Organizacija Računalnikov

Domača naloga:
MiMo model CPE

Jan Ljubič,

63210192

Kazalo:

Kratek povzetek:

V tej seminarski nalogi je dopolnjen model CPE MiMo. Dodani so razni ukazi, programi, vezja ter dodatna naprava, ki skupaj obsegajo obveznosti 1. domače naloge pri predmetu Organizacija Računalnikov.

Skozi celotno seminarsko nalogo sem zelo podrobno spoznal delovanje preprostega CPE: vse od generiranja ustreznih .ram datotek, ki predstavljajo vsebino RAM pomnilnika, pa do prižiganja LED lučk na povezanih napravah. Naučil sem se kako dodajati ukaze v model CPE, kako iz ukazov sestavljati programe ter kaj točno se sploh zgodi v vsakem izmed ukazov, saj je vsak ukaz sestavljen iz več mikroukazov.

Izmed vseh nalog pa mi je bila najbolj zanimiva 4. naloga, ki je obsegala generiranje poljubnega vzorca na »Frame Buffer LED 16x16«.

1.) Opis ukaza "SW Rd,Immed"

Preden lahko nek program (predstavljen z zaporedjem ukazov) izvedemo, je potrebno modelu MiMo podati ustrezna "navodila" za izvajanje teh ukazov (mikroprogramska realizacija ukazov). Za prevajanje te datoteke z "navodili" uporabimo zbirnik (program **micro_assembler.exe**), ki nam datoteko **basic_microcode.def** prevede v datoteki **ucontrol.rom** ter **udecision.rom**. Te datoteki nato vnesemo v MiMO model:

- **ucontrol.rom** v Control ROM
- **udecision.rom** v Decision ROM

Ko so sta obe datoteki z implementiranimi ukazi vneseni, se lahko lotimo še ustvarjanja, prevajanja ter vnašanja testnega programa v RAM pomnilnik modela MiMo.

Najprej ustvarimo preprosto datoteko **sw_test.s**, kjer bomo testirali delovanje ukaza **SW Rd,Immed**. Ta datoteka vsebuje preprost ukaz z le eno vrstico: **main: sw r2, 43981**. Ob prevedbi programa dobimo datoteko **basic_program.ram**, ki predstavlja vsebino RAM pomnilnika. Prav tako nam prevajalnik v CMD (ukazna vrstica) izpiše spodnji 2 vrstici:

```
0000: 00008202  1000001000000010      main:   sw      r2, 43981
0001: 0000abcd  1010101111001101
```

To datoteko nato vnesemo v RAM pomnilnik v programu **Logisim Evolution**.

Pred izvedbo vsakega ukaza pa se v programu izvedeta še 2 dodatna ukaza, ki poskrbita za ustrezno nalaganje ter začetek izvajanja našega ukaza:

ZAPOREDNA ŠT. MIKROUKAZOV	MIKROUKAZI	RAZLAGA MIKROUKAZOV
1.	addrsel=pc irload=1	Prenos ukaza v ukazni register iz RAM pomnilnika (iz naslova 0000 se v ukazni register prebere vrednost 00008202, oziroma 1000001000000010 dvojiško): - prvih 7 bitov predstavlja **opcode** (operacijska koda), ki programu pove, kateri ukaz se bo izvedel. V našem primeru: **opcode** = 65 (1000001 dvojiško). - ostalih 9 bitov predstavljaja vrednosti **treg**, **sreg** ter **dreg** (zaporedno vsak register po 3 bite). V našem primeru: **dreg** = 2 (10 dvojiško).
2.	pcload=1 pcsel=pc, opcode_jump	Inkrementacija programskega števca (PC) za 1. Skok na naslednji naslov v RAM pomnilniku.

Ukaz **SW Rd, Immed** je eden izmed ukazov, ki so bili že implementirani v datoteki **basic_microcode.def.** Ukaz je sestavljen iz 3 mikroukazov, katerih razlage so v spodnji tabeli:

ZAPOREDNA ŠT. MIKROUKAZOV	MIKROUKAZI	RAZLAGA MIKROUKAZOV
1.	addrsel=pc imload=1	Na **podatkovno vodilo** (Data BUS) se iz **RAM pomnilnika** prebere vrednost ABCD (43981). Ta vrednost se prav tako naloži v **takojšni register** (ImmedReg).
2.	addrsel=immed datawrite=1 datasel=dreg	Vrednost iz registra **dreg** se zapiše na določeno lokacijo v RAM pomnilniku (v našem primeru na lokacijo ABCD oziroma 43981).
3.	goto pcincr (pcinr: pcload=1 pcsel=pc, goto fetch)	Skok na ukaz **pcincr**. Inkrementacija programskega števca (PC) za 1. Skok na naslednji ukaz.

2.) Implementacija strojnih ukazov:

Cilj drugega dela naloge je bila implementacija dodatnih ukazov, ki jih lahko uporabimo v CPE modelu MiMo. Vsi implementirani ukazi so prisotni v datoteki **basic_microcode.def,** v katero sem poleg že implementiranih ukazov dodal še ukaze, ki sem jih implementiral sam. Pred vsakim ukazom pa je potrebno izvesti še dva dodatna mikroukaza, ki poskrbita, da se ukaz začne izvajati:

ZAPOREDNA URINA PERIODA:	MIKROUKAZ	RAZLAGA MIKROUKAZA
1.	addrsel=pc irload=1	Nalaganje ukaza iz pomnilniške lokacije, na katero kaže **programski števec** (**PC)** v **ukazni register (IR)**
2.	pcload=1 pcsel=pc, opcode_jump	Inkrementacija **programskega števca** (PC) za 1, skok na **operacijsko kodo** **(**opcode)

Nato sledijo mikroukazi, ki skupaj predstavljajo dejanski ukaz. Spodaj je opisanih nekaj izmed njih, prav tako pa je prikazano število urinih period, ki jih mikroukazi potrebujejo:

rem Rd,Rs,Rt (4)

ZAPOREDNA URINA PERIODA:	MIKROUKAZ	RAZLAGA MIKROUKAZA
1.	aluop=rem op2sel=treg dwrite=1 regsrc=aluout	Izvajanje operacije **rem** (ostanek pri deljenju) z uporabo **ALE** med **Rs** in **Rt** ter zapis rezultata v Rd.
2.	goto fetch	Skok na ukaz **fetch**, ki naloži nov ukaz v **ukazni register**.

subi Rd,Rs,immed (17)

ZAPOREDNA URINA PERIODA:	MIKROUKAZ	RAZLAGA MIKROUKAZA
1.	addrsel=pc imload=1	Branje ter nalaganje **takojšnega operanda** iz pomnilnika v **takojšni register.**
2.	aluop=sub op2sel=immed dwrite=1 regsrc=aluout	Izvajanje operacije **sub** (odštevanje) z uporabo **ALE** med **Rs** in **immed** (takojšni operand) ter zapis rezultata v Rd.
3.	goto pcincr	Skok na ukaz **pcincr**, ki inkrementira **PC** za 1.

addc Rd,Rs,Rt,immed (31)

ZAPOREDNA URINA PERIODA:	MIKROUKAZ	RAZLAGA MIKROUKAZA
1.	addrsel=pc imload=1	Branje ter nalaganje **takojšnega operanda** iz pomnilnika v **takojšni register**.

2.	aluop=add op2sel=treg dwrite=1 regsrc=aluout	Izvajanje operacije **add** (seštevanje) z uporabo **ALE** med **Rs** in **immed** (takojšni operand) ter zapis rezultata v Rd.
3.	if c then jump else pcincr	Skok na ukaz **jump** če je prižgana zastavica **c** (carry). V nasprotnem primeru skok na ukaz **pcincr**.

jeq Rs,Rt,immed (33)

ZAPOREDNA URINA PERIODA:	**MIKROUKAZ**	**RAZLAGA MIKROUKAZA**
1.	addrsel=pc imload=1	Branje ter nalaganje **takojšnega operanda** iz pomnilnika v **takojšni register**.
2.	aluop=sub op2sel=treg dwrite=1 regsrc=aluout	Izvajanje operacije **sub** (odštevanje) z uporabo **ALE** med **Rs** in **immed** (takojšni operand) ter zapis rezultata v Rd.
3.	if z then pcincr else jump	Skok na ukaz **pcincr** če je prižgana zastavica **z** (zero). V nasprotnem primeru skok na ukaz **jump**.

jmp immed (45)

ZAPOREDNA URINA PERIODA:	**MIKROUKAZ**	**RAZLAGA MIKROUKAZA**
1.	addrsel=pc imload=1	Branje ter nalaganje **takojšnega operanda** iz pomnilnika v **takojšni register**.
2.	pcload=1 pcsel=immed	Skok **programskega števca** (PC) na vrednost **takojšnega operanda.**
3.	goto pcincr	Skok na ukaz **pcincr**, ki inkrementira **PC** za 1.

Vse skupaj sem implementiral 24 ukazov. 8 izmed njih je preprostejših (do številke 15) ostalih 16 pa bolj »zahtevnih«. V tabeli so našteti vsi implementirani ukazi:

Preprosti ukazi (do št. 15):	**Zahtevnejši ukazi (nad št. 15):**
add, sub, mul, div, rem, and, or, xor	addi, subi, muli, divi, addc, subc, jeq, jne, jgt, jle, jlt, jge, jeqz, jmp, swi, swri

3.) Testni programi:

Da sem preizkusil ukaze implementirane v 2. delu naloge, sem za vsak ukaz posebej ustvaril kratek testni program, ki je preizkusil delovanje ukaza. Spodaj je opisanih nekaj izmed teh programov:

ŠT. URINIH PERIOD	TESTNI PROGRAM	RAZLAGA

4
4
3
5
SKUPAJ: 16
	main: li r1, 8
	li r2, 3
	mul r3, r0, r1
	sw r3, 16
		Ta program testira ukaz **mul**. V registra **r1** ter **r2** zapišemo 2 poljubni števili, ki ju potem zmnožimo v register **r3** ter rezultat iz **r3** zapišemo na pomnilniško mesto 16.

4 4 5 **SKUPAJ: 14**	main: li r1, 24 divi r2, r1, 3 sw r2, 16	Ta program testira ukaz **divi**. V register **r1** se naloži poljubno število nato pa se v **r2** zapiše količnik števila v **r1** ter poljubnega števila podanega kot takojšni operand. Na koncu se rezultat iz **r2** zapišemo na pomnilniško mesto 16.
4 4 5 5 **SKUPAJ: 18**	main: li r1, 10 li r2, 20 subc r3, r1, r2, 16 sw r3, 24	Ta program testira ukaz **subc**. V registra **r1** ter **r2** se naložita poljubni števili, nato pa se izvede odštevanje katerega rezultat se zapiše v **r3**. Ker je rezultat negativen se vključi zastavica **carry**, kar povzroči skok programa na pomnilniško mesto 16 (mimo ukaza **sw**). V nasprotnem primeru do skoka ne bi prišlo in bi se program izvajal naprej brez skoka.
4 4 5 **SKUPAJ: 14**	main: li r1, 6 li r2, 6 jeq r1, r2, 16	Ta program testira ukaz **jeq**. V registra **r1** ter **r2** se naložita poljubni števili, nato pa se izvede primerjava števil. V primeru, da sta enaki (vključena zastavica **z** ali **zero**), se zgodi skok na pomnilniško mesto 16. V nasprotnem primeru pa se program izvaja naprej brez skoka.

Pri pisanju ukazov sem vsak ukaz testiral na podoben način kot zgoraj. Da pa sem testiral delovanje več ukazov skupaj, sem se odločil spisati še kratek program z dejansko uporabnostjo. Ta program za poljubno število pove ali je praštevilo ali ne. Spodaj je celotna koda ter razlaga logike programa.

ŠT. URINIH PERIOD	PROGRAM ZA TESTIRANJE PRAŠTEVIL	
4 4 4 5	main:	li r1, 1 li r2, 29 li r3, 2 jle r2, r1, notprime

3
5
4
3
5
	checkdividers:	rem r4, r2, r3
		jeqz r4, notprime
		addi r3, r3, 1
		mul r5, r3, r3
		jle r5, r2, checkdividers

4 4	prime:	li r6, 1 jmp save
4	notprime:	li r6, 2
5	save:	sw r6, 100
4	end:	jmp end
SKUPAJ: 58		

Število urinih period je ob dejanskem izvajanju mnogo višje, saj se zaradi zanke v »**checkdividers«** ta del kode večkrat izvede. V tabeli je prikazano le število urinih period vsakega ukaza, če bi se izvedel le enkrat.

Razlaga programa za testiranje praštevil:

Program deluje tako, da se najprej v **r1** naloži število 1, v **r2** število, za katero nas zanima ali je/ni praštevilo ter v **r3** začetni delitelj, ki ga bomo povečevali (na začetku 2).

Preden začnemo preverjati delitelje moramo preveriti če je število v **r2** negativno, 0 ali pa 1, saj to pomeni, da ni praštevilo. Potem sledi deljenje tega števila z deliteljem v **r3**.

Če je ostanek pri deljenju (**rem**) različen od 0, potem število v **r2** ni delitelj našega števila in se program pomakne na preverjanje naslednjega števila. To se ponavlja vse dokler se ne najde števila, ki deli naše število ali pa dokler delitelj ni večji kot koren našega števila (oziroma v našem primeru dokler je kvadrat delitelja manjši od našega števila). Sledi zapis števila 1 (predstavlja odločitev, da je število praštevilo) v register **r6**.

Če je ostanek pri deljenju (**rem**) enak 0, potem je to število delitelj našega števila in zaradi tega to število ne mora biti praštevilo. Sledi zapis števila 2 (predstavlja odločitev, da število ni praštevilo) v register **r6**.

V obeh primerih pa sledi zapis registra **r6** (odločitev, ali je število praštevilo ali ne) na pomnilniško mesto 100 ter neskončna zanka, ki označuje konec programa.

4.) Umestitev izhodnih naprav FB 16x16 in TTY v pomnilniški prostor:

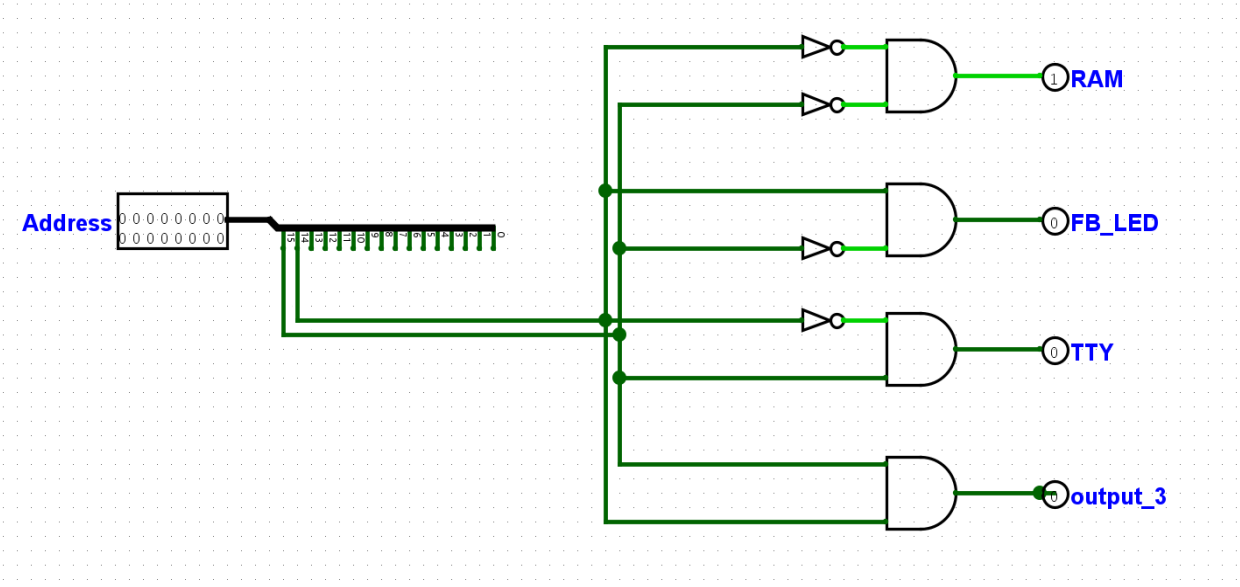
FB oziroma »**Frame Buffer LED 16x16**« je naprava, ki vsebuje mrežo LED diod velikosti 16x16. Vsaka LED dioda je lahko prižgana/ugasnjena.

TTY oziroma **terminal** pa je ekran za izpis znakov (črk, številk, ločil, posebnih znakov…).

Prva stvar, ki jo je zahtevala naloga, je bila dopolnitev vezja **naslovnega dekoderja** (address decoder). To vezje skrbi za preklapljanje med povezanimi napravami. V našem primeru smo imeli možnost povezave 4 naprav: **pomnilnika RAM**, **FB**, **TTY** ter **poljubne naprave**.

Naslovni dekoder kot vhod dobi naslov, katerega najvišja 2 bita predstavljata izhod, ki bo tisti trenutek aktiviran. Ker pa je na vsak izhod povezana naprava, to pomeni, da je posledično aktivirana ta naprava.

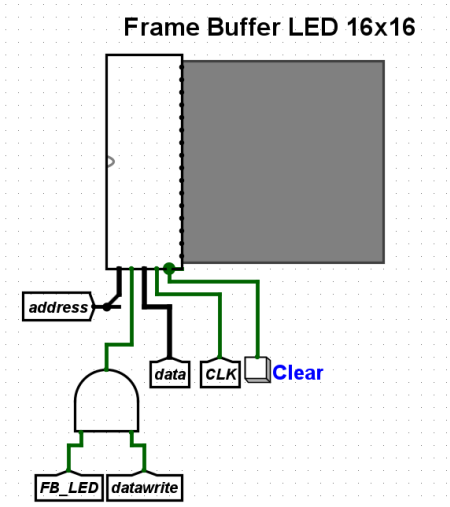
Shema dekoderja uporablja **razdelilnik** (**splitter**), da iz vhodnega naslova pridobi le prva 2 bita. Nato sledi preprosto aktiviranje ustreznega izhoda s pomočjo **AND** ter **NOT** vrat (npr. v primeru, da imamo bita 1 0, potem bo aktivirana naprava 3, kar je v našem primeru TTY).



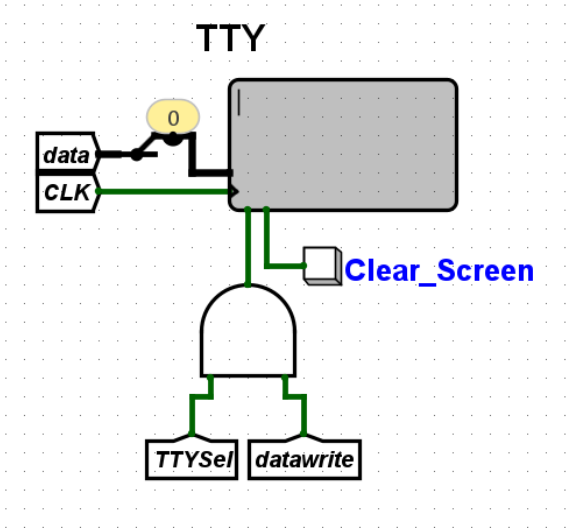
Slika 1: dodelano vezje naslovnega dekoderja

Poleg dekoderja je bilo potrebno v glavnem modelu MiMo dodati še preverjanje, da se na naprave zapisuje le, ko je poleg signala za napravo prižgan še **datawrite**, saj sta napravi **FB** ter **TTY** »write-only«, in se v njiju lahko zapisuje le, ko uporabljamo ukaze zapisovanja (v

nasprotnem primeru bi lahko naprava prejela napačen naslov, in bi bilo stanje v napravi nepredvideno) ter povezati **FB_LED** ter **TTYSel** z ustrezno napravo.



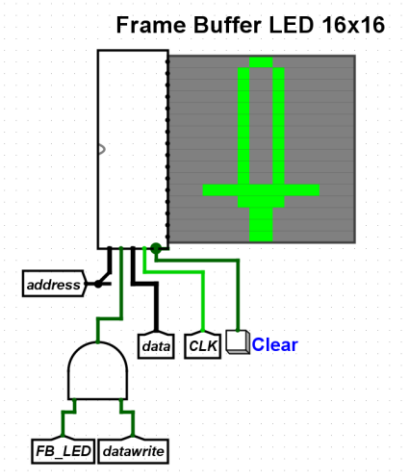
Slika 2: dodelano vezje Frame Buffer LED 16x16



Slika 3: dodelano vezje »TTY«

Kot zadnji del naloge pa je bilo potrebno v **FB** izrisati poljuben vzorec. To sem storil s spodnjo kodo (datoteka **FrameBufferLEDSword.s**), ki v **Frame Buffer LED 16x16** izriše meč:

```
main:       li      r1, 16384       # starting row
            li      r2, 0           # row counter
            li      r3, 3           # loop counter
            li      r4, 576         # blade
            li      r5, 8184        # guard
            li      r6, 960         # top handle part
            li      r7, 384         # handle & tip
handle:     swri    r7, r1, r2      # draw a handle row
            addi    r2, r2, 1       # next row
            subi    r3, r3, 1       # decrement counter
            jnez    r3, handle      # loop 3 times
tophandle:  swri    r6, r1, r2      # draw the top handle part
            addi    r2, r2, 1       # next row
guard:      swri    r5, r1, r2      # draw the guard
            addi    r2, r2, 1       # next row
            li      r3, 10          # reset loop counter
blade:      swri    r4, r1, r2      # draw a handle row
            addi    r2, r2, 1       # next row
            subi    r3, r3, 1       # decrement counter
            jnez    r3, blade       # loop 10 times
tip:        swri    r7, r1, r2      # draw the swort tip
end:        jnez    r1, end         # infinite loop
```



Slika 4: Izrisan meč v FB LED 16x16

Tukaj je prav tako povezava do posnetka, kjer je prikazan celoten izris meča:
https://youtu.be/xOcGs01ZgK4

5.) Vključitev dodatne vhodno-izhodne naprave:

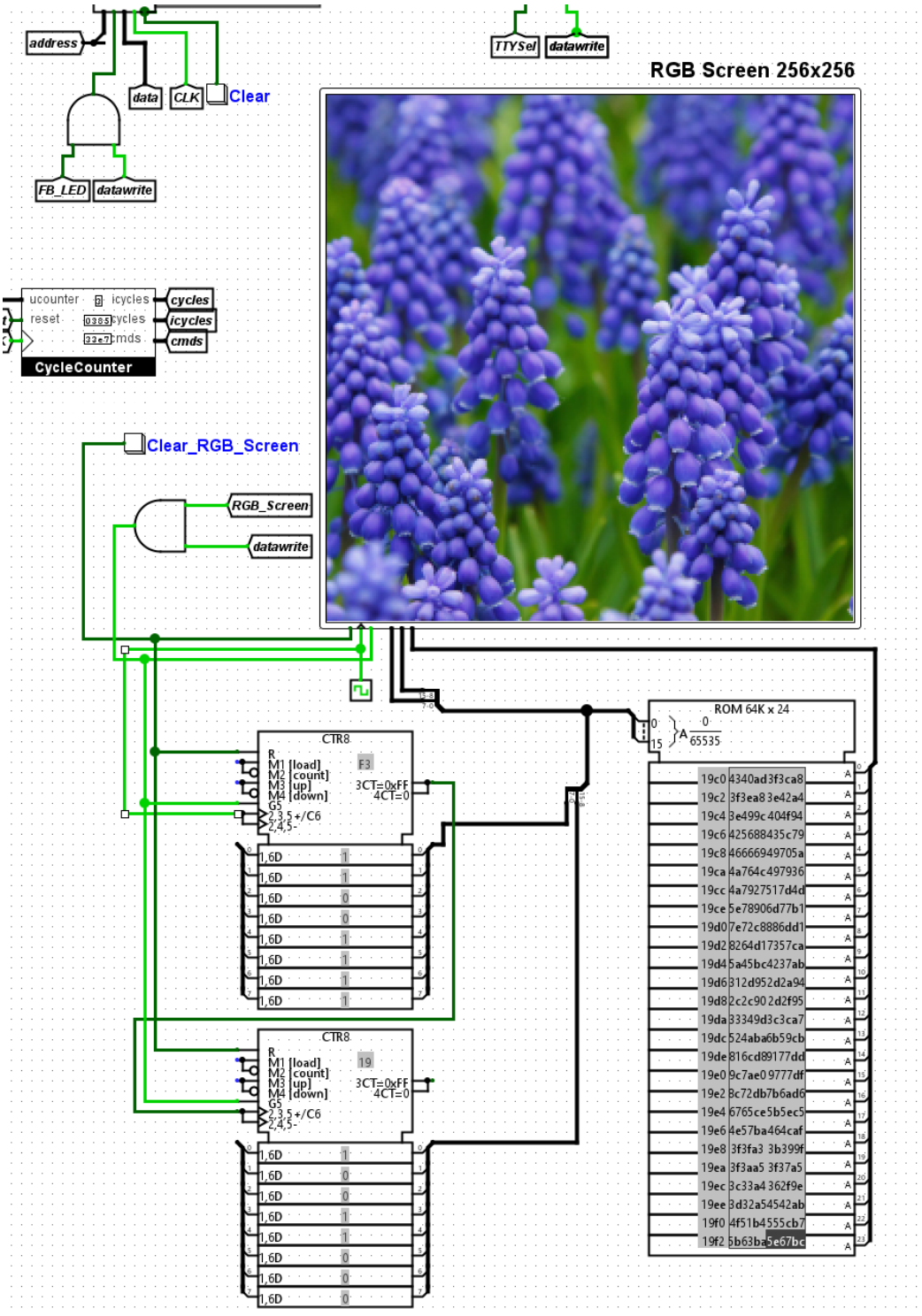
Kot zadnji del naloge pa sem moral v program **Logisim Evolution** dodati še eno poljubno napravo na podoben način, kot sta bila implementirana **Frame Buffer LED 16x16** ter **TTY**. Odločil sem se za **RGB Video** napravo, ki sem jo uporabil za izris slik velikosti 256x256 pikslov. Naprava se v programu nahaja v v mapi »**Input/Output**«.

RGB Video naprava deluje tako, da iz **ROM pomnilnika**, v katerem so naložene vrednosti barv vsakega piksla slike. Naprava nato s pomočjo dveh števcev (eden za širino slike ter eden za višino, oba velikosti 256) bere zaporedne vrednosti iz **ROM pomnilnika** (v njem je datoteka **blue-grape-hyacinths.htm)** ter jih zapiše na ustrezno mesto na **RGB ekranu**. Ta vrednost pa se nato translira v ustrezno barvo. To poteka od leve strani vrstice pikslov do desne, za vsako vrstico po vrsti od zgoraj navzdol.

Ker pa so vsi podatki že predhodno shranjeni v **ROM pomnilniku**, mi jih ni bilo potrebno pridobiti iz **RAM pomnilnika MiMo modela**. Poskrbeti sem moral le, da sem za vsak piksel shranil »navidezni podatek« na naslov moje naprave, saj se je s tem sprožila zastavica **datawrite** ter zastavica **RGB_Screen**, ki predstavlja mojo napravo v **Address Decoderju**.

To sem storil s programom **RGBScreenProgram.ram**:

```
main:       li r1, 49152          # address of the RGB Screen
writepixel: swi r2, r1, 0         # write a pixel
            jnez r1, writepixel   # loop
```



Slika 5: izris slike z RGB Video ekranom

Ker pa generiranje slike traja kar nekaj časa (+-15 min), sem se odločil posneti le del celotnega generiranja slike: https://youtu.be/cnpQj6IKTek

Literatura ter posnetki:

- Generiranje meča v »FB LED 16x16«:

https://youtu.be/xOcGs01ZgK4

- Generiranje slike v »RGB Video Screen«:

https://youtu.be/cnpQj6IKTek

- Inspiracija za »RGB Video Screen«:

https://www.youtube.com/watch?v=xJ8Ll2qJQR4&list=WL&index=26&t=324s