VEGOVA
ELEKTROTEHNIŠKO-RAČUNALNIŠKA
STROKOVNA ŠOLA IN GIMNAZIJA
LJUBLJANA

Raziskovanje mej RISC-a

(računalništvo in informatika)

Raziskovalna naloga

Avtor: Adrian Sebastian Šiška
Mentor: Aleš Volčini

Ljubljana, marec 2022

Kazalo

Slike

1 Zahvala

Zahvaljujem se vsem, ki so mi pomagali pri raziskovanju in pisanju. Še posebej se zahvaljujem mentorju Alešu Volčiniju za podporo in potrpežljivost. Zahvalil bi se tudi mojima prijateljema Oliverju Wagnerju (`oliwerix.com`) in Antonu L. Šijancu (`sijanec.eu`), ki sta s svojimi implementacijami emulatorja MUHI popestrila moje delo.

2 Povzetek

V raziskovalni nalogi sem si zamislil novo RISC arhitekturo, imenovano MUHI (*Minimal URISC hardware implementation*). MUHI je enobitna, dvo-instrukcijska arhitektura. Za njeno implementacijo sem raziskoval področje teorije izračunljivosti in arhitekture ostalih procesorjev. Za testiranje delovanja sem sprogramiral emulator, za lažje pisanje programov pa sem pripravil tudi zbirnik in razbirnik. Kot pomoč pri iskanju sestavljenih operacij sem napisal še program za iskanje kombinacij ukazov. Implementacija je Turing-kompletna, kar sem dokazal tako, da sem napisal celični avtomat, ki izvaja pravilo 110.

Ključne besede: RISC, URISC, arhitektura procesorja, Univerzalen procesor, navidezna naprava, emulacija

3 Uvod

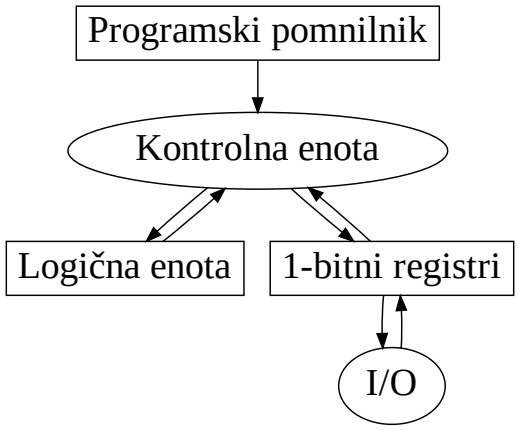
Procesorji so najpomembnejša komponenta v moderni zgradbi računalnika, kljub njihovi pomembnosti pa jih ljudje dandanes čedalje slabše razumejo. Oblikovanje procesorjev smo prepustili relativno majhnim skupinam razvijalcev, z nizkonivojskimi jeziki pa se le še malokdo ukvarja. To me čudi, saj vse od kar sem slišal za Intelov *management engine* ne morem nehati razmišljati o mojem procesorju. Spraševati sem se začel, kako lahko zares preverim, da management engine ni aktiven v mojem računalniku in ali lahko sploh zaupam svojemu procesorju. To me je nekako pripeljalo do ideje, da naredim svoj procesor. Tako sem si zamislil MUHI arhitekturo.

3.1 Hipoteza

V tej nalogi bom preveril resničnost sledeče hipoteze:

- Moja lahkotna arhitektura imenovana MUHI je Turing-kompletna.

4 Moja arhitektura



Slika 1: Shematika MUHI-ja

Imenuje se *Minimal URISC hardware implementation* oz. kratica MUHI.

4.1 Kaj je URISC?

Kratica URISC pomeni *ultimate reduced instruction set computer*, kar bi lahko v slovenščino prevedli kot *računalnik z minimalno množico operacij*. RISC arhitekture so v moderni praksi, boljše od CISC arhitektur, vsaj odkar pomnilnik ni več drag. To nam dokazujejo moderni pametni telefoni, saj skoraj vsi delujejo na procesorjih vrste ARM. To nam dokaže tudi najpopularnejši ustvarjalec CISC procesorjev, Intel, saj so dandanes njihovi procesorji interno

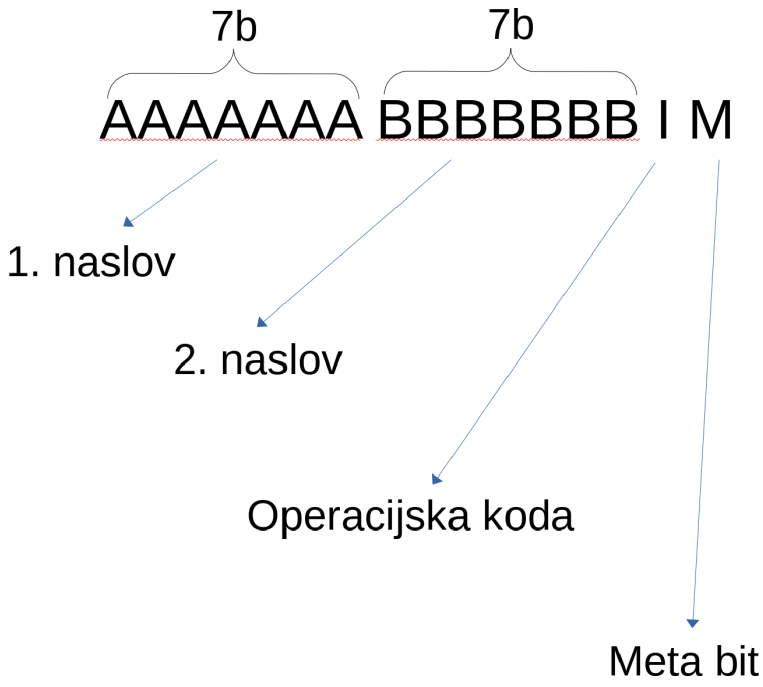
zgrajeni iz tolmača in manjšega RISC procesorja. Da arhitekturo označimo kot URISC, pa moramo iti še korak dlje in se znebiti vsek ukazov razen enega. Zanimiva posledica tega je dejstvo, da lahko, ko pišemo strojno kodo, popolnoma spustimo polje z instrukcijo. Tako ostanejo le še njeni argumenti.

V virih najdemo primere ukazov, ki so sami po sebi Turing-kompletni, npr. x86 mov ukaz[1]. Jaz sem se raje odločil za 2 instrukciji, ker se njuni rabi ne prekrivata zelo dosti, saj med drugim tudo operirata na različnih magnitudah količine podatkov. Posledica dodatne instrukcije je lažje pisanje programov. Ker bi bilo opisovanje programiranja MUHI preveč obsežno, sem se odločil, da se bom v tej nalogi posvetil arhitekturi sami.

4.2 Instrukcije

Obe instrukciji imata 3 argumente. Prva dva argumenta sta registra, tretji pa je meta bit. V moji izvedbi sem se odločil za 7-bitne[2] pomnilniške naslove. O organizaciji pomnilnika bom več povedal kasneje.

Strojna koda ukaza je sestavljena iz 1. naslova, 2. naslova, operacijske kode instrukcije in t. i. meta bita.



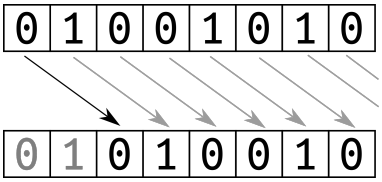
Slika 2: Prikaz sestave strojne kode

Meta bit je ostal zaradi poravnave dolžine ukaza na 2 bajta, s čimer je naredilo pisanje zbirnika in razbirnika nekoliko lažje. Obe instrukciji se obnašata tako, da vzameta podatke iz 1. in 2. naslova ter rezultat shranita nazaj na 2. naslov.

[1]Stephen Dolan, mov is Turing-complete

[2]Ta izbira je arbitrarna, arhitektura podpira n bitne naslove. 7-bitne sem izbral, ker je posledično dolžina ukaza v bitih strojne kode potenca števila 2.

4.2.1 Instrukcija "c16"



Slika 3: Grafični prikaz c16 instrukcije

To je 1. instrukcija, njena operacijska koda je 0. Njena notacija izhaja iz daljšega imena *copy 16 bits* Deluje tako, da od 1. podanega naslova 16 zaporednih registrov (*oz. bitov*) prekopira v začasen pomnilnik, potem pa jih prekopira na 2. naslov. Odločitev, da bo to 1. instrukcija, izhaja iz tega, da so same ničle v programu interpretirane kot ukaz brez posledic, kar je uporabno predvsem takrat, ko je potrebno z ničlami rabimo kaj poravnati na naslov, ki je nekaj mest nižje. Na začetku je bila zasnovana skoraj kot jump instrukcija, saj z NANDom ni mogoče vseh bitov v programskem števcu hkrati spremeniti na željeno vrednost. Posledica spremembe enega pa po enega bita v programskem števcu, bi bil skok takoj po 1. spremembi enega bita, saj kontrolna enota vedno preveri števec pred nalaganjem nove instrukcije.

Ko sem kasneje dodal še meta bit, sem se odločil, da se bo kopiranje ob nastavljenem meta bitu zgodilo iz programskega spomina. S tem je nalaganje konstant, kot so nizi ali pa naslovi funkcij, postalo zelo preprosto in hitro. Omejitev tega je, da sem uporabil 7 bitni naslov za pridobivanje podatkov iz programskega pomnilnika opisanega s 16 biti (uporabno je 128 bitov od 65536-ih). Zaradi tega večina pomnilnika še vedno ni na voljo, kot rešitev temu pa sem se odločil, da sem uporabil skrite registre kot nastavitev strani programskega pomnilnika (*memory page*). Tako dobimo 2^{15} strani programskega pomnilnika.

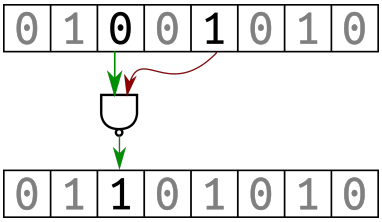
Skriti registri namreč nastanejo kot stranski učinek omejitve velikosti naslova na 7 bitov.

Kaj se na primer zgodi, če pri kopiranju uporabimo zadnji register? Ker instrukcija dostopa tudi do registrov nad 127, bi moral v tem primeru implementirati preliv (naslednji bi bil register 0) ali povsem ignorirati spreminjanje registrov od naslova 128 dalje. Jaz sem se odločil za tretjo možnost - ostranjevanje pomnilnika. Registre za zadnjim registrom sem najprej poimenoval *skriti registri*. Čeprav sem jih kasneje uporabil za ostranjevanje pomnilnika, njihovega poimenovanja še nisem zamenjal. Ti registri so torej dostopni oz. v uporabi le, kadar kopiramo registre od 113 do 127. Do njihove vsebine se da tako dostopati le s to instrukcijo, kar se mi ne zdi problem, saj se bodo strani pomnilnika menjavale le redko.

Torej se ukaz `c16 0x22, 0x44, 0` prevede v `0100010 1000100 0 0`, kar bi v programskem jeziku C tukaj lahko predstavili kot
`int *a=0x22; int *b=0x44; *b=*a;*(b+1)=*(a+1)...*(b+15)=*(a+15).`

Ker je instrukcija c16 edina instrukcija, ki je edina sposobna nastaviti programski števec na dano absolutno vrednost, sledi da mora biti tudi sposobna prekopirati vse bite programskega števca.

4.2.2 Instrukcija "Nox"



Slika 4: Grafični prikaz nox instrukcije

To je 2. instrukcija, njena operacijska koda je 1. Ime izhaja iz njenega kratkega opisa: *NAND or XOR*. Najprej sem poizkusil kot edini ukaz za manipulacijo bitov uporabiti le XOR. Vendar sem hitro ugotovil, da se samo z njo ne da rekonstuirati vseh ostalih logičnih vrat. Zato sem se odločil še za najpogosteje izbrana univerzalna vrata NAND.

Ta so s seboj privlekla nove probleme; npr. pogojni skoki naprej so postali zelo dragi. Programski števec bi sicer lahko bil viden samo instrukciji c16, vendar bi to preprečilo optimizacijo relativnih skokov na naslove, ki se od trenutnega razlikujejo v enem samem bitu (torej je odmik potenca števila 2). Pogojni relativni skoki so razširitev zgornje ideje. Ker vem, kje se nahaja trenuten ukaz, tudi vem, v kakšnem stanju je vsak bit programskega števca. S kombiniranjem bita z vrednostjo 1 v programskem števcu in nekega podatka v ramu lahko skočimo nazaj samo takrat, ko je tudi podatek v ramu enica. Pogojen skok naprej ni izvedljiv na tak način, saj bi na mestu bita v programskem števcu, ki ga želimo pogojno povečati, morala biti ničla. Operacija NAND v primeru, da, je katerikoli izmed argumentov 0, vedno da rezultat 1, kar pomeni, da naše *pogoj* iz pogojnega skoka ni bil upoštevan.

Zaradi dragih skokov in lahke ponastavitve celice na 0, sem se odločil, da dodam še XOR. Par vrat sem kombiniral v eno instrukcijo, kjer meta bit izbira med njima.

$$f(A,B,M) = \begin{cases} NAND(*A,*B) \to *B & M = 0 \\ XOR(*A,*B) \to *B & M = 1 \end{cases}$$

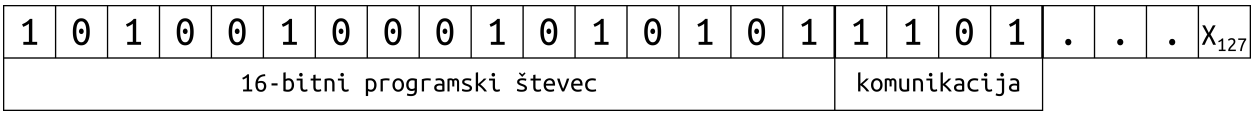
$*A$ pomeni priklic in zapis na register z naslovom A

Če zgornje izrazimo z logično notacijo, dobimo:

$$\overline{(AB)}(A + B + \overline{M})$$

Torej se ukaz `Nox 0x15, 0x53, 0` prevede v `0010101 1010011 1 0`, kar bi v programskem jeziku C tukaj lahko predstavili kot `int *a=0x15;int *b=0x53;*b=NAND(*a,*b)`.

4.3 Enobitni registri (*podatkovni pomnilnik*)



Slika 5: Grafični prikaz registrov

Ram sem si zamislil, kot *trak* enobitnih registrov. Zaradi preprostosti je programski števec preslikan čez prvih 16 registrov, da lahko z njim operira tudi instrukcija *Nox*. Za njim sledijo štirje biti vhodno-izhodnega (I/O) naslovnega prostora. Več o njih bom povedal kasneje. Število vseh registrov je funkcija dolžine naslova rama in velikosti programskega števca. Za računanje dolžine traka registrov sem pripravil preprosto formulo. $F(N, P)$, v mojem primeru $F(7, 16)$, izračuna število registrov, ki jih imam na razpolago, pri danem N in P.

$$F(N, P) = 2^N P - 1$$

$F(N, P)$ je število registrov, N je dolžina binarnega naslova, P je dolžina programskega števca

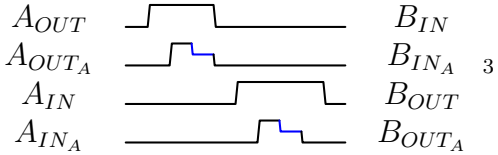
4.4 Programski pomnilnik

Programski pomnilnik je organiziran v šestnajst-bitne besede. Njegova velikost znaša 65536 besed oz. 131072 bytov. Vsebuje lahko instrukcije ali konstante. Programski števec vsebuje naslov instrukcije, ki se bo ob naslednjem ciklu izvedla. Edini način za branje iz njega je z c16 instrukcijo, za pisanje vanj pa trenutno še ni mehanizma.

4.5 Komunikacija z zunanjim svetom

Za izvedbo vhodno-izhodnih operacij se odločil, da bom implementirati svojo 4-žično komunikacijo. Ta lahko asinhrono prenaša informacije bit za bitom. Asinhronost je pomembna lastnost, saj arhitektura nima prostora za začasen pomnilnik, ki bi shranjeval podatke za čas, ko bi se program lahko začel z njimi ukvarjati.

Po drugi strani bi prekinitve (interrupts) dodale ogromno kompleksnosti, kot npr. najmanj 16 dodatnih registrov, instrukcijo za branje z njih, možnost spreminjanja registrov, ki ni posledica nekega ukaza itd. Lepa lastnost štirižičnega protokola je tudi, da se dve napravi z lahkoto skupaj poveže, saj moramo le izhode iz ene priklopti v vhode druge.



Primer pošiljanja enice od A do B in od B do A

[3] modra črta na sredini prikaže stanje, kjer ena naprava drži signal pri logični enici, druga naprava pa ga poizkuša nastaviti na ničlo

Za pošiljanje informacij najprej počakamo, da je OUT_A dovoljen, torej 0. Nato lahko pišemo, kar hočemo, v OUT, dokler ne želimo vrednosti *oddati*, kar naredimo s tako, da napišemo enico na OUT_A.

Za prejemanje informacij je postopek podoben. Začnemo s preverjanjem IN_A, kot da bi gledali nabiralnik, če nam je poštar kaj prinesel. Čim dobimo napisano enico, smemo interpretirati karkoli je trenutno v IN, kot podatke. Da dobimo naslednji znak, priklopimo pulldown upor na IN_A. To druga naprava zazna in čim prej spusti IN_A na 0.

5 Izdelava

Najprej sem se odločil da bom potreboval okolje, kjer bi lahko testiral arhitekturo, spremembe in moje programe. Zato sem se lotil pisanja virtualnega stroja. Ker je brez preizkusov težko vedeti, ali stroj pravilno deluje, sem začel delo na zbirniku, da mi ne bi bilo več treba na roke pisati enic in ničel. Seveda je izdelava zbirnika tudi težka, zato sem razvil še razbirnik, s katerim preverim delovanje zbirnika. Izdelava le tega je bila nekoliko lažja, kljub temu pa sem našel večjo napako pri dekodiranju ukazov šele po nekaj dneh.

Virtualni stroj, zbirnik in razbirnik sem napisal v programskem jeziku python.

5.1 Zbirnik (assembler)

Glej prilogo *asm.py*.

```
adrian@laptop:~/1bitvm$ ./disasm.py example2.out
======example2.out======
0x0000    c 0x01, 0x00, 1     0x0201
0x0001    a 0x00, 0x00, 0     0x0002
0x0002 nop
...
0x0008    a 0x21, 0x21, 1     0x4287
0x0009    a 0x11, 0x21, 1     0x2287
0x000a    a 0x21, 0x21, 0     0x4286
0x000b    a 0x21, 0x0d, 0     0x4236
0x000c    a 0x20, 0x20, 1     0x4083
0x000d    a 0x10, 0x20, 1     0x2083
0x000e nop
...
0x0010    a 0x21, 0x21, 1     0x4287
0x0011    a 0x13, 0x21, 1     0x2687
0x0012 nop
0x0013    a 0x21, 0x0d, 0     0x4236
0x0014    a 0x12, 0x12, 1     0x244b
0x0015    a 0x20, 0x12, 1     0x404b
0x0016    a 0x13, 0x13, 0     0x264e
0x0017    a 0x11, 0x11, 0     0x2246
0x0018    c 0x01, 0x00, 1     0x0201
adrian@laptop:~/1bitvm$
 733  *terminal*  24:24 All    Term : char run
```

Slika 6: Grafični prikaz nox instrukcije

Najprej celotno datoteko podamo NASMovemu preprocesorju.[4] Preprocessor skrbi za dodajanje večih datotek skupaj in razširjanje makrov ter definicij. Ker moramo zaradi oblike zbirnega jezika ohraniti vrstice iz izvorne kote, C-jev preprocesor ni primeren.

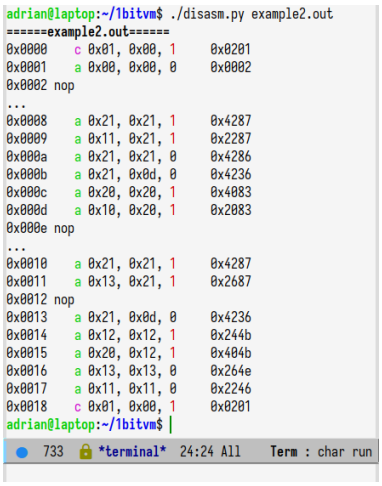
Potem za vsako vrstico v datoteki naredimo naslednje:

[4]NASM (*The Netwide Assembler*) je popularen x86 zbirnik, ki mi služi le kot preprocesor.

- preiščemo vrstico za par "< *py* >" simbolov, ki tvorita python blok. Vsaka uporabnikova konstanta je tako ali tako interpretirana kot izraz v pythonu, vendar nastane problem, ko želimo interpretirati izraze, ki vsebujejo vejice. Vejice so namreč ločilo med polji, zato moj preprost program nareže en pythonov izraz na več majhnih delov, ki sami po sebi niso več veljavni.
- Na koncu vse py bloke zamenjamo z njigovimi izračunanimi vrednostmi v vrstici.
- Nato klasificiramo vsako vrstico v eno izmed petih možnih vrst.
  - Komentar ali prazna vrstica, ki jo lahko varno spustimo,
  - Surovi binarni podatki, ki jih direktno zapišemo v datoteko,
  - Premik v programu: premaknemo mesto pisanja in nadaljujemo,
  - Instrukcija v procesorju, ki jo dekodiramo, izračunamo njene argumente in zapišemo,
  - Naslov, katerega pozicijo si zapomnimo, da bomo kasneje lahko omenjali predmet pred katerim stoji.

5.2 Razbirnik (Dissasembler)

Glej prilogo *disasm.py*.

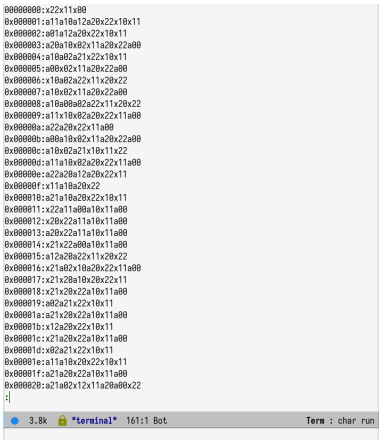


Slika 7: Primer uporabe razbirnika

Med izdelavo delujočega zbirnika sem naletel na nekaj problemov, zato sem z malo pomoči soseda Oliverja napisal še manjši program, ki se sprehodi čez datoteko in jo interpretira v instrukcije in njene argumente. Izpiše nam naslov, na katerem se ukaz nahaja, ukaz sam, meta bit in še šestnajstiški prikaz tega ukaza, saj ne loči med podatki in ukazi.

5.3 Iskanje ukazov s surovo silo

Glej prilogo *gate-bf*.



Slika 8: Prikaz tabele z vsemi možnimi rezultati

Pisanje makrov, kot so `add` je za človeka zaradi nepreglednosti težko opravilo. Seveda se da poiskati kombinacijo NAND-ov in XOR-ov, ki bi mi vrnila seštevek dveh številk, vendar ne morem biti prepričan, da sem res našel najkrajši način za to. Zato sem napisal program, ki najde najkrajše kombinacije ukazov namesto mene in jih zapiše v tabelo. V taki tabeli poiščemo mesto s številskim opisom našega problema in prepišemo zaporedje ukazov, ki ta problem reši. Ideja je v tem, da za vsak možen vhod v n celic preizkusimo vse možne ukaze, in gledamo, kdaj dobimo nov rezultat. Ko dobimo kaj novega, lahko preprosto ponovimo postopek. Tako lahko generiramo tabelo rezultatov glede na vse možne vhode.

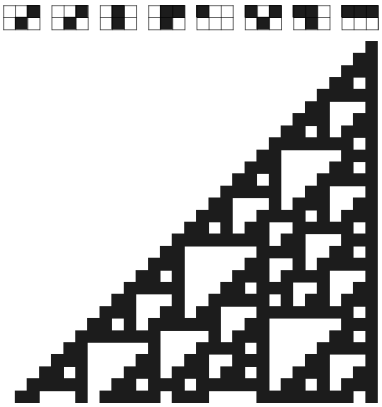
Prva implementacija tega programa je bila napisana v pythonu in bi po mojih ocenah trajalo nekaj tednov, da bi z njo dobil popolno tabelo vseh možnih rezultatov. Zato sem prepisal cel program v C in z uporabo več niti hkrati skrajšal čas iskanja na 3 sekunde.

6 Virtualna implementacija

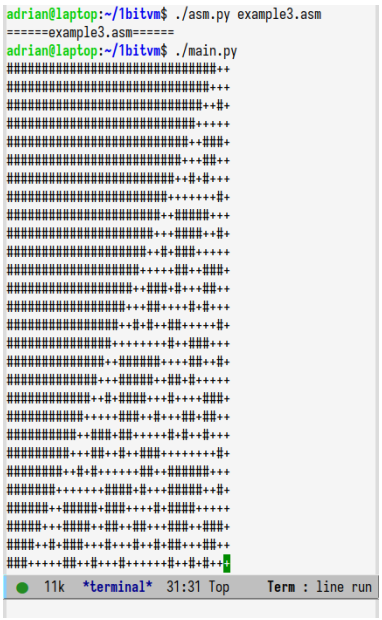
Glej prilogo *main.py*.

Da zares lahko pokažemo delovanje naprave, moramo nekako prikazati kako deluje. To na najlažji način danes dosežemo z implementacijo v virtualnem okolju. Zato sem se lotil pisanja virtualne naprave v pythonu. Virtualna naprava ima vse lastnosti prej opisane arhitekture. Ima implementirano konzolo, v razhroščevalnem načinu pa tudi vpogled v notranje stanje. Tako lahko preko terminala z navidezno napravo tudi komuniciramo. To lahko najlažje vidimo pri primeru št. 1 (priloga), kjer naprava izpiše “Zivjo”, ali v primeru št. 2, kjer nam naprava vrne nazaj vsak bit, ki ga prejme.

6.1 Celični avtomat



Slika 9: Pravilo 110



Slika 10: Prikaz delovanja celičnega avtomata na MUHI arhitekturi.

Za dokazovanje, da je arhitektura MUHI turing-kompletna, sem izbral emulacijo drugega turing-kompletnega sistema. Eden izmed najbolj preprostih, čeprav še vedno dovolj kompleksnih, da je univerzalen, je Celični avtomat s pravilom 110. [3] Seveda ne morem implementirati neskončno dolgega enodimenzionalnega traka, na katerem bi simuliral izvajanje tega pravila, lahko pa pokažem, da bi se to dalo implementirati, če bi moja naprava imela neskončno rama.

7 Zaključek

7.1 Hipoteza

- Moja lahkotna arhitektura imenovana MUHI je Turing complete. Hipoteza potrjena, saj lahko popolnoma emulira sistem, za katerega je že bilo dokazano, da je turing-kompleten.

8 Priloge

Vse priloge in izvorna koda so dosegljive na `https://github.com/adimineman/raziskovalna` in so licencirane z odprtokodno licenco.

Listing 1: asm.py

```
#!/usr/bin/env python3
"assembler for 1 bit processor"

import sys
import subprocess
import typing
import math

PAGE = 2**7
DEBUG = False

IN = 0x10
IN_A = 0x11
OU = 0x12
OU_A = 0x13


def write_safe(outfile: typing.IO, data: bytes) -> bool:
    "writes int to current position in file, also checks for possible overwriting"
    oli_pojntr = outfile.tell()
    assert len(data) == 2, "Something wong!"
    if int.from_bytes(outfile.read(2), "big") != 0:
        print(
            f"\033[33mOverwriting prevented on {outfile.tell()//2:#06x}\033[0m",
            file=sys.stderr,
        )
        return False
    outfile.seek(oli_pojntr)
    outfile.write(data)
    return True


def py_eval(line: str, loc: dict):
    "eval py regions in lines"
    if line.count("<py>") % 2 == 0 and line.count("<py>") > 1:
        # breakpoint()
        splitline = line.split("<py>")
        for i, py_macro in enumerate(splitline[1::2]):
            splitline[i * 2 + 1] = str(eval(py_macro, globals(), loc))
        return "".join(splitline)
    return line


def write_inst(outfile: typing.IO, line: str, loc: dict) -> bool:
    "interpret line and write as instruction"
    outfile.seek(math.ceil((outfile.tell() % 2) / 2), 1)
    tokens = line.split(" ")
    xor = 0
    if tokens[0] == "xor":
```

```
        com = 1
        xor = 1
    else:
        com = 1 if tokens[0] == "nand" else 0
    tokens = tuple(map(str.strip, "".join(tokens[1:]).split(",")))
    op1 = eval(tokens[0], globals(), loc)
    op2 = eval(tokens[1], globals(), loc)
    out = (com % 2) << 1
    if len(tokens) >= 3 and len(tokens[2]) > 0:
        out |= eval(tokens[2], globals(), loc) % 2
    if xor:
        out |= 1
    out |= (op1 % 128) << 9
    out |= (op2 % 128) << 2
    outb: bytes = out.to_bytes(2, "big")
    if not write_safe(outfile, outb):
        return False
    return True


def alignto(here: int, length: int, pattern: int):
    "align PC to next pattern"
    mask: int = 2**length - 1
    if here & mask <= pattern:
        return here & ~mask | pattern & mask
    return (((here >> length) + 1) << length) & ~mask | pattern & mask


def by2(arg: int) -> bytes:
    "helper function that casts ints(?) to 2 bytes"
    return arg.to_bytes(2, "big")


def cmp(filename: str) -> bool:
    "compile"
    nasm = subprocess.run(
        ["nasm", "-e", filename], stdout=subprocess.PIPE, text=True, check=True
    )
    labels: dict[str, int] = {}
    fail: bool = False
    with open(f"{''.join(filename.split('.')[:-1])}.out", "w+b") as outfile:
        print("==" * 3 + filename + "==" * 3)
        for num, line in enumerate(nasm.stdout.splitlines()):
            line = line.strip()
            here = outfile.tell()
            line = py_eval(line, locals())
            if DEBUG:
                print(f"{num:6d} : {here//2:#6x} : {line}")
            if (
                len(line) < 3 or (line[0] in ["#", "/", "%"]) or line == "None"
            ):  # ignore
                continue
            if ":" in line:  # label
                label_name = line.split(":")[0].strip()
                poz = outfile.tell() // 2
                if DEBUG:
                    print(f"{label_name} na {poz:#06x}")
                labels[label_name] = poz
            elif ".orgr" in line:  # jump in out file
                org = eval(line.removeprefix(".orgr").strip())
                outfile.seek(org, 1)
            elif ".org" in line:  # jump in out file
                org = eval(line.removeprefix(".org").strip())
                outfile.seek(org)
            elif ".db" in line:  # binary data dump
                data_bin: bytes = eval(line.removeprefix(".db ").strip())
                for part in (data_bin[i : i + 2] for i in range(0, len(data_bin), 2)):
                    if len(part) < 2:
                        part = part + b"\x00"
                    write_safe(outfile, part)
            else:  # assume instruction
                fail = not write_inst(outfile, line, locals())
            if fail:
```

```
                return False

        return True


def main():
    "Nisem ponosen na ta main"
    for opt in sys.argv:
        if opt.startswith("-") and "d" in opt:
            global DEBUG
            DEBUG = True
        if opt.startswith("-") and "n" in opt:
            return
    sys.exit(
        int(not all(map(cmp, filter(lambda x: not x.startswith("-"), sys.argv[1:]))))
    )


if __name__ == "__main__":
    main()
```

Listing 2: disasm.py

```
#!/usr/bin/env python3
"disassembler for 1 bit processor"

import sys


def as_hex(num, length=2):
    "formats as hex with zero padding"
    return "{0:#0{1}x}".format(num, length + 2)  # +2 to account for 0x


def disassm(filename: str) -> bool:
    "Deasseble input binary"
    with open(filename, "rb") as infile:
        print(Bcolors.BOLD + "==" * 3 + filename + "==" * 3 + Bcolors.ENDC)
        pc = 0  # Program counter is byteNo/2
        nops = 0  # Count adjacent nops
        while instruction := infile.read(2):
            instruction = int.from_bytes(instruction, byteorder="big")
            if instruction == 0 and nops:
                if nops == 1:
                    print("...")
                nops += 1
            elif instruction == 0:
                print(f"{as_hex(pc,4)} nop")
                nops = 1
            else:
                nops = 0
                op1 = (instruction & 0xFE00) >> 9
                op2 = (instruction & 0x01FC) >> 2
                ins = (
                    Bcolors.OKGREEN + "a" + Bcolors.ENDC
                    if (instruction & 0x0002) >> 1
                    else Bcolors.HEADER + "c" + Bcolors.ENDC
                )
                mag = Bcolors.FAIL + "1" + Bcolors.ENDC if instruction & 0x0001 else "0"
                print(
                    f"{as_hex(pc, 4)}    {ins} {as_hex(op1)}, {as_hex(op2)}, {mag}     {as_hex(instruction
                        , 4)}"
                )
            pc += 1
        return 1


def main():
    "Se en krasen main"
    sys.exit(
        int(
            not all(map(disassm, filter(lambda x: not x.startswith("-"), sys.argv[1:])))
```

```
        )
    )


class Bcolors:
    "vt100 color codes"
    HEADER = "\033[35m"
    OKBLUE = "\033[34m"
    OKCYAN = "\033[36m"
    OKGREEN = "\033[32m"
    WARNING = "\033[33m"
    FAIL = "\033[31m"
    ENDC = "\033[0m"
    BOLD = "\033[1m"
    UNDERLINE = "\033[4m"


if __name__ == "__main__":
    main()
```

Listing 3: example.asm

```
.org 4
%include "std.asm"

%macro print2 2
    c16 %1, %2, 1
    set_out_b %2 + 0 , 0x14
    set_out_b %2 + 1 , 0x14
    set_out_b %2 + 2 , 0x14
    set_out_b %2 + 3 , 0x14
    set_out_b %2 + 4 , 0x14
    set_out_b %2 + 5 , 0x14
    set_out_b %2 + 6 , 0x14
    set_out_b %2 + 7 , 0x14
    set_out_b %2 + 8 , 0x14
    set_out_b %2 + 9 , 0x14
    set_out_b %2 + 10, 0x14
    set_out_b %2 + 11, 0x14
    set_out_b %2 + 12, 0x14
    set_out_b %2 + 13, 0x14
    set_out_b %2 + 14, 0x14
    set_out_b %2 + 15, 0x14
%endm

word:
.db b"Zivjo\n"

main:
    print2 labels["word"]   , 0x15
    print2 labels["word"]+1, 0x15
    print2 labels["word"]+2, 0x15
    exit

init "main"
```

Listing 4: example2.asm

```
.org 4
%include "std.asm"

main:
    get_in_b 0x20,0x21
    set_out_b 0x20,0x21
    not IN_A
    c 1, 0, 1

init "main"
```

Listing 5: example3.asm

```
.org 4
pCell:
    .db b"\000\000"
loop:
    .db b"\000\000"

%include "std.asm"

znaka:
    .db b"#"
newl:
    .db b"\n"

%macro eval_cell 1
    a 2+%1, 0+%1
    a 1+%1, 0+%1
    a 2+%1, 0+%1
    x 1+%1, 0+%1
    a 0+%1, 0+%1
%endm

%macro do_cell 1
    cpy 0x26,0x25
    cpy 0x27,0x26
    cpy %1+1,0x27
    eval_cell 0x25
    cpy 0x25, %1
%endm

main:
    set1 0x4f
loop_f:
    set0 0x25
    set0 0x26
    set0 0x27
    do_cell 0x2f
    do_cell 0x30
    do_cell 0x31
    do_cell 0x32
    do_cell 0x33
    do_cell 0x34
    do_cell 0x35
    do_cell 0x36
    do_cell 0x37
    do_cell 0x38
    do_cell 0x39
    do_cell 0x3a
    do_cell 0x3b
    do_cell 0x3c
    do_cell 0x3d
    do_cell 0x3e
    do_cell 0x3f
    do_cell 0x40
    do_cell 0x41
    do_cell 0x42
    do_cell 0x43
    do_cell 0x44
    do_cell 0x45
    do_cell 0x46
    do_cell 0x47
    do_cell 0x48
    do_cell 0x49
    do_cell 0x4a
    do_cell 0x4b
    do_cell 0x4c
    do_cell 0x4d
    do_cell 0x4e
    do_cell 0x4f
    c labels["pCell"],0,1
```

```
%macro print_cell 1
    //.org alignto(here,)
    c labels["znaka"],0x15,1
    cpy %1, 0x19

    printc_m 0x15,0x14
%endm

print_cells:
    print_cell 0x30
    print_cell 0x31
    print_cell 0x32
    print_cell 0x33
    print_cell 0x34
    print_cell 0x35
    print_cell 0x36
    print_cell 0x37
    print_cell 0x38
    print_cell 0x39
    print_cell 0x3a
    print_cell 0x3b
    print_cell 0x3c
    print_cell 0x3d
    print_cell 0x3e
    print_cell 0x3f
    print_cell 0x40
    print_cell 0x41
    print_cell 0x42
    print_cell 0x43
    print_cell 0x44
    print_cell 0x45
    print_cell 0x46
    print_cell 0x47
    print_cell 0x48
    print_cell 0x49
    print_cell 0x4a
    print_cell 0x4b
    print_cell 0x4c
    print_cell 0x4d
    print_cell 0x4e
    print_cell 0x4f
    c labels["newl"], 0x15, 1
    printc_m 0x15,0x14
c labels["loop"],0,1

.org labels["pCell"]*2
.db by2(labels["print_cells"])
.org labels["loop"]*2
.db by2(labels["loop_f"])


init "main"
```

Listing 6: std.asm

```
%define a nand
%define c16 copy
%define x xor
%define nop copy 0,0,0

%macro not 1
    a %1,%1
%endm
%macro set0 1
    x %1,%1
%endm
%macro set1 1
    set0 %1
    not %1
%endm

%macro cpy 2
```

```
    xor %2,%2
    xor %1,%2
%endm

    ;; jumps to "main"
    ;; requires .org 4 at begining
    ;; main <-(args)
%macro init 1
    .org 2
    .db by2(labels[%1])
    .org 0
    c 1, 0, 1
%endm

    ;; wait for reg to become 1
    ;; reg, trash
%macro wait_for 2
    .org alignto(here,4,0)
    cpy %1, %2
    not %2
    a %2, 13
%endm

    ;; wait for reg to become 0
    ;; reg, trash
%macro wait_forn 2
    .org alignto(here,4,0)
    cpy %1, %2
    nop
    a %2, 13
%endm

    ;; set "input" to STDOUT
    ;; input, trash
%macro set_out_b 2
    wait_forn OU_A, %2
    cpy %1,OU
    not OU_A
%endm

    ;; get "output" from STDIN
    ;; output, trash
%macro get_in_b 2
    wait_for IN_A, %2
    cpy IN, %1
%endm

    ;; add together 2 addreses, store in second, 3 is carry
    ;; generated by brute force
    ;; A => A, B => A+B, Carry
    ;; 5 cycles
%macro add 3
    a %3, %3
    x %3, %2
    x %1, %3
    a %2, %3
    a %2, %2
    x %1, %2
    x %2, %3
%endm

%macro call 1
    set1 0x6e
    set0 0x6f
    c 0, 0x70, 0
    add 0x6e,0x7a,0x6f
    add 0x6e,0x79,0x6f
    not 0x6e
    add 0x6e,0x78,0x6f
    add 0x6e,0x77,0x6f
    add 0x6e,0x76,0x6f
    add 0x6e,0x75,0x6f
    add 0x6e,0x74,0x6f
```

```
    add 0x6e,0x73,0x6f
    add 0x6e,0x72,0x6f
    add 0x6e,0x71,0x6f
    add 0x6e,0x70,0x6f
    c labels[%1], 0, 1
    .orgr 34
%endm

%macro ret 0
    c 0x70, 0, 0
%endm

%macro exit 0
    .org alignto(here,2, 0)
    not 0xf
%endm

%macro printc_m 2
    set_out_b %1+ 0 , %2
    set_out_b %1+ 1 , %2
    set_out_b %1+ 2 , %2
    set_out_b %1+ 3 , %2
    set_out_b %1+ 4 , %2
    set_out_b %1+ 5 , %2
    set_out_b %1+ 6 , %2
    set_out_b %1+ 7 , %2
%endm
```

Listing 7: main.py

```
#!/usr/bin/env python3
"main_project_file"

import sys
import oneb_vm

DEBUG = False


def dbg(x: oneb_vm.VirtM):
    x.dump_state()


def main():
    "What_do_i_put_into_docstrings?"
    virtual_machine = oneb_vm.VirtM()
    virtual_machine.load("example3.out")
    for opt in sys.argv:
        if opt.startswith("-") and "d" in opt:
            global DEBUG
            DEBUG = True
    if DEBUG:
        virtual_machine.run(-1, dbg, 0.25)
    else:
        virtual_machine.run(-1)


if __name__ == "__main__":
    main()
```

Listing 8: oneb_vm.py

```
#!/usr/bin/env python3
"main_project_file"

import sys
import oneb_vm

DEBUG = False
```

```
def dbg(x: oneb_vm.VirtM):
    x.dump_state()


def main():
    "What_do_i_put_into_docstrings?"
    virtual_machine = oneb_vm.VirtM()
    virtual_machine.load("example3.out")
    for opt in sys.argv:
        if opt.startswith("-") and "d" in opt:
            global DEBUG
            DEBUG = True
    if DEBUG:
        virtual_machine.run(-1, dbg, 0.25)
    else:
        virtual_machine.run(-1)


if __name__ == "__main__":
    main()
```

Listing 9: gate-bf

```
#include <assert.h>
#include <math.h>
#include <pthread.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define THREADS 16
#define cells 3

int cell_s;
char **rez;
pthread_mutex_t rezL;

typedef struct {
  uint64_t start;
  uint64_t end;
  int len;
} job_t;

void printL(int len, int *list) {
  for (int i = 0; i < len; i++)
    printf("%d,", list[i]);
  printf("\n");
}

int fanint(int n) {
  int o = 0;
  for (int i = 0; i < cell_s; i++) {
    o |= ((i / (1 << (n))) % 2) << i;
  }
  return o;
}

uint64_t filw1(int len) { return (1 << len) - 1; }

uint64_t set(uint64_t in, int ind, int val) {
  in &= ~(filw1(cell_s) << (ind * cell_s));
  in |= (val & filw1(cell_s)) << (ind * cell_s);
  return in;
}

int get(uint64_t in, int ind) { return (in >> (ind * cell_s)) & filw1(cell_s); }

uint64_t init() {
  uint64_t o = 0;
  for (int i = 0; i < cells; i++) {
```

```
    o = set(o, i, fanint(i));
  }
  return o;
}

int nand(int x, int y) { return (~(x & y)) & filw1(cell_s); }

uint64_t effect(uint64_t state, int a, int b, int c) {
  state = set(state, b,
              c ? get(state, a) ^ get(state, b)
                : nand(get(state, a), get(state, b)));
  return state;
}

void *gate(void *in) {
  job_t *arg = in;
  long done = 0;
  for (uint64_t poz = arg->start; poz < arg->end; poz++) {
    if (rez[poz] != 0 && ((int)strlen(rez[poz]) == (arg->len - 1) * 3)) {
      done = 1;
      for (int x = 0; x < cells; x++)
        for (int y = 0; y < cells; y++)
          for (int m = 0; m < 2; m++) {
            uint64_t eff = effect(poz, x, y, m);
            if (rez[eff] == 0) {
              pthread_mutex_lock(&rezL);
              char *tmp = calloc(3 * arg->len + 1, sizeof(char));
              strncpy(tmp, rez[poz], (arg->len - 1) * 3);
              tmp[arg->len * 3 - 3] = '0' + x;
              tmp[arg->len * 3 - 2] = '0' + y;
              tmp[arg->len * 3 - 1] = '0' + m;
              rez[eff] = tmp;
              pthread_mutex_unlock(&rezL);
            }
          }
    }
  }
  free(in);
  return (void *)done;
}

int main() {
  assert(cells < 5);
  cell_s = 1 << cells;
  rez = calloc(1L << ((uint64_t)cells * cell_s), sizeof(char *));
  assert(rez != NULL);
  int j = 1;
  int to_do = 1;

  pthread_mutex_init(&rezL, NULL);

  char *def = malloc(sizeof(char));
  strcpy(def, "");
  rez[init()] = def;
  fprintf(stderr, "%ld\n", init());

  pthread_t niti[THREADS];

  while (to_do) {
    to_do = 0;
    for (int i = 0; i < THREADS; i++) {
      job_t job = {.start = i * (1 << (cells * cell_s)) / THREADS,
                   .end = (i + 1) * (1 << (cells * cell_s)) / THREADS,
                   .len = j};
      job_t *job2 = malloc(sizeof(job_t));
      memcpy(job2, &job, sizeof(job_t));
      pthread_create(&niti[i], NULL, gate, job2);
    }
    for (int i = 0; i < THREADS; i++) {
      void *out;
      pthread_join(niti[i], &out);
      long outi = (long)out;
      to_do |= outi;
```

```
    }
    fprintf(stderr, "%d\n", j++);
  }

  int najd = 0;
  for (int i = 0; i < 1 << (cells * cell_s); i++) {
    if (rez[i]) {
      printf("%#08x:%s\n", i, rez[i]);
      free(rez[i]);
      najd++;
    }
  }
  fprintf(stderr, "%d/%d\n", najd, 1 << (cells * cell_s));
  pthread_mutex_destroy(&rezL);
  free(rez);
  return 0;
}
```

9 Viri in literatura

[1] Andreas Abel, upops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures, ACM, 2019, dosegljivo: https://uops.info/ [23.3.2022]
[2] Crystal Chen, RISC architecture, 2000, dosegljivo: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/ [23.3.2022]
[3] Matthew Cook, Universality in Elementary Cellular Automata, Department of Computation and Neural Systems, 2004, dosegljivo: https://wpmedia.wolfram.com/uploads/sites/13/2018/02/15-1-1.pdf [23.3.2022]
[4] Stephen Dolan, mov is Turing-complete, Computer Laboratory, University of Cambridge, 2013, dosegljivo: https://harrisonwl.github.io/assets/courses/malware/spring2017/papers/mov-is-turing-complete.pdf [19.4.2022]