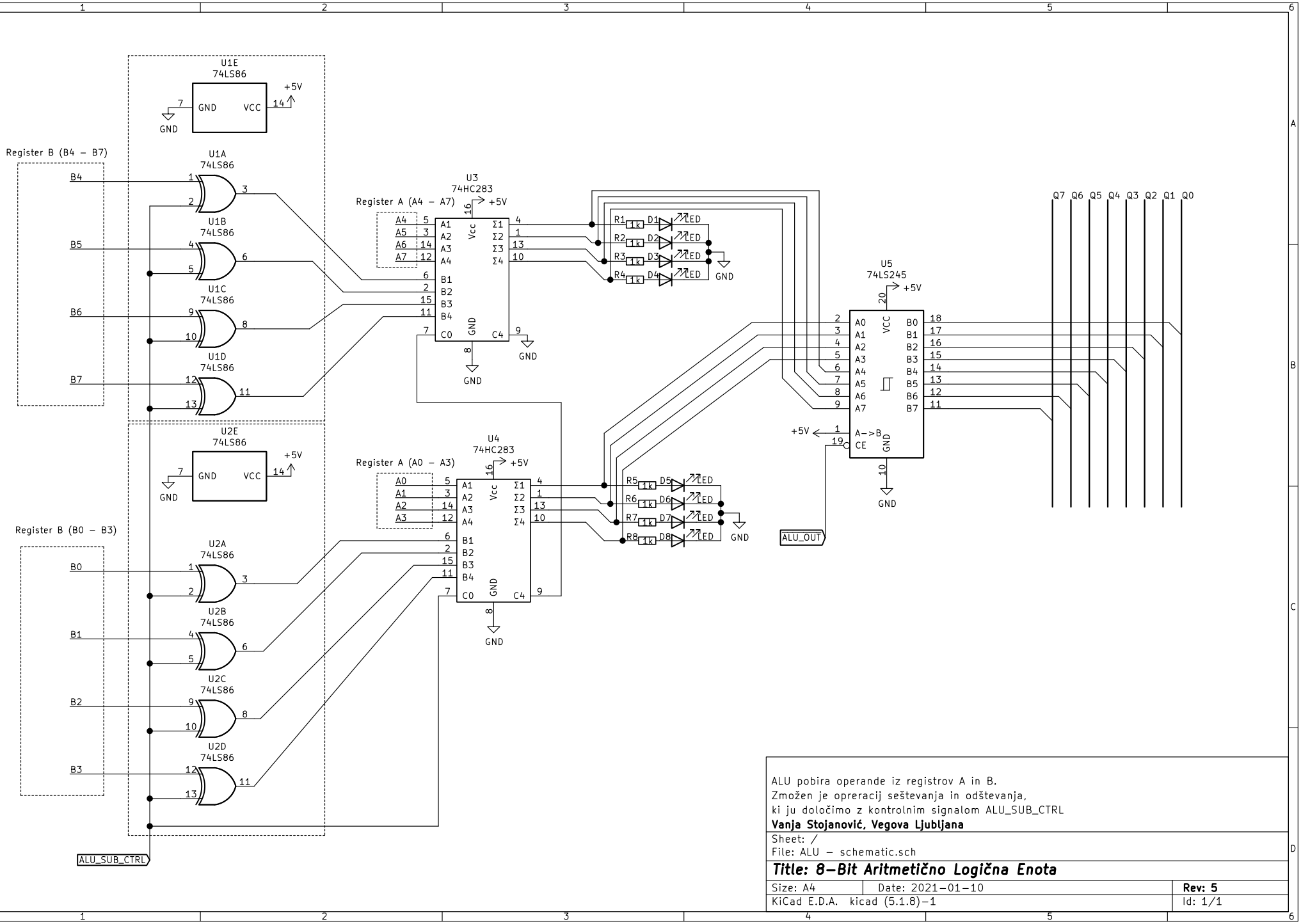
Tretje omenjena komponenta ALU-ja je logično vezje za izvajanje **eniškega** in **dvojiškega** komplementa. Vezje je sestavljeno iz 2 čipov 74LS86, ki sta četverna XOR logična vrata v integriranem vezju. Na B vhode čipa je povezana vsebina B registra ustrezno, na A vhode pa je zvezana kontrolna linija s signalom SUBTRACT (*SU-CONTROL*), prav tako je C_{in} 1. seštevalnika povezan s tem signalom.



Slika 1.11: Logični diagram logičnega vezja za kontrolo operacije

Tako efektivno dobimo optimiziran način za menjanje med operacijami seštevanja in odštevanja vsebine registrov.

Shema vezja:



ALU pobira operande iz registrov A in B.
Zmožen je opreracij seštevanja in odštevanja,
ki ju določimo z kontrolnim signalom ALU_SUB_CTRL

Vanja Stojanović, Vegova Ljubljana

Sheet: /
File: ALU – schematic.sch

Title: 8–Bit Aritmetično Logična Enota

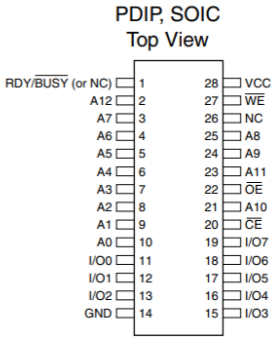
Size: A4	Date: 2021–01–10	**Rev: 5**
KiCad E.D.A. kicad (5.1.8)–1		Id: 1/1

1.1.3.4 Izhodni modul

Izhodni modul je namenjen prikazovanju podatkov računalnika, v okolju prijaznem formatu, *t. j.* desetišekem zapisu. Sestavljen je iz treh delov:

- EEPROM čip
- 7-segmentni prikazovalniki
- Multiplekser za prikazovalnike
- Izhodni register

EEPROM čip je 28C64, s 13 naslovnimi linijami od A_0 do A_{12} in 8 vhodno-izhodnih linij s WRITE ENABLE, OUTPUT ENABLE in CHIP ENABLE kontrolnimi signali.



Slika 1.12: Pinout diagram 28C64 EEPROM-a

EEPROM je sprogramiran za dekodiranje 8-bitnih vrednosti v decimalne na 7-segmentnih prikazovalnikih. Izhodi so povezani na 7-segmentne prikazovalnike s skupno katodo, glede na njihovo ustrezno zaporedje, tako da se na njemu prikaže prava desetiška vrednost.

Programator za EEPROM je enostaven in sestavljen iz mikrokrmilnika ARDUINO UNO in logičnim vezjem shift registrov, ki nadomestijo majhno število izhodnih enot mikrokrmilnika. Ta v skladu s kontrolnimi signali programira EEPROM (*t. j.* WRITE ENABLE ipd.). Na sliki 1.13 je izsek programa za ARDUINO UNO, ki v EEPROM zapiše podatke iz tabele $KODA\_7SEG$ (podatki so dolgi 1B), te vrednosti so pravilno kodirani podatki za desetiški prikaz na 7-segmentnih prikazovalnikih.

Multiplekser za prikazovanje je sestavljen iz 555-timer čipa v astabilnem vezju, ta preklaplja med tremi 7-segmentnimi s pomočjo 2-bitnem števcem, ki je le dvojna JK pomnilniška celica. Izhodi tega so povezani na $2 \rightarrow 4$ linijski demultiplekser. Tako dosežemo popolno multipleksiranje 7-segmentnih prikazovalnikov in izhodov EEPROM-a.

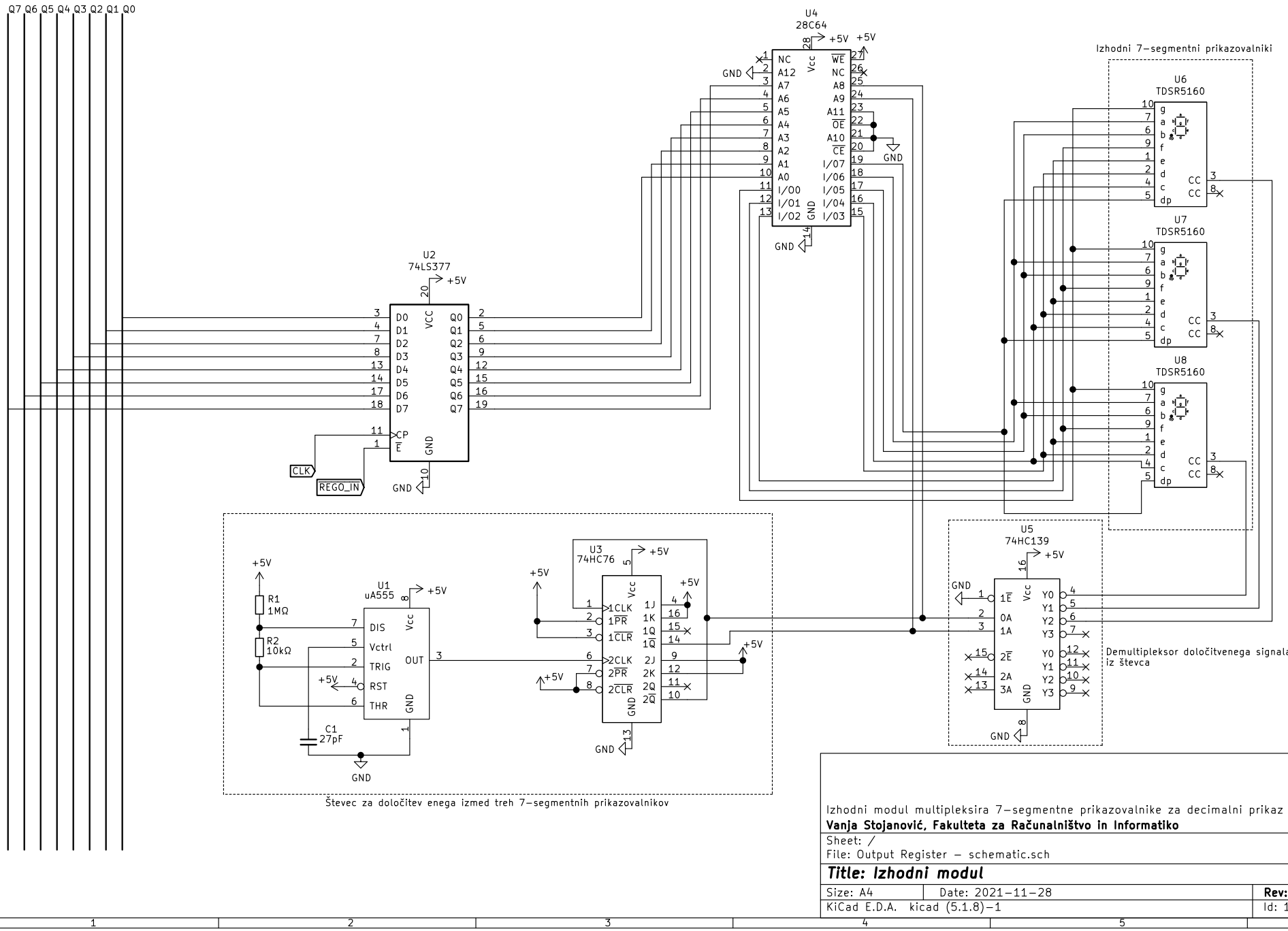
Glavno vodilo je povezano s naslovnimi vodili EEPROM-a, ki je sprogramiran tako, da glede na binarno vrednost, ki je na naslovnem vodilu, se prikaže tudi ustrezno pretvorjena desetiška vrednost na izhodih prikazovalnikih.

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);

  pinMode(SHIFT_SERIAL_DATA, OUTPUT);
  pinMode(SHIFT_CLK, OUTPUT);
  pinMode(SHIFT_SR_CLK, OUTPUT);

  digitalWrite(WE, HIGH);
  pinMode(WE, OUTPUT);

  byte KODA_7SEG[] = {0x7e, 0x30, 0x6d, 0x79, 0x33, 0x5b, 0x5f, 0x70, 0x7f, 0x7b};
  //                   0     1     2     3     4     5     6     7     8     9

  Serial.println("\nProgramiranje EEPROM-a");
  for(int ADDR = 0; ADDR < 255; ADDR++){
    WRITE_TO_CHIP(ADDR, KODA_7SEG[ADDR % 10]);
  }

  for(int ADDR = 0; ADDR < 255; ADDR++){
    WRITE_TO_CHIP(ADDR + 256, KODA_7SEG[(ADDR / 10) % 10]);
  }

  for(int ADDR = 0; ADDR < 255; ADDR++){
    WRITE_TO_CHIP(ADDR + 512, KODA_7SEG[(ADDR / 100) % 10]);
  }

  Serial.print("Branje EEPROM-a...\n");
  READ_CHIP_FROM_TO(0x00, 0x2000);

}
```

Slika 1.13: Program za programiranje EEPROM-a

Izhodni register je le povezan z glavnim vodilom, preko katerega pridobiva podatke za prikaz, na 7-segmentnih prikazovalnikih. Vrednost v registru je dejansko zaklenjena, dokler jo ne prepiše nova vrednost iz glavnega vodila. Dostop do registra upravlja kontrolni signal **$\overline{\text{REGO\_IN}}$**.

Shema vezja:



Izhodni modul multipleksira 7-segmentne prikazovalnike za decimalni prikaz

Vanja Stojanović, Fakulteta za Računalništvo in Informatiko

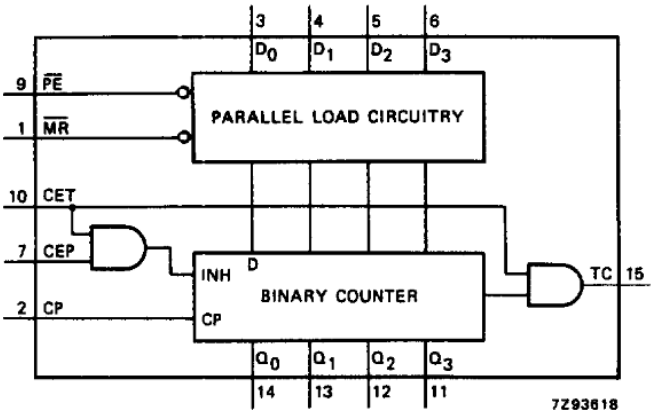
Sheet: /
File: Output Register - schematic.sch

Title: Izhodni modul

Size: A4	Date: 2021-11-28	**Rev: 1**
KiCad E.D.A. kicad (5.1.8)-1		Id: 1/1

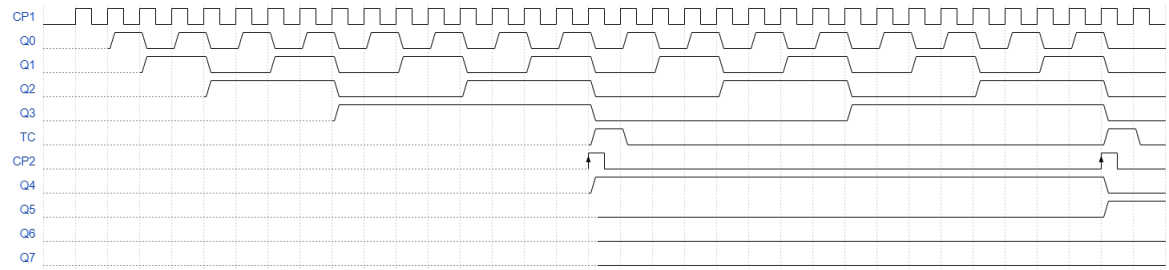
1.1.3.5 Modul programskega števca

Programski števec je register, ki sledi kje v programu se trenutno nahajamo oz. kje v pomnilniškemu prostoru. Programi se bodo začeli izvajati od pomnilniškega prostora 0x00 *t. j.* oz. prvega naslovnega prostora, saj je računalnik preprost in ne vsebuje nobenih zagonskih programov ali dodatne programske opreme. Ta je sestavljen iz dveh 4-bitnih števcev, kar skupaj sestavlja 8-bitni števec, to pomeni, da so lahko programi dolgi **256 naslovnih prostorov**. Integrirano vezje uporabljeno za števec je 74HC163, medtem ko je kontrola dostopa do samega vodila, ponovno isto izvedena s 74LS245, kot pri ostalih modulih.



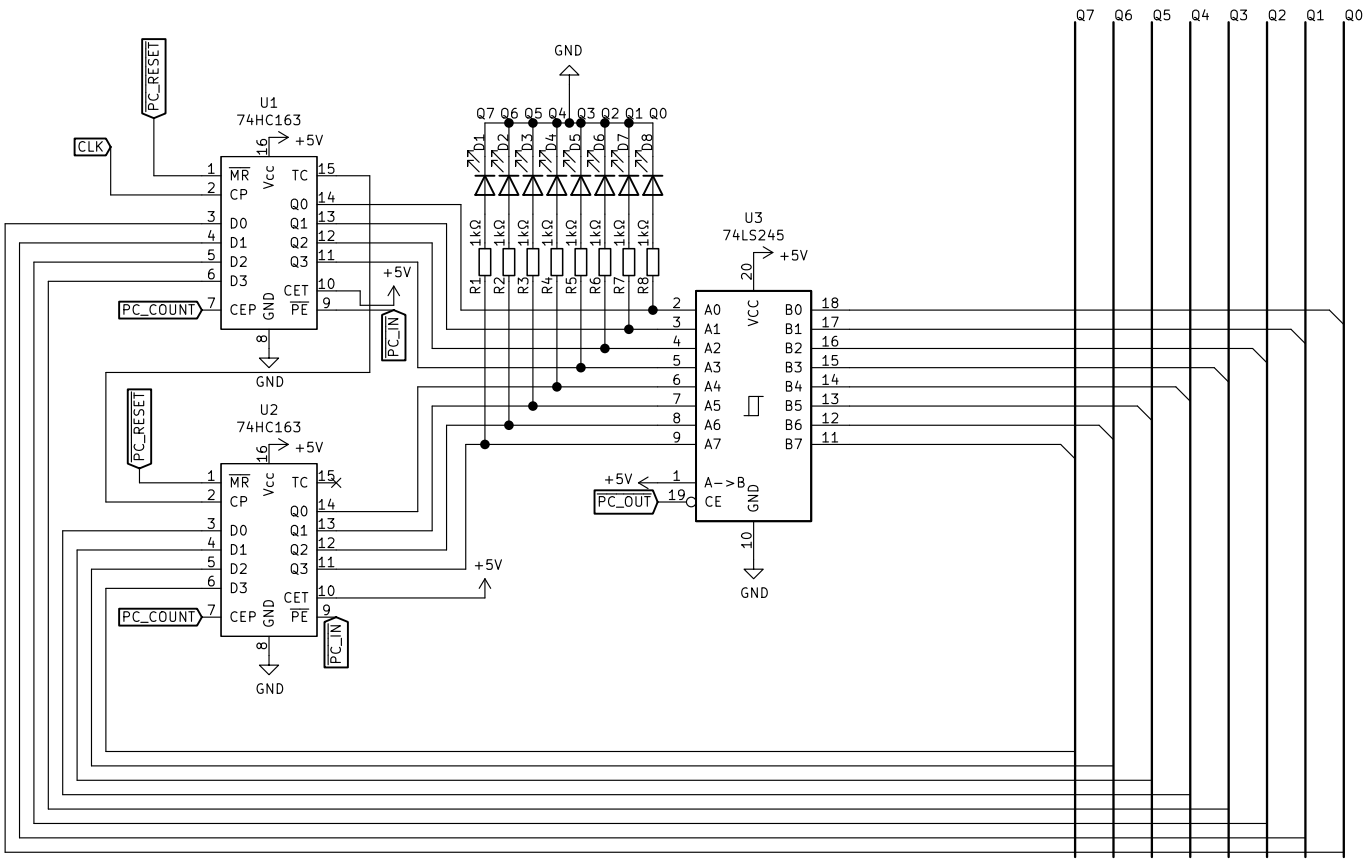
Slika 1.14: Logični diagram 74HC163 čipa

Vhodi $D_0 - D_3$ na obeh števcih so ustrezno povezani s glavnim vodilom, te vhodi so namenjeni določanju vrednosti števcu, od katere naprej naj bi štel, to je uporabno pri raznih programskih skokih. Medtem so pa izhodi $Q_0 - Q_3$ povezani s 74LS245 vezjem za kontrolo dostopa do vodila. Ko prvi seštevalnik prešteje do konca svojega dosega (*t. j.* 1111) spusti signal iz izhoda TC (*angl.* Terminal Count) ta je povezan z vhodom CET (*angl.* Count Enable Carry Input), ki je povezan s Clock vhodom drugega seštevalnika. Na koncu dobimo tak signalni spekter:



Slika 1.15: Spekter signalov programskega števca

Na spektru so prikazani izhodi prvega in drugega seštevalnika v vrstnem redu od $Q_0 - Q_7$, CP1 je urin pulz iz urinega modula, medtem ko CP2 signal sproža TC prvega seštevalnika, tako se nadaljuje štetje v drugem. Kontrolni signal CET mora biti povezan na logično "1", medtem ko CEP (*angl.* Count Enable Input) kontroliramo s kontrolnim signalom COUNT, kar efektivno določa ali programski števec šteje ali ne. **Shema vezja:**



Programski števec s 8-Bit omogoča sledenje programu in preskoke v kodi.

Vanja Stojanović, Vegova Ljubljana

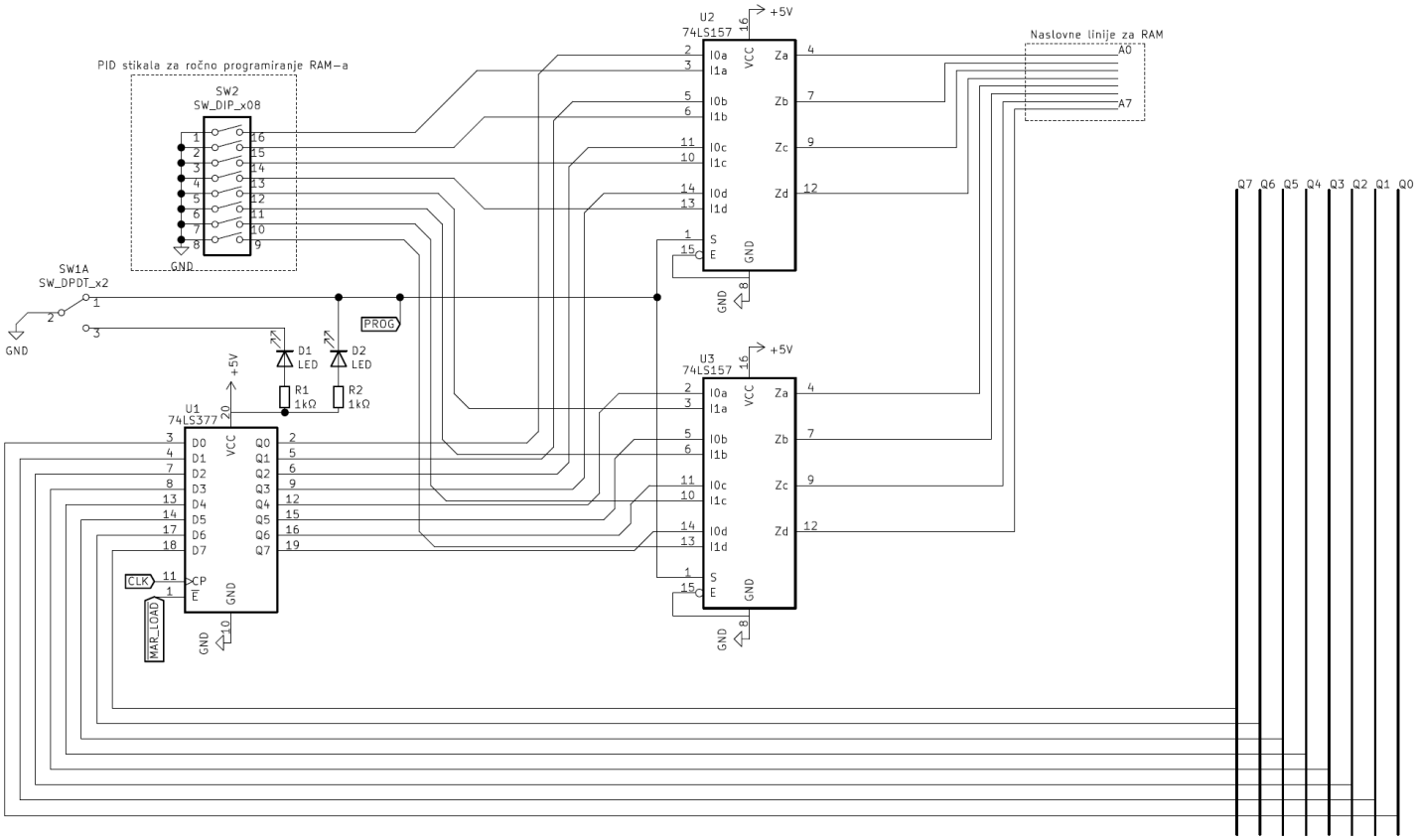
Sheet: /
File: Program Counter.sch

Title: 8-Bit Programski Števec

Size: A4	Date: 2020-12-07	**Rev: 3**
KiCad E.D.A. kicad (5.1.8)-1		Id: 1/1

1.1.3.6 Modul pomnilniškega registra

Modul pomnilniškega registri se dejansko sklicuje na MAR register (*angl.* Memory Address Register) za delovanje. MAR je sestavljen iz treh komponent samega registra, ki je isti čip kot za splošni register *t. j.* 74LS377, 8 DIP stikal, ki simulirajo ročno vnašanje naslovne vrednosti v pomnilnik in multiplekser, ki povezuje dva ločena signala v enega. Ročno vnašanje je namenjeno dejanskemu programiranju računalnika, točno določene vrednosti v binarnem sestavu vnesemo binarne vrednosti ukazov in operandov, tako vprogramiramo vrednost na točno določen naslov.

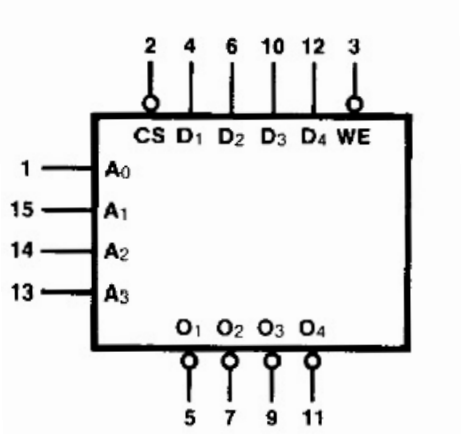


Slika 1.16: Shema MAR registra

Izhod multiplekserja kontroliramo s stikalom SW1A, PROG bo signal, ki nam pove ali imamo MAR v *t. r.* "programing" načinu ali v običajnem načinu delovanja. PID stikala so vezana na zemljo oz. 0V, saj ima multiplekser 74LS157, vhode privzetno zvezane na visoko stanje (*t. j. angl.* pulled high). Linije $Q_0 - Q_7$ predstavljajo glavno podatkovno vodilo.

1.1.3.7 Modul glavnega pomnilnika

Glavni pomnilnik oz. računalnikov RAM (*angl.* Random Access Memory), je sestavljen iz dveh 4-bitnih statičnih pomnilnikov 74LS189, ki skupaj sestavljata 8-bitni RAM. En sam čip je 64-bitni RAM organiziran, kot **16 besed po 4-bite**, torej sta dva taka čipa 2048-bitni RAM organiziran, kot **256 besed po 8-bitov**. Poleg samega statičnega pomnilnika, ima 74LS189 invertirane izhode, zato jih moramo dodatno negirati s logičnim vezjem, to je realizirano s 74LS05 heksalnimi NOT vrati oz. negatorji. Izhodi tega so nato preko 74LS245 povezani s glavnim vodilom.

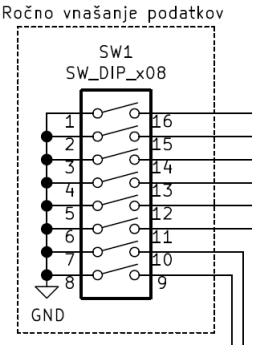


Slika 1.17: Logični diagram 74LS189 čipa

Naslovne linije od $A_0 - A_7$ (preostale 4 so del drugega RAM čipa) so povezane s ustreznimi linijami iz MAR-ovega multiplekserja za določanje naslova, kot je vidno na sliki 1.16. Medtem so podatkovne linije $D_0 - D_7$ povezane s nekakšnim MDR (*angl.* Memory Data Register), samo brez dejanskega registra (zato ga bomo imenovali MDM *t. j. angl.* Memory Data Module). Ta je sestavljen iz treh komponent;

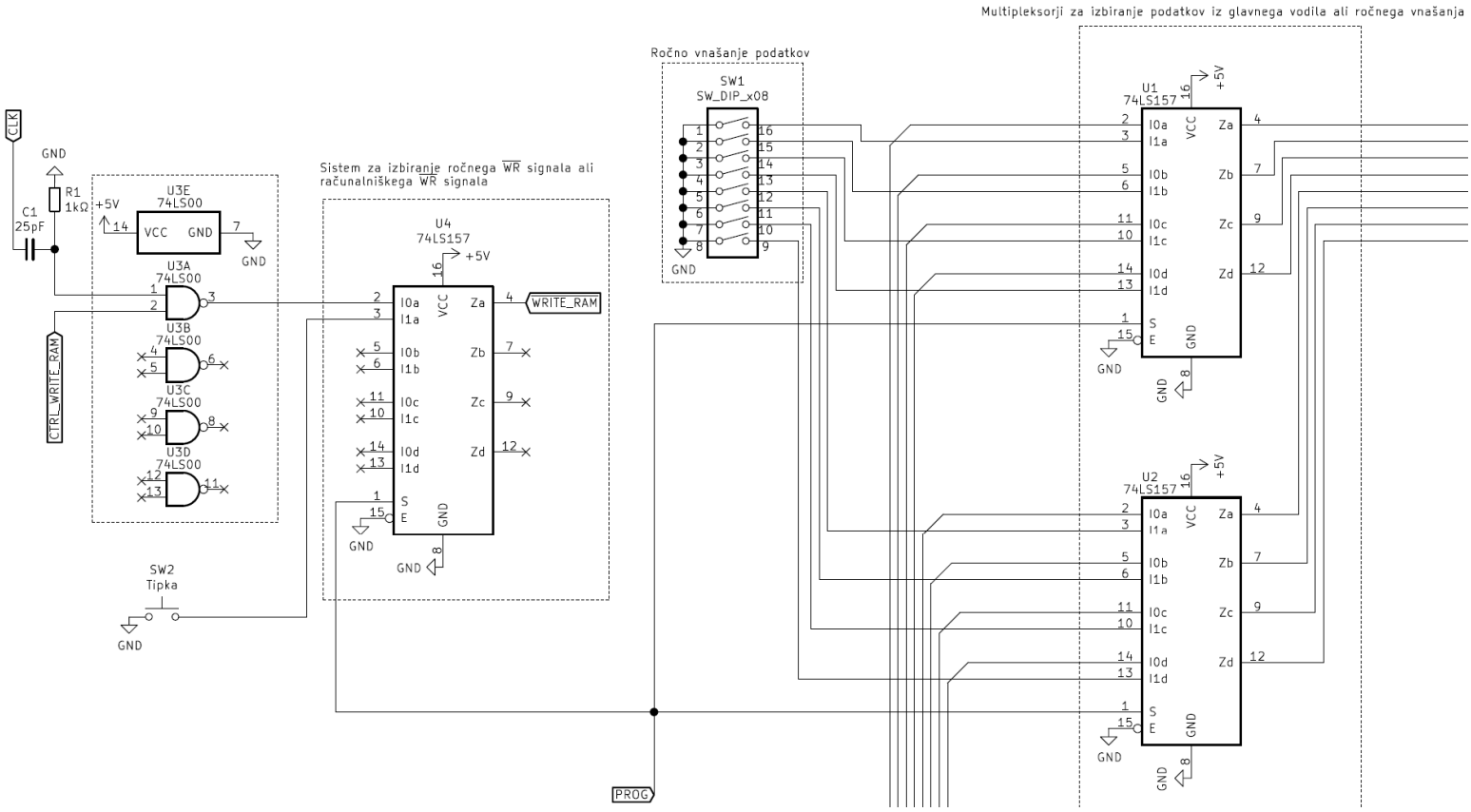
- DIP stikal za ročno vnašanje podatkov v RAM
- Sistem za izbiranje ročnega vnašanja ali branja glavnega vodila
- Logično vezje za izvajanje pisalnega cikla

DIP stikala so enostaven koncept, le 8 stikal, ki so privzeto povezana z zemljo oz. logično "0", drugi konec pa je povezan s sistemom za izbiranje (glej sliko 1.18). Sistem za izbiranje je dejansko sestavljen iz dveh multiplekserjev, istih ki smo jih uporabili že prej, to so 74LS157, 2-linijski v 1-linijski multiplekserji. Prve linije vhodov v multiplekserje povezujejo glavno vodilo, medtem ko druge linije povezujejo sistem za ročno programiranje. Sistem za izbiranje je neposredno povezan s logičnim vezjem za izvajanje pisalnega cikla, saj skupaj sprožijo preklop med branjem glavnega vodila v branje ročno vnešenih vrednosti.



Slika 1.18: DIP stikala za ročno programiranje

Na spodnji sliki (1.19) so prikazani vsi deli MDM modula, od ročnega vnašanja podatkov, sistema za izbiranje ročnega vnašanja ali branja glavnega vodila do Logičnega vezja za izvajanja pisalnega cikla:



Slika 1.19: Diagram vezja Memory Data Modula

Pini 1 vseh multiplekserjev, v MDM-ju in MAR-u so povezani skupaj preko PROG signala, ta je v visokem logičnem stanju, ko izberemo način programiranja, kot omenjeno in skicirano v poglavju *1.1.3.6 Modul pomnilniškega registra*. Integrirano vezje U4 na sliki 1.19, izbere vhode I1, ko smo v načinu programiranja. $\overline{WE}$ signal je aktivno nizek, kar pomeni, da za njegovo aktiviranje in posledično zagon pisalnega cikla na RAM-u, potrebujemo nizek logičen signal. 74LS157 multiplekser ima na vhodih pull-up upor, kar pomeni, da je vhod privzeto v visokem stanju, ko pritisnemo tipko SW2 (kot prikazano na sliki 1.19) se na vhodu in izhodu pojavi logična "0" in cikel pisanja se izvede. V tem ciklu RAM na naslov, ki smo ga nastavili z MAR registrom, vpiše podatke, ki smo jih ročno nastavili v MDM modulu.

Poleg ročnega vnašanja podatkov in izvajanja pisalnega cikla, lahko tudi računalnik sam izvede isti proces s kontrolnim signalom CTRL_WRITE_RAM. Ta je skupaj s urinim pulzom povezan v NAND logična vrata, saj želimo, da se pisalni cikel izvede ko je kontrolni signal logična "1" in hkrati v naslednjem urinem ciklu. Vrata so NAND saj želimo ravno obratno od AND vrat, ko sta visoka oba signala za pisanje (*t. j.* urin pulz in CTRL_WRITE_RAM) potrebujemo na izhodu logično "0" zaradi $\overline{WE}$ aktivno nizkega signala.

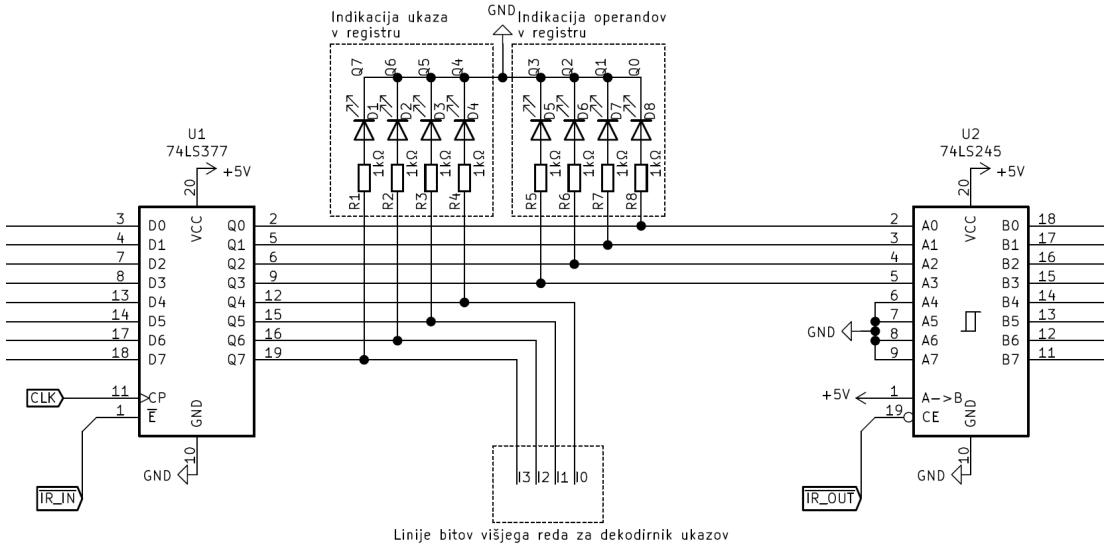
Urin pulz je s NAND vrati povezan preko RC vezja, čigar čas polnjenja mora biti oz. je priporočena vrednost 25ns. Razlog tega je sledeč; dokler se kondenzator polni, je na našem vhodu od NAND vrat logična "1", ta čas je v dokumentaciji 74LS189 RAM čipa definiran, kot tipični čas za aktiviranje pisalnega cikla preko $\overline{WE}$ vhoda. Čas za polnjenje kondenzatorja v RC vezju je definiran kot:

$$t = R \cdot C$$

Pri čemer je R vrednost upora v ohm-ih(Ω), C vrednost kondenzatorja v faradih(F) in t čas polnjenja v sekundah(s). Pisalni cikel se torej izvede v tem času.

1.1.3.8 Modul ukaznega registra

Ukazni register, je specializiran register in za razliko od splošnih v njemu ne hranimo podatkov, saj ima posebno nalogo. Njegov namen je pridobiti ukaz in njegov ustrezen operand iz pomnilnika, kjer se nahaja program, in dekodirati ukaz in operand v strojno kodo, ki potem pove kontrolni logiki, kako izvesti ukazni cikel. Register je 8-biten, kot vsi ostali in uporablja isto integrirano vezje 74LS377, kot splošni registri in MAR. Prav tako uporablja kontrolo dostopa do vodila s 74LS245 čipom, a nanj pošilja le svoje 4 bite nižjega reda (*t. j.* lokacijo operanda).



Slika 1.20: Diagram ukaznega registra

Podobno, kot ostali registri, tega kontrolirata 2 kontrolna signala, za vhod oz. branje podatkov iz glavnega vodila v register in izhod oz. pošiljanje podatkov na glavno vodilo.

Logično je register razdeljen na sledeči način:

0000	0000
Biti višjega reda	Biti nižjega reda

Biti višjega reda bodo predstavljali kodirane ukaze, ki bodo v naboru. Medtem ko bodo biti nižjega reda predstavljali naslov v pomnilniku, kjer se bodo nahajali operandi. Izvajanje je realizirano tako zaradi omejenega števila registrov, tako bo potrebno, kot vsak ukaz, sprogramirati tudi vsak operand v pomnilnik ročno.

2 Kontrolna logika in zbirni jezik

Zbirni jezik (*angl.* assembly language) je tesno povezan z arhitekturo računalnika, še tesneje pa z njegovo kontrolno logiko. Računalnik ima vnaprej določen nabor ukazov, kateri so v posebnih pomnilnikih zapisani v strojni kodi oz. v dejanskih 1 in 0, torej binarni kodi. Naš računalnik dekodira vsak ukaz njegove operande, ki jih prebere v pomnilniku, ko ve kateri ukaz se mora izvesti, jih izvede s **vnaprej določenimi zaporedji ugašanja in prižiganja danih kontrolnih signalov**. Torej je ukaz sestavljen iz teh manjših zaporedij, primer;

mov a , b

Program je enostaven, in opravlja premik vsebine registra "b" v register "a". Ampak zaporedje signalov, ki morajo v harmoniji izvesti to operacijo je veliko bolj zahtevno. Tem zaporedjem iz katerih so ukazi sestavljeni pravimo **mikrokoda**. Za naš primer bi se ta glasila; postavi vsebino registra "b" na glavno vodilo, nato naj register "a" prebere vsebino glavnega vodila, register "a" naj neha brati glavno vodilo in na koncu register "b" naj vsebino ne pošilja na glavno vodilo. Skupaj 4 koraki, ki se morajo opraviti in s tem 4 kontrolni signali, ki jih mora kontrolna logika upravljati za izvedbo operacije.

Mikrokoda se zapisuje v korakih, po katerih se mora izvesti oz. **strojnih ciklih**. Strojni cikel je sestavljen iz večih urinih ali *angl.* clock ciklov, in definira čas v katerem računalnik iz pomnilnika pridobi ukaz in operande (***fetch***), jih dekodira s pomočjo ukaznega registra (***decode***), nato izvede s kontroliranjem vseh kontrolnih signalov in uporabo mikrokode (***execute***) in na koncu še rezultate shrani ali pa zgolj samo pokaže na izhodnem modulu (***store***).

To poglavje (*Kontrolna logika in zbirni jezik*) je namenjeno grajenju in opisu kontrolne logike, programiranje mikrokode, kodiranju ukazov novega zbirnega jezika v operacijsko kodo, pojasnilu poteka in izvajanja programa, ter pisanje nekaj osnovnih teoretičnih programov.

2.1 Kontrolna beseda in logika

Kontrolno logiko si lahko predstavljamo, kot možgane računalnika, ta je nekakšen center za opravljanje in izvrševanje dejanskih ukazov, ki mu jih zadamo. Brez nje, bi bil register zgolj register, ALU zgolj ALU itn. saj na enem mestu zbere vse kontrolne signale, in jih, tako rečeno, kontrolira avtomatsko, namesto nas.

Prvo je potrebno zapisati vse kontrolne signale in njihove funkcije, za pregled nad potrebami naše kontrolne logike. Te so navedeni v spodnji tabeli:

	Kontrolni signal	**Krajšava**	**Pomen kratice**	**Opis kontrolne mehanike**
1.	**HLT**	HLT	HALT	S logičnim vezjem ustavi urin pulz, ko je v visokem logičnem stanju. Efektivno ustavi računalnik.
2.	$\overline{\textbf{REGA\_IN}}$	$\overline{\texttt{RA}_i}$	Register A Input	Omogoči branje glavnega vodila v registru A, s aktivno nizkim logičnim signalom.
3.	$\overline{\textbf{REGA\_OUT}}$	$\overline{\texttt{RA}_o}$	Register A Output	Omogoči pisanje vsebine registra A na glavno vodilo, s aktivno nizkim logičnim signalom.
4.	$\overline{\textbf{REGB\_IN}}$	$\overline{\texttt{RB}_i}$	Register B Input	Omogoči branje glavnega vodila v registru B, s aktivno nizkim logičnim signalom.
5.	$\overline{\textbf{REGB\_OUT}}$	$\overline{\texttt{RB}_o}$	Register B Output	Omogoči pisanje vsebine registra B na glavno vodilo, s aktivno nizkim logičnim signalom.
6.	$\overline{\textbf{REGO\_IN}}$	$\overline{\texttt{RO}_i}$	Register Out Input	Omogoči branje glavnega vodila v izhodnem registru, s aktivno nizkim logičnim signalom.
7.	$\overline{\textbf{ALU\_OUT}}$	$\overline{\texttt{ALU}_o}$	ALU Output	Omogoči pisanje vsebine Aritmetično-Logične enote na glavno vodilo, s aktivno nizkim logičnim signalom.
8.	**ALU_SUB_CTRL**	$\texttt{ALU}_s$	ALU Subtract Control	Omogoči izvedbo dvojiškega in eniškega komplementa nad vsebino registra B in izvede odštevanje (A - B).
9.	$\overline{\textbf{PC\_IN}}$	$\overline{\texttt{PC}_i}$	Program Counter Input	Omogoči branje glavnega vodila v programskem števcu, s aktivno nizkim logičnim signalom.
10.	$\overline{\textbf{PC\_OUT}}$	$\overline{\texttt{PC}_o}$	Program Counter Output	Omogoči pisanje vsebine programskega števca na glavno vodilo, s aktivno nizkim logičnim signalom.
11.	$\overline{\textbf{PC\_RESET}}$	$\overline{\texttt{PC}_r}$	Program Counter Reset	Ponastavi vrednost programskega števca na $0_{(10)}$.
12.	**PC_COUNT**	$\texttt{PC}_c$	Program Counter Count	V enem urinem ciklu inkrementira vrednost programskega števca za $1_{(10)}$.
13.	$\overline{\textbf{MAR\_LOAD}}$	$\overline{\texttt{MAR}_l}$	MAR Register Load	Omogoči branje glavnega vodila v Pomnilniško-naslovnem (MAR) registru, s aktivno nizkim logičnim signalom.
14.	$\overline{\textbf{RAM\_OUT}}$	$\overline{\texttt{RAM}_o}$	RAM Output	Omogoči pisanje vsebine trenutnega naslovljenega pomnilniškega prostora na glavno vodilo, s aktivno nizkim logičnim signalom.

	Kontrolni signal	Krajšava	Pomen kratice	Opis kontrolne mehanike
15.	$\overline{\textbf{WRITE\_RAM}}$	$\overline{\texttt{RAM}_w}$	Write to RAM	Aktivira pisalni cikel v RAM modulu, v katerem vsebino iz MDD modula na trenutni pomnilniški naslov zapiše v pomnilnik.
16.	$\overline{\textbf{IR\_IN}}$	$\overline{\texttt{IR}_i}$	Instruction Register Input	Omogoči branje glavnega vodila v ukaznem registru, s aktivno nizkim logičnim signalom.
17.	$\overline{\textbf{IR\_OUT}}$	$\overline{\texttt{IR}_o}$	Instruction Register Output	Omogoči branje glavnega vodila v registru A, s aktivno nizkim logičnim signalom.

Slika 2.1: Nabor kontrolnih signalo

Na podlagi teh signalov, lahko začnemo graditi izhode kontrolne logike, ki bodo logična "1" ali "0", glede na funkcijo, katero morajo opraviti. To zaporedje izhodov kontrolnih signalov, imenujemo **kontrolna beseda** in je v našem primeru *17-bitna*, zaradi 17 kontrolnih signalov.

Opomba (Sekvenca kontrolnih signalov). Zaporedje kontrolnih signalov in oštevilčenje vrstice v sliki 2.1 je naključno in ni povezano s pomembnostjo v sami kontrolni besedi.

Zgradba kontrolne besede je odvisna od naše domišljije, saj si jo lahko izmislimo sami. Vendar naj sledi nekemu logičnem zaporedju, zato bo strukturirana na sledeči način.

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HLT	$\overline{\texttt{RA}_i}$	$\overline{\texttt{RA}_o}$	$\overline{\texttt{RB}_i}$	$\overline{\texttt{RB}_o}$	$\overline{\texttt{RO}_i}$	$\overline{\texttt{ALU}_o}$	$\texttt{ALU}_s$	$\overline{\texttt{PC}_i}$	$\overline{\texttt{PC}_o}$	$\overline{\texttt{PC}_r}$	$\texttt{PC}_c$	$\overline{\texttt{MAR}_l}$	$\overline{\texttt{RAM}_o}$	$\overline{\texttt{RAM}_w}$	$\overline{\texttt{IR}_i}$	$\overline{\texttt{IR}_o}$

Slika 2.2: Kontrolna beseda v obliki bitnega polja

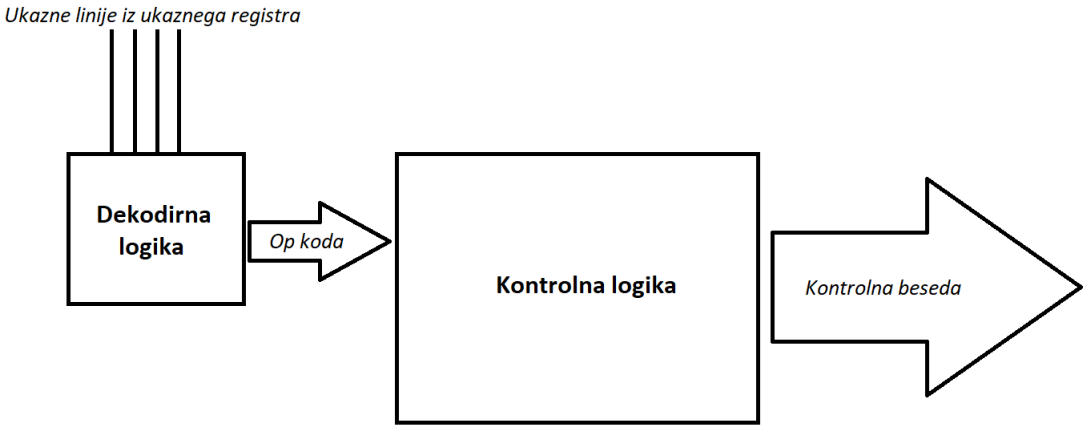
Kaj sedaj lahko počnemo s sestavljeno kontrolno besedo? Z njo lahko dejansko *programiramo in avtomatiziramo računalnik*, da izvaja razne operacije, ki mu jih zadamo in katere je tudi sposoben izvesti. Vsake toliko časa se kontrolna beseda spremeni, da opravi neko delno operacijo, vendar zakaj delno? Kot omenjeno so ukazi in programi razdeljeni v manjše enote izvršitve, ki jim pravimo **mikrokoda**, ker procesor ne zna kar sešteti ali naložiti nekega števila, za to potrebuje definiran postopek zaporednih kontrolnih signalov, ki se morajo aktivirati. S kontrolno besedo lahko dosežemo ravno to, v vsakem trenutku lahko računalniku povemo kateri kontrolni signali so aktivni in kateri so neaktivni. Torej je kontrolna beseda **osnovna enota kontrole** v našem računalniku.

Samo kontrolno logiko, *t. j.* torej nekakšno *logično vezje*, ki nam bo v nekem določenem koraku CPU cikla proizvedlo pravilno logično besedo, le ta izvede pravilno. To lahko realiziramo s običajno tehniko logičnih vezij v digitalni tehniki, zapišemo logično funkcijo, ki jo želimo in preko preračunavanja dobimo izraz boolove algebre, ki ga lahko enačimo s logičnimi vrati. Dejansko pa lahko to ogromno logično vezje zamenjamo s **EEPROM čipom**, podobno kot pri izhodnem registru (*poglavje 1.1.3.4 Izhodni modul*). Zadeva je realizirana tako, da ko dekodirna logika operacijo dekodira v operacijsko kodo, oz. 4-bitno kodiranje ukaza, bo ta postopoma brala EEPROM kontrolne logike, v kateri se bo nahajalo zaporedje "1" in "0", tako da bodo skupaj sestavili zgoraj definirano kontrolno besedo.

2.1.1 Podlaga za dekodirno logiko

Dekodirna logika, je tesno povezana s kontrolno logiko in je konceptualno tudi združena v naši arhitekturi. Kontrolna logika, računalniku pove kako nekaj narediti, medtem pa dekodirna logika dekodira program, ki smo mu ga podali in računalniku pove kaj mora narediti (*t. j.* operacijo). To bomo počeli s **skrajšanim CPU ciklom, kjer se navidezno "združita" decode in execute koraka**. Decode je v našem primeru zgolj naslovitev s strani ukaza in koraka strojnega ukaza.

Kot je definirano v ukaznem registru (*poglavje 1.1.3.8 Modul ukaznega registra*), dekodirna logika vhod prejme iz 4 bitov višjega reda, ki nam povedo za katero operacijo gre. To nam lahko opiše sledeča shema:



Slika 2.3: Logični diagram dekodirne in kontrolne logike

2.1.2 Izvajanje mikrokode in shema kontrolne logike

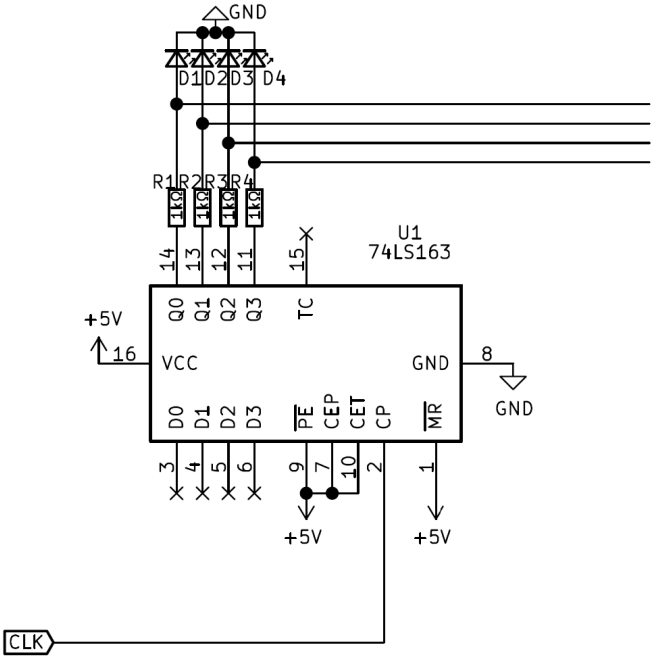
Mikrokoda mora biti za vsak ukaz definirana posebej, a ta le sledi CPU ciklu. To v praksi pomeni, da je fetch, decode in store proces skorajda enak pri vseh operacijah. Ukazi so definirani v naslednjem podpoglavju (*2.2 Lasten zbirni jezik*). Prvo definirajmo splošno mikrokodo fetch cikla:

fetch					
$\overline{\texttt{PC}_o}$	$\overline{\texttt{MAR}_l}$				
$\overline{\texttt{RAM}_o}$	$\overline{\texttt{IR}_i}$	$\texttt{PC}_c$	$\texttt{PC}_o$	$\texttt{MAR}_l$	
			$\texttt{RAM}_o$	$\texttt{IR}_i$	$\overline{\texttt{PC}_c}$

Slika 2.4: Mikrokoda fetch cikla

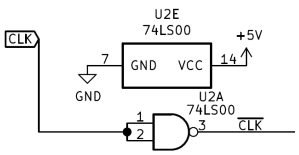
Na levi strani diagrama so dejanski kontrolni signali, ki jih moramo aktivirati, na levi strani pa njihove negacije, ki se aktivirajo en urin cikel kasneje, tako vrnejo računalnik v nevtralno stanje. V praksi ne bomo rabili ročno izvesti negacije, saj se ta izvede, ko se na izhodu EEPROM-a le zapiše logično nasproten signal.

Urin pulz je najosnovnejši interval v katerem računalnik nekaj "počne". En sam ukaz se ne more izvesti v enem urinem ciklu, kar je logično. Zato se cikel v katerem se izvede ukaz, imenuje **ukazni cikel**, ta je sestavljen iz manjših ciklov imenovani **strojni cikli**. To bi pomenilo, da *fetch* traja en strojni cikel. A v strojnem ciklu moramo vedeti, v katerem koraku se nahajamo, kot je narisano v zgornji ilustraciji (slika 2.8), so te koraki implicirani s vrsticami. Najenostavnejša rešitev bi bila, da bi kontrolna logika imela svoj neodvisni števec, ki bi sledil tem korakom in se ponastavil vsak strojni cikel. "*Kontrolni števec*" naj bo **4-biten**, saj noben strojni ukaz ne bo presegal 16 korakov.



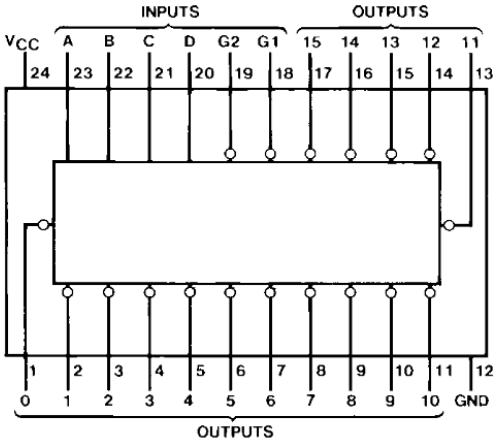
Slika 2.5: Shema kontrolnega števca

Poleg kontrolnega števca, potrebuje kontrolna logika tudi svoj **urin pulz**, zato da "*pripravi*" kontrolno besedo preden jo pošlje po vsem računalniku in njen vpliv sproži urin pulz računalnika. Dejansko hočemo, da je kontrolni urin pulz le **za malo zamaknjen** od originalnega, zato lahko enostavno tega invertiramo s **negatorjem** in ga povežemo s CP vhodom kontrolnega števca.



Slika 2.6: Negiran CLK signal

Da bi še bolj poenostavili logično vezje in programiranje EEPROM-a kontrolne logike, lahko vnaprej določimo točno na katerem koraku v strojnem ciklu smo, s **demultiplekserjem izhodnih bitov kontrolnega števca**. Demultiplekser bo 4-linijski v 16-linijski, vendar prirejeni saj nikoli **ne bomo presegli 4 korake**. Uporabili bomo 74LS154 integrirano vezje.



Slika 2.7: Diagram povezav 74LS154 čipa

Edina kompenzacija, ki jo bomo morali narediti je prilagoditev EEPROM-a na invertirani izhode demultiplekserja. Hkrati pa lahko izhod 5 na tem čipu uporabimo, kot "*master reset*" na kontrolnem števcu, tako da ta **ne demultipleksira preko 4**.

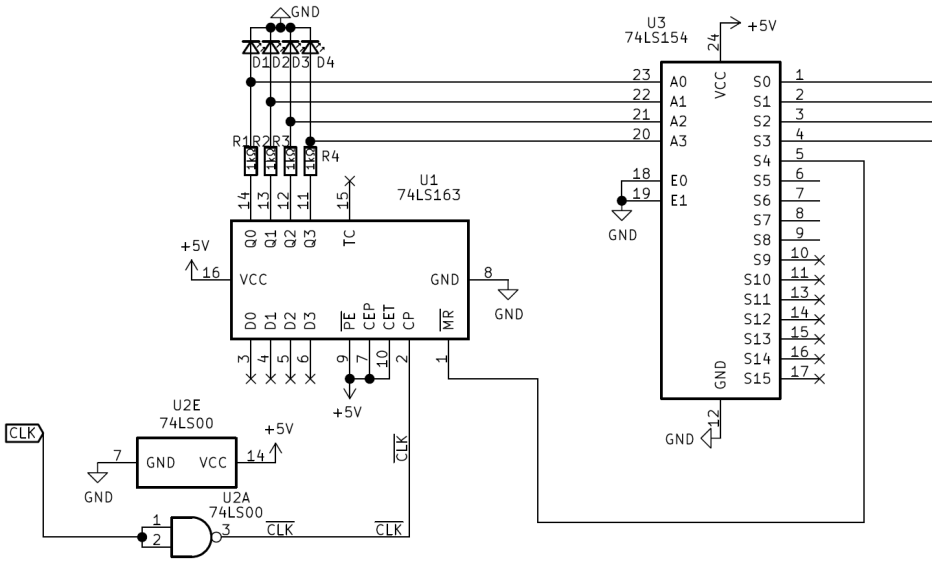
Korake v strojnem ciklu bomo označili s oznako $\mathtt{S}$ in indeksom $n \in \{0, 1, 2, 3\}$, torej bi fetch signal označili:

	fetch					
$\mathtt{S}_0$		$\overline{\mathtt{PC}_o}$	$\overline{\mathtt{MAR}_l}$			
$\mathtt{S}_1$	$\overline{\mathtt{RAM}_o}$	$\overline{\mathtt{IR}_i}$	$\mathtt{PC}_c$	$\mathtt{PC}_o$	$\mathtt{MAR}_l$	
				$\mathtt{RAM}_o$	$\mathtt{IR}_i$	$\overline{\mathtt{PC}_c}$

Slika 2.8: Mikrokoda fetch cikla

Opomba (Zadnji korak). Zadnjega koraka, kjer ni signalov, ki jih moramo aktivirati, torej so samo še invertirani signali prejšnjih korakov, **ne označujemo**, saj se mora stanje kontrolne besede po fetch ciklu, postaviti glede na sledeči ukaz ali v kakšnem drugem primeru, se postavi na nevtralno stanje računalnika.

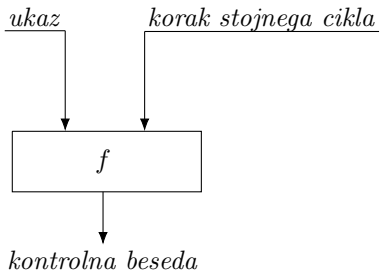
Torej vezje s kontrolnim števcem in multiplekserjem izgleda tako:



Slika 2.9: Shema kontrolnega števca in multiplekserja

2.1.2.1 EEPROM-i in ustvarjanje kontrolne besede

S nadzorom nad strojnim ciklom in posledično ukaznim ciklom, lahko začnemo s ustvarjanjem logičnega vezja za kontrolno logiko, ali v našem primeru, programiranje EEPROM-a. A da bi vedeli kaj vprogramirati na razne lokacije našega čipa, moramo sestaviti resničnostno tabelo, v kateri razstavimo funkcijo, ki prejme vrednost multiplekserja, torej na katerem koraku smo in vrednost ukaza iz ukaznega registra.



Slika 2.10: Prikaz vloge EEPROM-a, kot logične funkcije

Na zgornji sliki (slika 2.10) logična funkcija f predstavlja EEPROM, ki je, kot smo omenili, nadomestilo za logično vezje, zaradi enostavnosti. Sledeča resničnostna tabela definira to funkcijo in stanja EEPROM-a za fetch cikel;

cikel	*ukaz*	S_n	*kontrolna beseda*
fetch	xxxx	$S_0(0000)$	0 1 1 1 1 1 1 0 1 0 1 0 0 1 1 1 1
	xxxx	$S_1(0001)$	0 1 1 1 1 1 1 0 1 1 1 1 1 0 1 0 1

Slika 2.11: Definicija kontrolne besede za fetch cikel

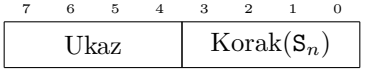
Seveda fetch cikel ni edini del ukaznega cikla, ki mora biti definiran, sedaj moramo definirati tudi decode in execute cikel, ki se bosta izvedla. Te dva cikla bosta v nekem smislu, malce združena, saj ne obstaja posebna komponenta dekodirne logike, ki bi ukaz dekodirala, kot v večini procesnih enot, temveč je ta logika že del EEPROM-a. Sledeča tabela je definicija kontrolne besede za vse obstoječe ukaze v zbirnem jeziku iz poglavja "2.2 Lasten zbirni jezik";

cikel	*ukaz*	S_n	*kontrolna beseda*
lda			
decode execute	0000	$S_2(0010)$	0 1 1 1 1 1 1 0 1 1 1 0 0 1 1 1 0
	0000	$S_3(0011)$	0 0 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1
ldb			
decode execute	0001	$S_2(0010)$	0 1 1 1 1 1 1 0 1 1 1 0 0 1 1 1 0
	0001	$S_3(0011)$	0 1 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1
ldra			
decode execute	0010	$S_2(0010)$	0 1 1 1 1 1 1 0 1 1 1 0 0 1 1 1 0
	0010	$S_3(0011)$	0 1 0 1 1 1 1 0 1 1 1 0 1 1 0 1 1
ldrb			
decode execute	0011	$S_2(0010)$	0 1 1 1 1 1 1 0 1 1 1 0 0 1 1 1 0
	0011	$S_3(0011)$	0 1 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1
outa			
decode execute	0100	$S_2(0010)$	0 1 0 1 1 0 1 0 1 1 1 0 1 1 1 1 1
outb			
decode execute	0101	$S_2(0010)$	0 1 1 1 0 0 1 0 1 1 1 0 1 1 1 1 1
add			
decode execute	0100	$S_2(0010)$	0 1 1 1 1 0 0 0 1 1 1 0 1 1 1 1 1
sub			
decode execute	0101	$S_2(0010)$	0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1
	0101	$S_3(0011)$	0 1 1 1 1 0 0 0 1 1 1 0 1 1 1 1 1

cikel	*ukaz*	$\mathtt{S}_n$	*kontrolna beseda*
addr			
decode execute	0110	$\mathtt{S}_2(0010)$	0 1 1 1 1 1 1 0 1 1 1 0 0 1 1 1 0
	0110	$\mathtt{S}_3(0011)$	0 1 1 1 1 1 0 0 1 1 1 0 1 1 0 1 1
subr			
decode execute	0111	$\mathtt{S}_2(0010)$	0 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 0
	0111	$\mathtt{S}_3(0011)$	0 1 1 1 1 1 0 0 1 1 1 0 1 1 0 1 1
hlt			
decode execute	1000	$\mathtt{S}_2(0010)$	1 x x x x x x x x x x x x x x x x

Slika 2.12: Definicija mikrokode vseh ukazov

Naslovne linije v EEPROM-u bodo sestavljene na sledeč način:



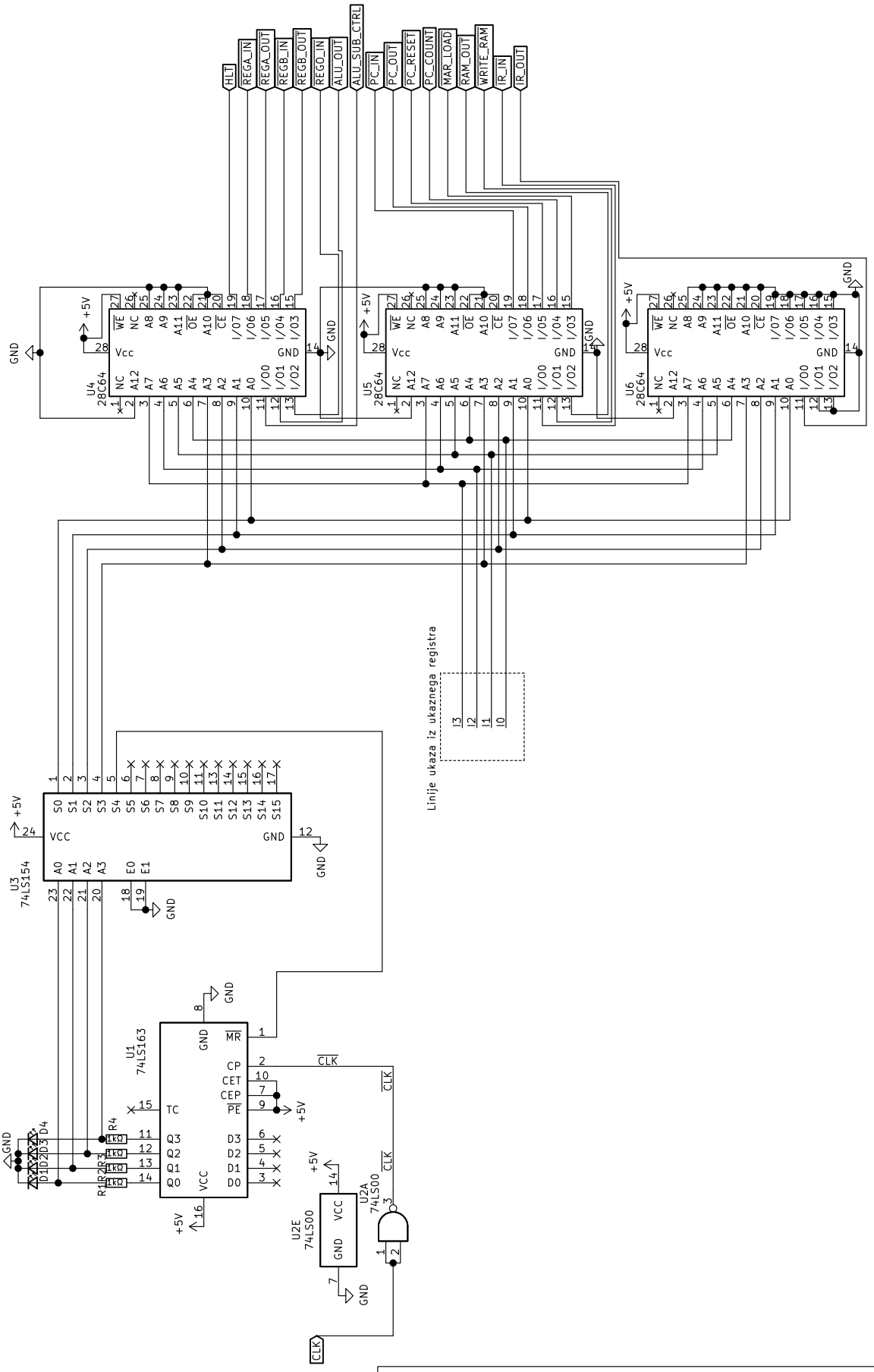
Slika 2.13: Naslovni bajt kontrolne logike

Kjer ukaz določijo linije iz ukaznega registra (*1.1.3.8 Modul ukaznega registra*), korak pa demultiplekser in kontrolni števec. Tako bo vedel, katero kontrolno besedo proizvesti. Njegovi izhodi, je dejansko 17-bitna kontrolna beseda, s najvišjim bitom `HLT` signalom in najnižjim $\overline{\mathtt{IR}_o}$, kakor opisuje diagram kontrolne besede na sliki 2.2.

Shema je vezje je sestavljeno iz treh vzporedno vezanih EEPROM-ov, saj en sam nima dovolj izhodnih signalov, da bi sam sestavil kontrolno besedo, saj ima samo 8-bitni izhod, kar bi pokrilo le 8-bitov 17-bitne besede.

Opomba (Vzporednost EEPROM-ov). EEPROM-e naslavljamo s istimi naslovi, to zagotovi njihovo vzporednost v vezju. Zgolj podatki na teh naslovih so drugačni oz. prirejeni temu kateri del kontrolne besede mora določiti. Skupaj imajo EEPROM-i 24-bitni izhod, od katerih porabimo le 17-bitov.

Shema vezja:



Kontrolna logika je sposobna upravljanja programiranega računalnika

Vanja Stojanović, Fakulteta za Računalništvo in Informatiko

Sheet: /		
File: Kontrolna logika.sch		
Title: Kontrolna logika		
Size: A4	Date: 2021-11-24	**Rev: 1**
KiCad E.D.A. kicad (5.1.8)-1		Id: 1/1

2.2 Lasten zbirni jezik

2.2.1 Teoretična podlaga in omejitve

Ustvarjanje zbirnega jezika je prost koncept, ki ga omejuje le arhitektura za katero ga pišemo, ostalo je prosto izbirno po naši volji. Zbirni jezik bom napisal po naslednjem konceptu:

Zbirni jezik bo imel nabor ukazov, ki bodo 4-bitni, to pomeni, da je ukazov lahko teoretično 2^4 ali 16, kjer ima vsak ukaz svojo ustrezno kodo. Primer;

lda

Ukaz lda bo v splošni register "a" naložil neko vsebino (ta bo uporabljena, kot operand ukaza). Ukaz lda bom zakodiral, kot 0000. Naslednji sestavni del ukaza so operandi, katere bom zapisoval na sledeči način, primer:

lda 14

Zgornji ukaz v splošni register "a" naloži podatke **s pomnilniškega mesta** 14, saj tako lahko v registre zapisujemo 8-bitne vrednosti, kajti ukazni register ima za operande rezervirane le 4 bite, isto, kot za ukaze. V praksi nas to omejuje, da lahko podatke jemljemo le iz pomnilniških mest 0000 - 1111.

2.2.2 Nabor ukazov

Zaradi manjšega števila mogočih ukazov, imamo CISC arhitekturo (*angl.* Complex Instruction Set Computer), kar pomeni, da posamezen se ukaz izvaja celoten CPU cikel *fetch, decode, execute, store*. Za nekakšno uporabnost računalnika, potrebujemo takšen nabor ukazov, da bomo lahko izvajali osnovne preproste operacije, tako aritmetične, kot ostale. Vendar teoretično imamo prosto izbiro nad tem katere ukaze bomo dodali in katere ne.

Prvo bomo spisali nabor ukazov, ki **upravljajo nalaganje oz. prenos podatkov** v razna pomnilniška mesta, *t. j.* v registre ali pa v glavni pomnilnik.

Ukazi	**4-bitno kodiranje**	**Opis ukaza**
lda `[MEMVAR]`	0000	Naloži vsebino s pomnilniškega naslova `MEMVAR` v register"a"
ldb `[MEMVAR]`	0001	Naloži vsebino s pomnilniškega naslova `MEMVAR` v register"b"
ldra `[MEMVAR]`	0010	Naloži vsebino registra "a" v pomnilnik na naslov `MEMVAR`
ldrb `[MEMVAR]`	0011	Naloži vsebino registra "b" v pomnilnik na naslov `MEMVAR`
outa	0100	Naloži vsebino registra "a" v vsebino izhodnega registra, kjer se ta izpiše na izhod
outb	0101	Naloži vsebino registra "b" v vsebino izhodnega registra, kjer se ta izpiše na izhod

Slika 2.14: Nabor ukazov za upravljanje s nalaganjem

Sedaj, ko lahko podatke prosto prestavljamo po splošnih registrih in glavnem pomnilniku, potrebujemo še nek nabor ukazov, ki nam dopušča izvajanje **pomenskih operacij** nad temi podatki, te imenujemo aritmetično-logične operacije.

Ukazi	4-bitno kodiranje	Opis ukaza
add	0100	ALU pasivno sešteje vsebini registra "a" in "b" in rezultat postavi v register izhodne enote za prikaz
sub	0101	ALU pasivno odšteje vsebino registra "a" s vsebino registra "b" in rezultat postavi v register izhodne enote za prikaz
addr `[MEMVAR]`	0110	ALU pasivno sešteje vsebini registra "a" in "b" in rezultat shrani na pomnilniško mesto **MEMVAR**
subr `[MEMVAR]`	0111	ALU pasivno odšteje vsebino registra "a" s vsebino registra "b" in rezultat postavi na pomnilniško mesto **MEMVAR**
hlt	1000	Postavi kontrolno besedo v pasivno stanje računalnika, ter MSB bit HLT signala postavi na 1, tako se računalnik po koncu programa ustavi

Slika 2.15: Nabor ukazov za osnovne aritmetične operacje

2.2.2.1 Psevdo ukazi in oznake

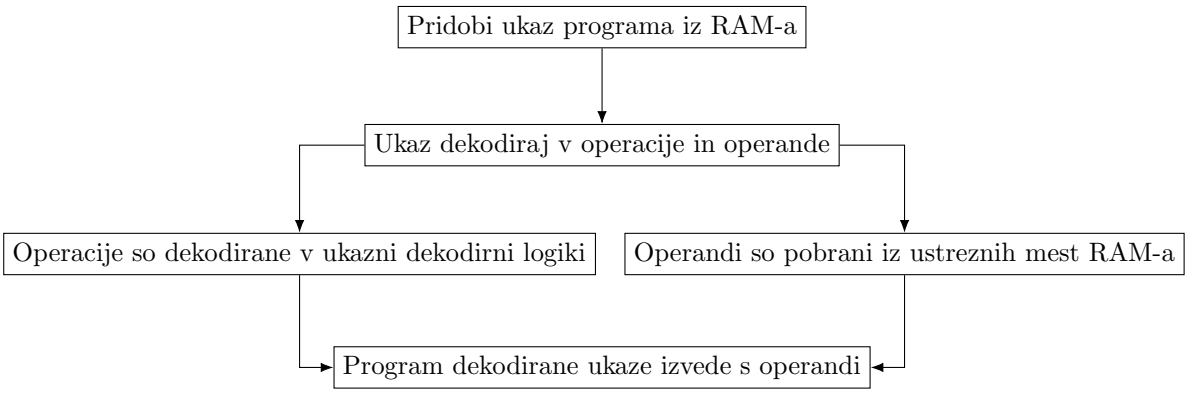
Psevdo ukazi in oznake, so posebni del kode, ki nam (*programerju*) povedo kaj se v programu dogaja in, kot ime pove, označijo razne dele kode, operande, operacije. Uporabljajo se zaradi berljivosti napisane kode, saj je ta tako preprosta, da lahko izgubimo sled programa, na dejanski program pa nimajo vpliva, saj se ne izvedejo.

Psevdo ukazi	Opis psevdo ukaza
`@`	Povzeto po zbirnem jeziku za CPU ARMx86, je to oznaka za komentar v kodi
`_start_program_:`	Posebna oznaka, ki nam pove kje se začne naš program, čigar prvi ukaz je zapisan na naslov 0x00 v pomnilniku.
`_end_program_:`	Posebna oznaka, ki nam pove kje se naš program konča, dolžina našega programa je omejena na 256 naslovov oz. ukazov, zaradi velikosti programskega števca.

Slika 2.16: Nabor psevdo ukazov in oznak

Sicer računalnik sam ni **sposoben razumeti oz. vanj ni mogoče vprogramiranti teh psevdo ukazov in oznak**, saj jih tudi ne podpira s svojo arhitekturo in operacijsko kodo, so uporabni za programerje in berljivost kode oz. neko formalizacijo tega zbirnega jezika.

Računalniško dekodiranje teh ukazov pa poteka na sledeč način. Kontrolna logika s pomočjo programskega števca pridobi vsebino registra na naslovu, ki je enak vrednosti programskega števca. Računalnik to vrednost prenese v MAR modul, kjer naslov dekodira in naslovi v RAM-u. Iz RAM-a to vrednost potem prenese v ukazni register, kjer ukaz dekodira na kodo operacije in kodo operanda. Kodo operacije oz. dejanskega ukaza pošlje v **dekodirno logiko ukazov**, kjer jo logika dekodira in s pomočjo kontrolne logike dejansko operacijo izvede. Koda operandov, pa v našem primeru pomeni **pomnilniško mesto na katerem se nahaja zaželeni operand oz. število**. Diagram poteka je prikazan na sliki 2.17. To je (*delni*) *fetch* in *decode* cikel CPU cikla.



Slika 2.17: Diagram poteka dekodiranja ukaza

V bolj kompleksnih sistemih, pri tem procesu sodeluje več komponent, ponavadi še kakšen dodaten register, ali del kontrolne logike. Razlog so bolj kompleksne arhitekture CPE-ja in tudi večji nabori ukazov, ter optimizacija tega procesa (*t. j.* da se proces dekodiranja izvede čim hitreje). A ker gre tu za dokazovanje nekega teoretičnega sistema, je optimizacija nekoliko zanemarljiva.

2.3 Teoretičen program in programiranje računalnika

2.3.1 Programiranje računalnika

Napisan program, je potrebno lastnoročno prevesti v strojno kodo in operacijsko kodo, s pomočjo nabora ukazov, kjer je zapisano njihovo kodiranje.

Program se ročno preko MAR in MDD modula vpiše na pomnilnik, ta se vedno začne na **pomnilniškem mestu 0x00**. Programiranje nam omogočajo tipke, na katere so vezani kontrolni signali za tako rečeno "*lastnoročni način*", v katerem je računalnik v pasiven stanju in ne izvaja ničesar, takrat mora biti tudi `HLT` signal v **logično visokem nivoju**, da bi preprečili kakršnokoli nezaželeno delovanje.

Ko program bit za bitom vpišemo v računalnik je programiranje zaključeno in računalnik lahko začne program izvajati.

2.3.2 Pravila za pisanje programa

Pravila za pisanje programa, so definirana s namenom, da bi preprečila nezaželeno delovanje računalnika, kot so neskončno izvajanje programa ipd. Pravil je malo a so sledeča:

- Program se začne s psevdo ukazov `_start_program_:` in konča s psevdo ukazov `_end_program_:`, med njima so lahko poljubni ukazi in komentarji.
- Vsak program je lahko dolg do 2^8 ukazov, zaradi velikosti programskega števca, vendar je potrebno paziti, saj ukazi lahko operande jemljejo in shranjujejo samo na pomnilniška mesta od 0x00 do 0x0F.
- Vsak program se mora končati s `HLT` ukazom.
- Pri naslavljanju operandov iz pomnilniškega prostora 0x00 - 0x0F, ne sme priti do prepisovanja ukazov, saj tako program postane napačen in se izgubi minimalno en ukaz. Obnašanje računalnika postane nedefinirano.

2.3.3 Teoritični programi

Sedaj bomo napisali teoretičen program, na podlagi zbirnega jezika, definiranega v prejšnjem podpoglavju 2.2 Lasten zbirni jezik. Začnimo s osnovnimi primeri.

Program, ki **pridobi vrednost pomnilniškega prostora 4** in jo shrani v register a.

```
_start_program_:
            lda 4
            hlt
_end_program_:
```

Program, ki **pridobi vrednost pomnilniškega prostora 4**, jo shrani v register b, in jo nato prestavi v register a.

```
_start_program_:
            ldb 4
            ldrb 5
            lda 5
            hlt
_end_program_:
```

Program, ki naloži vrednost pomnilniškega prostora 0x2 v register a, v register b postavi vrednost pomnilniškega prostora 0x1, opravi odštevalno operacijo med registroma a in b, ter rezultat shrani v register a in ga premakne v izhodni register, kjer ga tudi izpiše.

```
_start_program_:
            lda 2
            ldb 1
            subr 3
            lda 3
            outa
            hlt
_end_program_:
```

2.4 Razcep programa na mikrokodo

Oglejmo si popolno delovanje računalnika in njegovo mikrokodo na primeru prejšnjega primera programa:

```
_start_program_:
            lda 1
            ldb 2
            subr 3
            lda 3
            outa
            hlt
_end_program_:
```

2.4.1 Fetch cikel

Program se začne na pomnilniškem mestu 0x0, v katerem je vpisan prvi ukaz `lda` 1. Programski števec je na vrednosti $0x0_{(16)}$ oz. $00000000_{(2)}$ torej se naš ukaz, ki ga moramo izvesti nahaja na pomnilniški lokaciji 0x0, **kjer se tudi začne program**. Mikrokoda *fetch* cikla je torej definirana kot (za referenco kontrolnih signalov in definicije fetch cikla poglejte poglavje 2.1 Kontrolna beseda in logika):

	fetch						
S_0	$\overline{PC_o}$	$\overline{MAR_l}$					
S_1	$\overline{RAM_o}$	$\overline{IR_i}$	PC_c	PC_o	MAR_l		
				RAM_o	IR_i	$\overline{PC_c}$	

Slika 2.18: Mikrokoda fetch cikla

Opomba. Ostali obstoječi kontrolni signali so v nevtralnem stanju (to so ustrezno visoki ali nizki nivo glede na vhod, ki ga kontrolirajo).

2.4.2 Decode in execute cikel

Ko v ukazni register prejmemo ukaz iz poljubnega pomnilniškega naslova, se ta po zasnovi samega modula, loči na dva dela, **operacijo** in **operand**. Operacija se pasivno prenese v dekodirno logiko, kjer ta dekodira ukaz, ter izvede potrebne kontrolne operacije in mikrokodo za izvedbo originalnega ukaza. Operand pa ostane v ukaznem registru, kjer ga nato prenesemo v MAR register, za naslavljanje pomnilniškega prostora, če je to potrebno.

Prvi "**lda** 2" ukaz se prevede v sledečo mikrokodo;

	decode & execute				
S_0	$\overline{IR_o}$	$\overline{MAR_l}$			
S_1	$\overline{RAM_o}$	$\overline{RA_i}$	IR_o	MAR_l	
			RAM_o	RA_i	

Slika 2.19: Mikrokoda "**lda** 2" ukaza

Pri čemer se vrednost 0x01 prenese v MAR register in RAM je posledično naslovljen z njo. Podatki na tem naslovu, se nato prenesejo v splošni register "a" za nadaljnjo uporabo. Da bi izvedli naslednji ukaz se ponovno izvede tudi fetch cikel (*slika* 2.18).

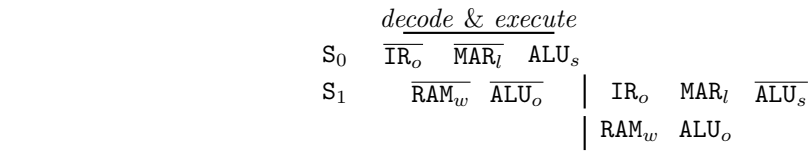
Drugi "**ldb** 1" ukaz se prevede v sledečo mikrokodo;

	decode & execute				
S_0	$\overline{IR_o}$	$\overline{MAR_l}$			
S_1	$\overline{RAM_o}$	$\overline{RB_i}$	IR_o	MAR_l	
			RAM_o	RB_i	

Slika 2.20: Mikrokoda "**ldb** 1" ukaza

Verjetno ni očitno, vendar v register "a" se je naložila vrednost same operacijske kode ukaza **ldb** 2, kar je v binarnem $00010010_{(2)}$, prav tako se je v register "b" naložila vrednost ukaza **lda** 1, kar pa je $00000001_{(2)}$. Na kar se ponovno ponovi fetch cikel in pridobi;

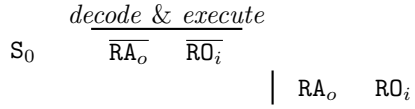
Tretji ukaz "**subr** 3" se prevede v sledečo mikrokodo:



Slika 2.21: Mikrokoda "**subr** 3" ukaza

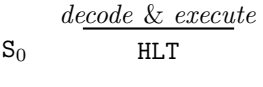
Ukaz se izvede tako, da prvo dostopa do pomnilniškega naslova 0x03, saj je ta definiran na tak način, da izvede operacijo odštevanja in nato rezultat shrani na omenjen naslov. S tem ukazom, smo na pomnilniškem naslovi 0x03 efektivno povozili kakršenkoli prej programiran ukaz, kar dejansko ne vpliva na program, saj je bil ta že izvršen. S fetch ciklom ponovno pridobimo naslednji ukaz programa;

Četrti ukaz "**lda** 3" se prevede v isto mikrokodo, kot prej, na sliki 2.19. Ta naloži vsebino RAM-a na naslovu 0x03, kar je dejansko rezultat prejšnje operacije odštevanja. Temu sledi peti ukaz "**outa**", ki se prevede v sledečo mikrokodo;



Slika 2.22: Mikrokoda "**outa**" ukaza

Tu efektivno le kopiramo vrednost iz registra "a" v izhodni register, kjer je "zaklenjena" in se izpisuje na 7-segmentne prikazovalnike. Kar nas pripelje na šesti in zadnji ukaz v programu, **hlt**;



Slika 2.23: Mikrokoda "**hlt**" ukaza

Tu se naš program ustavi, ne le program, temveč tudi celoten računalnik in se postavi v pasivno stanje, pripravljen na ponovno programiranje. Pasivno stanje računalnika je definirano, kot stanje v katerem se **program ne bere iz pomnilnika** in hkrati mora biti **urin cikel efektivno ustavljen**.

Opomba (Store cycle). Cikel shranjevanja, je namenoma izpuščen, saj je večina ukazov definirana z nekim shranjevanjem že v mikrokodi. To pomeni, da lahko uporabimo skrajšan CPU cikel definiran v poglavju *1.1.1.1 Osvnovna izvedba programa.*

3 Zaključek in ugotovitve

V osnovi s današnjo tehnologijo in znanjem takšen projekt ni težko narediti, a je bil pravi izziv znanstvenikom, pred 50 leti, ki so v take vode prvič stopali sami, brez predhodnega znanja. Računalnik pa zdaleč ni popoln, ima kar veliko napak in optimizacijskih priložnosti, ki bi se lahko izkoristile in računalnik izboljšale. Vendar ta ni nikoli bil namenjen generalni uporabi ali pa potrošniški proizvodnji, temveč le za praktično oz. teoretično prikazovanje osnovnih funkcij, gradnikov CPE-ja in Von Neumannovega računalniškega modela.

Računalnik ima vse osnovne komponente, ki bi jih potreboval za delovanje, vse od splošnih registrov, do specializiranih registrov, kot so programski števec, ukazni register, pomnilniški registri in izhodni register pa do drugih enot, kot so pomnilnik, izhodni modul, dekodirna in kontrolna logika, ter urin pulz, ki sinhronizira celoten CPE. Ker je računalnik zgrajen po Von Neumannovem modelu, za svoje delovanje predvideva program, ki mu pove kaj narediti. Ta program je napisan v jeziku, ki je zelo blizu strojne kode *t. j.* zbirnem jeziku, ki smo ga sami definirali in tudi kodirali, da je lahko shranjen v pomnilniku. Vendar sam program iz 1 in 0 ne pride prav, če ga računalnik ne razume, zato poskrbi kontrolna logika, ki po dekodiranju ukaza, s pomočjo naše definirane mikrokode proizvede tako kontrolno besedo, da orkestrira vse komponente računalnika k opravljanju iste pomenske operacije.

Seveda sem omenil nekaj priložnosti na področju izboljševanja računalnika, ena izmed bolj pomembnih pa je organizacija pomnilnika. Program bi se lahko začel na mestu 0x16 namesto 0x00, in programski števec bi zmeraj štel od 0x16 naprej. To bi bila enostavna izboljšava, saj po definiciji formata ukazov, lahko do operandov v pomnilniku dostopamo le od naslova 0x00 do 0x15, tako tega mesta ne bi zasedali ukazi. Obstaja tudi možnost posodobitve ALU modula, s raznimi izvedbami, kot zastavica prenosa ali zastavica preliva, hkrati pa tudi druge matematične operacije, množenje in deljenje. S programskim števcem bi lahko omogočili tudi skoke v programu ali pa zanke, lahko bi vpeljali tudi pogojne izvršitve s kontrolno logiko.

Vendar konec koncev je to le teoretičen koncept namenjen izobraževanju in ponazoritvi izdelave takšnega elektronskega sistema za razumevanje osnove računalništva.

Viri literature

1. STOJANOVIĆ, Vanja. 2021. *8-bitna Računalniška Enota* (Maturitetno delo). Ljubljana.
2. STARIČ, Marko. 2005. *Elektronika v fiziki : za študente Fizikalne merilne tehnike*. Ljubljana: Univerza v Ljubljani, Fakulteta za Matematiko in Fiziko.

Viri slik

- 1.5 FARICHILD SEMICONDUCTOR, μA555 SINGLE TIMING CIRCUIT, FAIRCHILD LINEAR INTEGRATED CIRCUIT
- 1.6 MOTOROLA, octal D Flip-Flop with ENABLE
- 1.7 MOTOROLA, octal D Flip-Flop with ENABLE
- 1.9 FARICHILD SEMICONDUCTOR, 3-state Octal Bus Transceiver, Marec 2000
- 1.10 PHILIPS SEMICONDUCTORS, 4-bit binary full adder with fast carry, December 1990
- 1.12 ATMEL, Parallel EEPROM with Page Write and Software Data Protection, 1999
- 1.14 PHILIPS SEMICONDUCTORS, presettable synchronous 4-bit binary counter; synchronous reset, December 1990
- 1.17 FAIRCHILD SEMICONDUCTOR, 64-BIT RANDOM ACCESS MEMORY with 3-state output
- 2.7 FAIRCHILD SEMICONDUCTOR, 4-Line to 16-Line Decoder/Demultiplexer, Marec 2000