1. Jasno zapišite **število in zaporedje mikroukazov**, ki se izvedejo pri izvedbi strojnega ukaza "li Rd, Immed "

Primer: `li r3, 150`

Mikrokoda: `addrsel=pc  dwrite=1  regsrc=databus, goto pcincr`

- `addrsel=pc` multiplekser za izbor naslova nastavi programski števec. (vrednost pc=0)
- `dwrite=1` omogoči se pisanje v D register (Rd, v primeru `r3`)
- `regsrc=databus` multiplekser za vnos v register izbere podatkovno linijo (vrednost databus=0). Delovalo bi tudi `regsrc=immed` (vrednost v multiplekserju=1).
- `goto pcincr` skoči na »`pcincr`«. Programski števec se poveča za 2 (ukaz + takojšnji operand).

Skupno število mikroukazov: 3 (fetch + mikroukaz + pcincr)

2. Iz seznama strojnih ukazov, ki jih podpira zbirnik **izberite vsaj štiri** in zanje podajte ustrezna zaporedja mikroukazov za njihovo realizacijo.

Implementiral sem vse opisane ukaze, tu bom opisal 4 in preostale navedel spodaj. Tu bom pokazal ukaze ki uporabljajo sklad (push, pop, jsr, rts). Implementacije preostalih bodo podane v datoteki basic_microcode.def .

- ```
  #jsr immed (59)
  # R7--, LOAD IMM, PC ++
  59:  aluop=sub  op2sel=const1  swrite=1  regsrc=aluout imload=1
  # PC++
  pcload=1  pcsel=pc
  # M[R7], PC <- immed
  addrsel=sreg datasel=pc datawrite=1, goto jump
  ```
- ```
  #rts  (60)
  #    PC <- M[R7]
  60:  addrsel=sreg imload=1
  pcload=1 pcsel=immed
  # R7++
  aluop=add  op2sel=const1  swrite=1  regsrc=aluout, goto fetch
  ```
- ```
  #push Rd (68)
  # R7--
  68:  aluop=sub  op2sel=const1  swrite=1  regsrc=aluout
  # M[R7] <- Rd  PC <- PC + 1
  addrsel=sreg datawrite=1 datasel=dreg, goto fetch
  ```
- ```
  #pop  Rd (69)
  # Rd <- M[R7]
  69:  addrsel=sreg dwrite=1 regsrc=databus
  # R7++  PC <- PC + 1
  aluop=add  op2sel=const1  swrite=1  regsrc=aluout, goto fetch
  ```

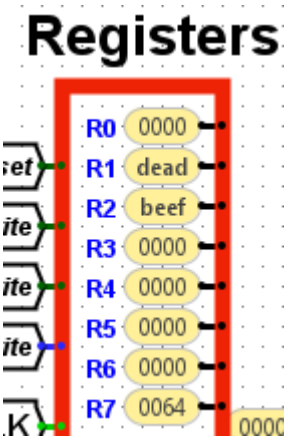
3. Nove strojne ukaze iz 2. naloge **uporabite v lastnem testnem programu** (enem ali večih) v zbirniku in preizkusite njihovo delovanje. Razložite vsebino ustreznih strojnih ukazov. Opišite dogajanje ob vsakem strojnem ukazu in določite, **koliko urinih period traja** vsak od njih in vsi skupaj (zapišite trajanja za vsak ukaz v številu urinih period v posebni tabeli). Opišite testne programe in jih vključite v poročilo.

```
# Ob koncu izvajanja:
# r1 = 0xdead
# r2 = 0xbeef
# r7 = 100
# r5 = 0
# r6 = 0

 li  r7, 100          # stack pointer
 li  r1, 0xbeef
 li  r2, 0xdead
 li  r5, 0
 li  r6, 0
 jsr reverse
 br  _end
 li r6, -1  # naj se nebi izvedlo


reverse:   push r1
           push r2
           pop  r1
           pop  r2
           rts
           li r5, -1  # naj se nebi izvedlo
_end:   br  _end        # Konec
```

Testni program inicializira kazalec na sklad na pomnilniški naslov 100, nato v r1 shrani vrednost 0xbeef in v r2 0xdead ter počisti registra r5 in r6. Zatem skoči v podprogram »reverse«, ki zamenja vrednosti v registrih r1 in r2 s pomočjo sklada. Ko se vrne iz funkcije skoči na »_end«, kjer se ustavi (oziroma ostane v neskončni zanki.

Če vsi ukazi delujejo pravilno naj bi bile vrednosti v registrih ob koncu programa:

r1 = 0xDEAD
r2 = 0xBEEF
r5 = 0
r6 = 0
r7 = 100 (0x64)
Na sliki desno so vrednosti registrov ob koncu izvajanja.



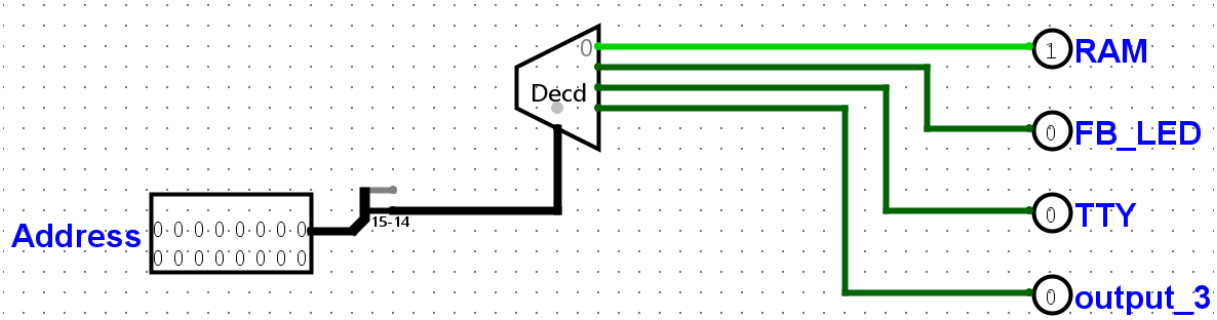
Trajanje ukazov po urinih periodah (CPI, cycles per instruction):

- `jsr` – v sklad zapiše naslov naslednjega ukaza, nato skoči na naslov takojšnjega operanda – 5 CPI
- `rts` – z sklada pobere povratni naslov in nanj nastavi programski števec – 4 CPI
- `push` – na sklad porine vrednost v podanem registru - 3 CPI
- `pop` – z sklada pobere vrednost in jo zapiše v podani register - 3 CPI

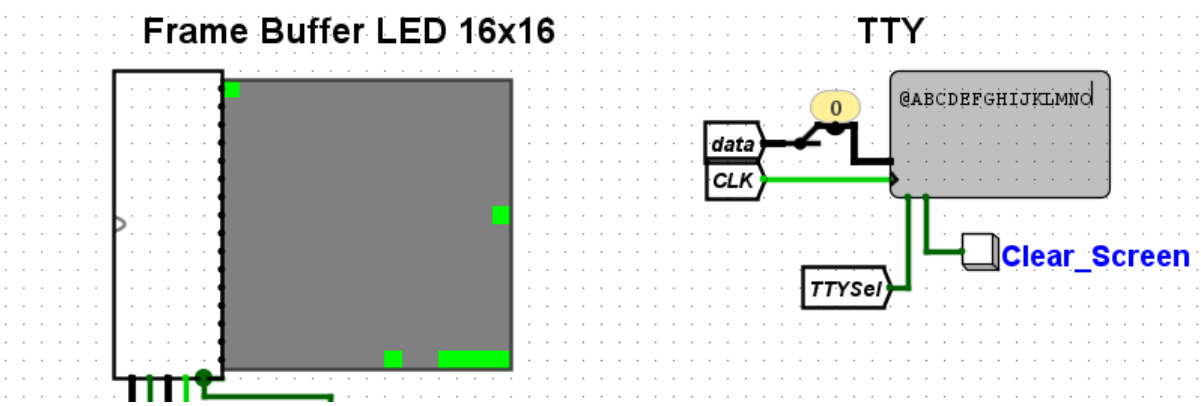
Za demonstracijo delovanja rekurzije bom spodaj prilepil program za izračun fibonaccijevega števila.

4. **Pravilno umestite v pomnilniški prostor izhodni napravi FB 16x16 in TTY** po načelu "pomnilniško preslikanega vhoda/izhoda"- lahko uporabite nepopolno naslovno dekodiranje. Poskrbite, da se bo testni program za IO napravi po tej spremembi pravilno izvajal - torej se bosta izhodni **napravi pojavili za pisanje res samo na njima dodeljenih naslovih**. Spremenite testni program, da bo izrisoval vzorec na FB napravi po vaši zamisli.

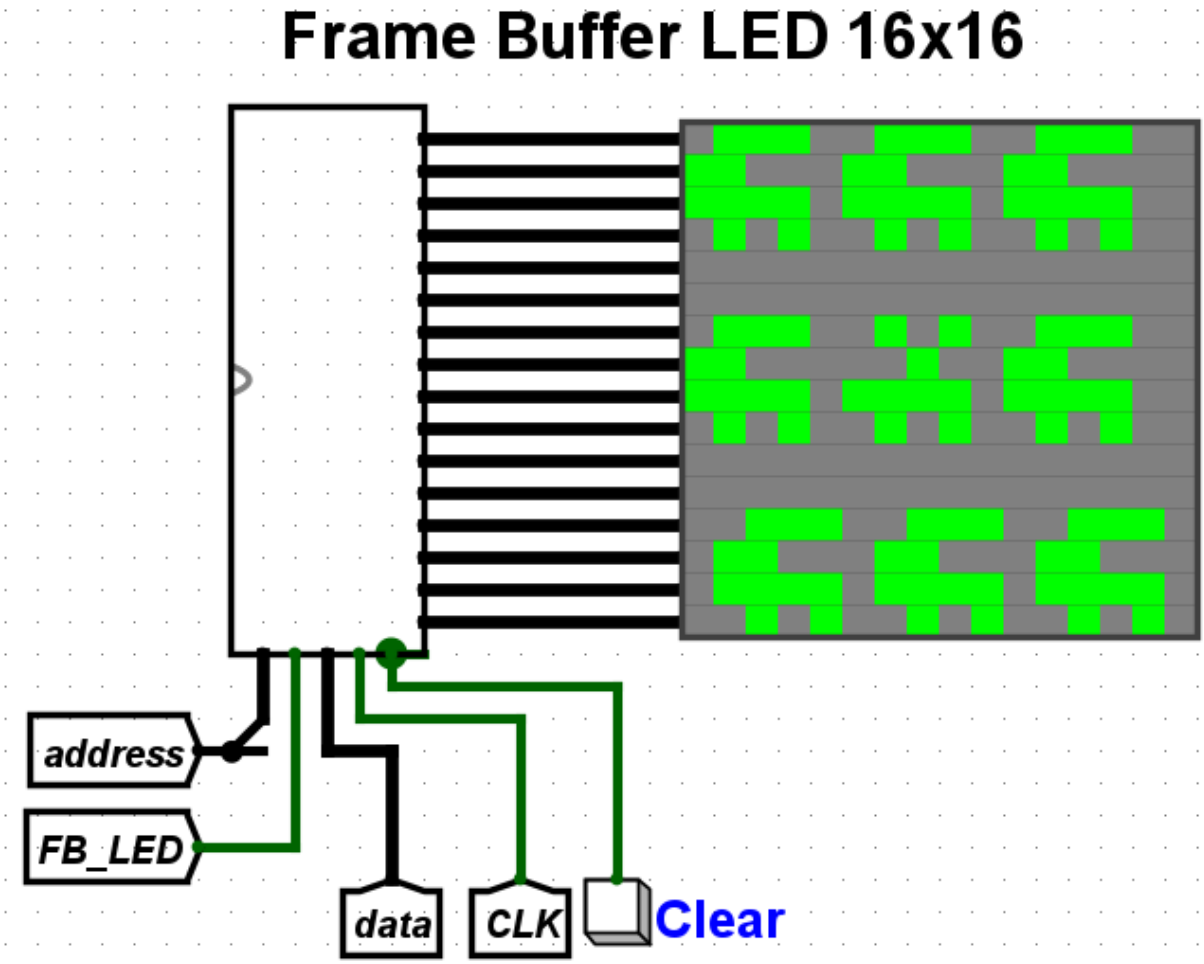
Za umestitev pomnilniškega prostora sem v dekodirniku naslovov s pomočju razdelilnika 2 najbolj pomembna bita napotil v dekoder, ki izbere pravilno napravo. Nato sem v posamezne naprave napotil vrednost iz dekoderja (namesto konstant).



Po koncu izvajanja originalnega testnega programa (spremenjenega le tako da se ne izvaja v nedogled) sta napravi FB_LED in TTY v stanju vidnem na sliki.



FB_LED z mojim vzorcem:

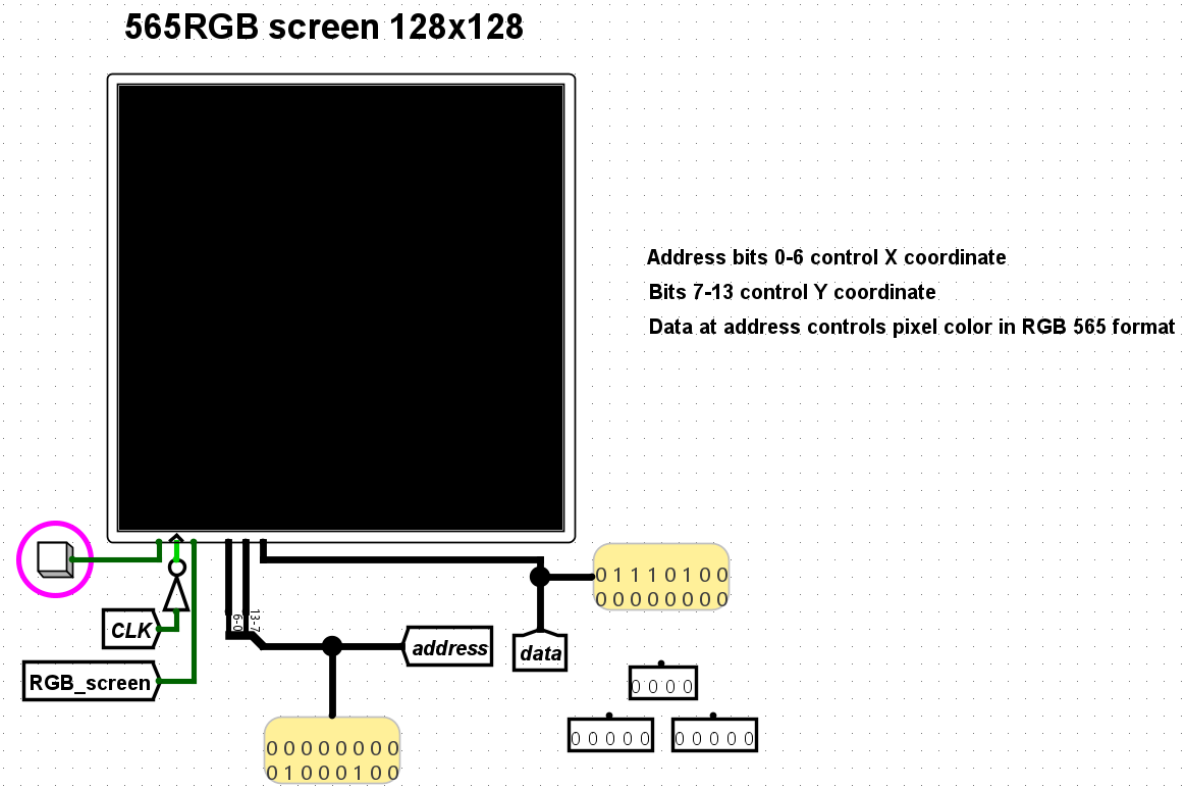


5. V MiMo modelu je pripravljen prostor za še eno vhodno izhodno napravo. Opišite vsa potrebna opravila za **pravilno vključitev dodatne vhodno izhodne naprave** v MiMo model in jo tudi realizirajte (vsaj v enostavnejši obliki) ter preizkusite z ustreznim programom v zbirniku. Podajte jasen opis vseh korakov z vsemi potrebnimi podrobnostmi. V poročilu dodano napravo tudi predstavite in ustrezno opišite.

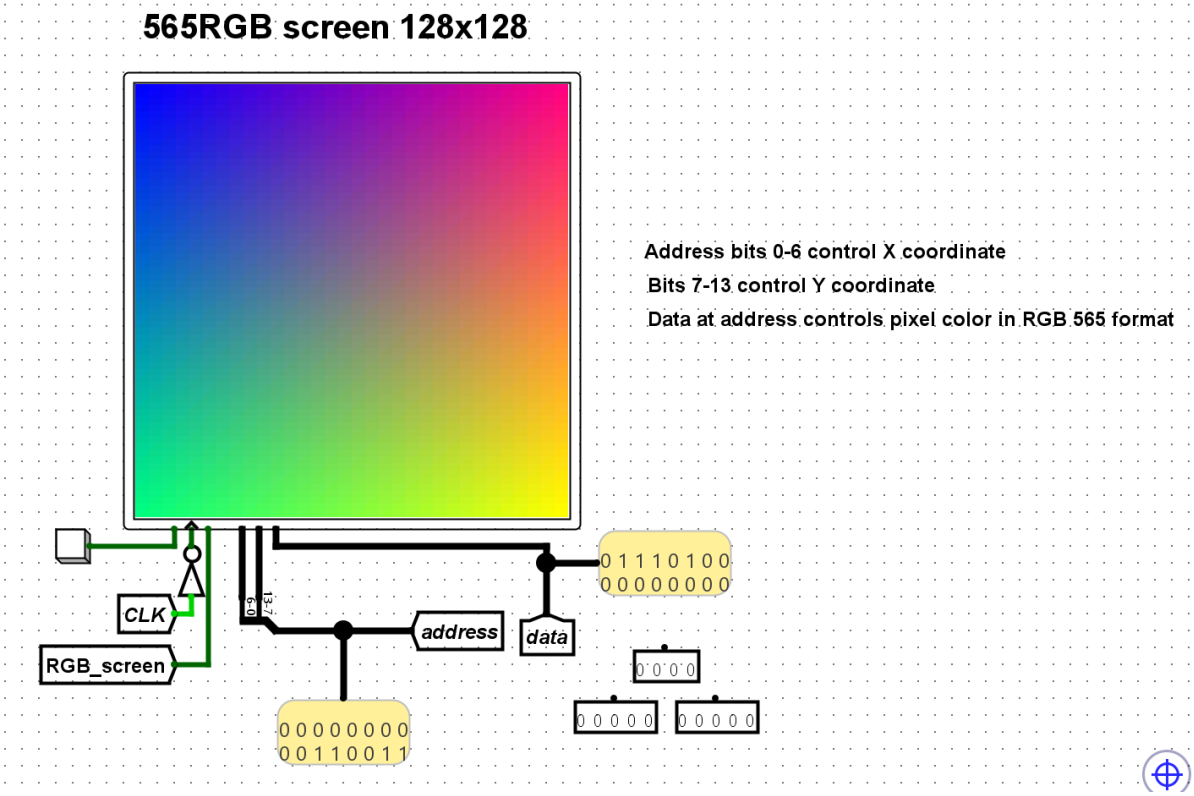
Za dodatno izhodno napravo sem priklopil 128x128 pikseln zaslon, ki uporablja barvni format RGB 565. Zaslona nisem naredil sam, vgrajen je v program Logisim Evolution.

Bita 15 in 14 sta uporabljena za naslovno preslikavo, biti 13 do 7 predstavljajo Y koordinato, biti 6 do 0 pa X koordinato. Vrednost na naslovu predstavlja barvo piksla in sicer biti 15-11 predstavljajo rdečo, 10-5 zeleno in biti 4-0 modro komponento barve.

Da sem pisanje na zaslon uskladil z procesorjem, sem urin signal pred zaslonom napotil skozi NOT vrata. Slika zaslona v mirovanju:



Za demonstracijo zaslona sem uporabil preprost program, ki izriše barvni preliv.



Zaradi velikosti zaslona in omejitev simulacije je izvajanje programov ki rišejo nanj počasno. Zgornji program je za izvajanje potreboval skoraj uro (nastavljena hitrost simulacije 1 kHz, višja hitrost ni imela vidnega vpliva).

3.1 Dodatna demonstracija jsr in rts – izračun Fibonaccijevega števila

Spodnji program z repno rekurzijo izračuna n-to Fibonaccijevo število do omejitve 16 bitnih števil (32.767). N je shrajen v registru r0.

```
    li r7, 1024     # stack pointer
    li r0, 10       # n-th digit
    li r1, 0        # a = 0
    li r2, 1        # b = 1
    li r3, 0         # result
    jsr fib
    jmp _end
fib: push r0         # Save n
          push r1         # Save a
          push r2         # Save b

          # if n <= 0, return a
          li r4, 0
          ble r0, r4, base_case

          subi r0, r0, 1     # n = n - 1
          move r5, r1        # save old a in temp (r5)
          move r1, r2        # a = b
          add  r2, r5, r2    # b = old a + b

          jsr fib       # recurse
          jmp fib_return

base_case:move r3, r1        # result = a

fib_return:     pop r2
                pop r1
                pop r0
                rts
_end:   br _end
```