Univerza v Ljubljani
Fakulteta za računalništvo
in informatiko

ORGANIZACIJA RAČUNALNIKOV

1. DOMAČA NALOGA – MIMO MODEL

Luka Šveigl, 63200301

1. NALOGA:

Zapišite **zaporedje mikroukazov**, ki se izvedejo pri izvedbi strojnega ukaza "**SW Rd,immed**" (primer: "SW r2, 65535")

- ta ukaz je že implementiran v datoteki z definicijami mikroprogramskih realizacij
- ob vsakem mikroukazu krajše **opišite njegovo delovanje** (stanje podatkovne enote, kaj se pravzaprav takrat zgodi, itd....)

Ukaz SW Rd, immed ("store word", "register d, "immediate operand") shrani "word" iz registra **Rd** v pomnilnik na lokacijo, na katero kaže takojšnji operand **immed.**

V basic_microcode.def je ta ukaz definiran v dveh vrsticah, prva vrstica vsebuje ukaze ***addrsel=pc imload=1***, druga vrstica pa vsebuje ukaze ***addrsel=immed datawrite=1 datasel=dreg, goto pcincr***.

1.1 Analiza prve vrstice:

V prvi vrstici ukaza SW (***addrsel=pc imload=1***) naložimo takojšnji operand. Ta vrstica sestoji iz 2 mikroukazov, ki sta:

- addrsel=pc, ta ukaz pridobi naslov v programskem števcu
- imload=1, ta ukaz pa določi, da pomnimo takojšnji operand ukaza

1.2 Analiza druge vrstice:

V drugi vrstici ukaza SW (***addrsel=immed datawrite=1 datasel=dreg, goto pcincr***) uporabimo vrednost takojšnjega operanda kot naslov, v katerega zapišemo vrednost registra Rd, nato pa se skočimo na oznako pcincr, v kateri inkrementiramo PC (programski števec), da preskočimo takojšnji operand, nato pa skočimo na nov ukaz. Ta vrstica sestoji iz 4 mikroukazov, ki so:

- addrsel=immed, ta ukaz pridobi vrednost takojšnjega operanda kot naslov
- datawrite=1, ta ukaz določa, da v naslov pišemo
- datasel=dreg, ta ukaz določa izvor podatkov za pisanje
- goto pcincr, ta ukaz skoči na oznako pcincr, katera nas premakne na nov ukaz

1.3 Analiza dejanske izvedbe ukaza:

Pri dejanski izvedbi ukaza SW sem uzel kar primer, podan v navodilu, torej SW r2, 65535.

Po izvedbi prve vrstice (***addrsel=pc imload=1***), je bilo stanje v podatkovni enoti sledeče:

- Data Bus = ffff
- Address Bus = 0003
- Imm. Reg = ffff
- IR = 8202
- PC = 0003
- Imload = 1

Po izvedbi druge vrstice (***addrsel=immed datawrite=1 datasel=dreg***), pa je bilo stanje v podatkovni enoti sledeče:

- Data Bus = 0003
- Address Bus = ffff
- Imm. Reg = ffff
- IR = 8202
- PC = 0003
- Datawrite = 1

2. NALOGA:

Iz seznama strojnih ukazov, ki jih podpira zbirnik **izberite vsaj tri** in zanje podajte ustrezna zaporedja mikroukazov za njihovo realizacijo. **Izbrani ukazi** naj bodo **bolj zahtevni oz. netrivijalni** in iz različnih skupin - sestavljeni naj bodo iz vsaj 4 ali še bolje več mikroukazov (vaša uspešnost se bo merila tudi po tem kriteriju). Kratko opišite realizacijo in razložite predvsem mikroprogramske zapise dodanih strojnih ukazov v mikro-zbirniku:

- potrebno je določiti **realizacijo strojnega ukaza s pomočjo mikroukazov**
- zaporedje mikroukazov je potrebno dodati v datoteko "**basic_microcode.def**" in jo vnesti ter uporabiti v MiMo modelu.

V sklopu te naloge sem se odločil, da bom implementiral vse osnovne ukase aritmetično logične enote, vse ukase aritmetično logične enote, ki uporabljajo takojšnji operand, ukaz JEQ za skok, ukaz LW za nalaganje podatkov iz pomnilniških naslovov, in ukaze MOVE, NEG in CLR.

2.1 Implementacija osnovnih ukazov ALE:

Zaradi lažjih testnih programov, in zaradi lažjega izrisovanja vzorca na napravo FB 16x16 sem se odločil, da bom implementiral vse osnovne ukase ALE, saj je njihova mikroprogramska koda izredno preprosta.

Vsak osnovni ukaz sestoji iz 5 mikroukazov in je oblike ***aluop=”ime_ukaza” op2sel=”drugi_operand” dwrite=”1 ali 0” regsrc=”kraj_rezultata”, goto fetch***.

Realizacija ukaza sub je torej zgledala tako:

- aluop=sub op2sel=treg dwrite=1 regsrc=aluout, goto fetch

Če pogledamo, kaj dela vsak izmed teh mikroukazov, ugotovimo sledeče:

- aluop=sub, ta ukaz določi operacijo, ki se bo izvedla, v tem primeru odštevanje
- op2sel=treg, ta ukaz določi drugi operand
- dwrite=1, ta ukaz določi pisalno operacijo iz vhoda v izbrane registre
- regsrc=aluout, ta ukaz določi ponor pisanja
- goto fetch, ta ukaz skoči na oznako fetch, v kateri skočimo na naslednji ukaz

Isti postopek sem ponovil za vseh 15 osnovnih ukazov ALE, pri čemer sem le spreminjal vrednost mikroukaza aluop.

2.2 Implementacija ukazov ALE z takojšnjim operandom:

Zaradi istih razlogov, da sem implementiral osnovne ukaze ALE sem se odločil, da bom implementiral tudi ukaze ALE, ki uporabljajo takojšnje operande, saj je bila tudi implementacija le teh trivialna.

Vsi taki ukazi sestojijo iz 2 vrstic, v prvi sta mikroukaza ***addrsel=pc imload=1***, katera določata da naložimo takojšnji operand, druga vrstica pa je identična istemu osnovnemu ukazu ALE, torej je oblike ***op2sel="drugi_operand" dwrite="1 ali 0" regsrc="kraj_rezultata", goto pcincr***.

Ker sem pomene vseh mikroukazov, uporabljenih v tem poglavju že razložil,v prejšnjih poglavjih jih tu nebom ponovno razlagal, ampak bom pokazal le primer za odštevanje, ki pa je sledeč:

```
17: addrsel=pc  imload=1
    aluop=sub  op2sel=immed  dwrite=1  regsrc=aluout, goto pcincr
```

2.3 Implementacija ukaza JEQ:

Ker navodilo te naloge zahteva, da implementiramo ukaze iz različnih skupin, sem se odločil, da bom implementiral tudi en vejitveni ukaz, torej **JEQ Rs,Rt,"immed"** oziroma "Jump if equal", ki deluje tako, da skoči na naslov določen z takojšnjim operandom, če sta vrednosti v registrih Rs in Rt enaki.

Definicija tega ukaza pravi: ***if Rs == Rt, PC <-immed else PC <- PC + 2***

Ta ukaz sem realiziral tako, da sem registra odštel, in v primeru da je bila po odštevanju zastavica Z (zero) nastavljena, skočil na naslov shranjen v immed, drugače pa sem skočil na naslednji ukaz.

Mikroprogramska realizacija tega ukaza sestoji iz 2 vrstic, prva vsebuje mikroukaza ***addrsel=pc imload=1***, druga pa vsebuje mikroukaze ***aluop=sub op2sel=treg, if z then jump else pcincr***.

Ker sem večino mikroukazov, uporabljenih v tem ukazu že opisal, bom opisal novi mikroukaz, torej ***if z then jump else pcincr***. Ta ukaz preveri zastavico Z, in v primeru da je nastavljena, skoči na naslov shranjen v takojšnjem operandu, drugače pa se pomakne na naslednji ukaz.

Celotna realizacija tega ukaza:

```
33: addrsel=pc  imload=1
    aluop=sub  op2sel=treg, if z then jump else pcincr
```

2.4 Implementacija ukaza LW:

Ker je bil v datoteki basic_microcode.def že definiran ukaz **SW Rd,"immed"** sem se odločil, da bom implementiral tudi ukaz **LW Rd,"immed"**, ki nam omogoča, da iz podatkovnega naslova, določenega v takojšnjem operandu preberemo vrednost v register Rd.

Definicija tega ukaza pravi: ***Rd <- M[immed], PC <- PC + 2***

Ta ukaz sestoji iz 2 vrstic, prva vsebuje mikroukaza ***addrsel=pc imload=1***, v drugi pa so mikroukazi ***addrsel=immed dwrite=1 regsrc=databus, goto pcincr.***

Edina novost tu je ***regsrc=databus***, ki določa,da vhod izhaja iz podatkovnega vodila.

Celotna realizacija tega ukaza:

```
64: addrsel=pc  imload=1
    addrsel=immed  dwrite=1  regsrc=databus, goto pcincr
```

2.5 Implementacija ukaza MOVE:

Ukaz **MOVE Rd,Rs** nam omogoča, da vrednost registra Rs neposredno premaknemo v register Rd. Ta ukaz sestoji le iz 3 mikroukazov, ki so ***dwrite=1 regsrc=sreg, goto fetch***.

Definicija tega ukaza pravi: ***Rd <- Rs, Pc <- PC + 1***

Celotna implementacija tega ukaza:

```
70:  dwrite=1  regsrc=sreg, goto fetch
```

2.6 Implementacija ukaza NEG:

Ukaz NEG Rs nam omogoča, da negiramo vrednost, ki je shranjena v registru Rs. Čeprav ukaz zgleda preprosto, v ozadju deluje kar 9 mikroukazov.

Definicija tega ukaza pravi: ***Rs <- -Rs, PC <- PC + 1***

Celoten ukaz je sestavljen iz 2 vrstic, prva vsebuje mikroukaze ***aluop=not op2sel=treg swrite=1 regsrc=aluout***, druga pa vsebuje mikroukaze ***aluop=add op2sel=const1 swrite=1 regsrc=aluout, goto fetch.*** Ukaz NEG deluje tako, da najprej nad vrednostjo v registru izvede logično negacijo, nato pa negirani vrednosti prišteje vrednost ena (***op2sel=const1***).

Celotna impelmentacija tega ukaza:

```
72: aluop=not  op2sel=treg  swrite=1  regsrc=aluout
    aluop=add  op2sel=const1  swrite=1  regsrc=aluout, goto fetch
```

2.7 Implementacija ukaza CLR:

Za konec sem implementiral še ukaz CLR Rs, ki izprazni register Rs, torej njegovo vrednost postavi na 0.

Definicija tega ukaza pravi: ***Rs <- 0, PC <- PC + 1***

Celoten ukaz je sestavljen iz ene vrstice, ki vsebuje mikroukaze ***aluop=and op2sel=const0 swrite=1 regsrc=aluout, goto fetch.***

Celotna implementacija tega ukaza:

```
71: aluop=and  op2sel=const0  swrite=1  regsrc=aluout, goto fetch
```

3. NALOGA:

Nove strojne ukaze iz 2. naloge **uporabite v lastnem testnem programu** (enem ali večih) v zbirniku in preizkusite njihovo delovanje. Napišite tak(e) program(e), da se bodo dodani ukazi temeljito preizkusili. Razložite vsebino ustreznih strojnih ukazov. Opišite dogajanje ob vsakem strojnem ukazu in določite, **koliko urinih period traja** vsak od njih (zapišite trajanja za vsak ukaz v številu urinih period v posebni tabeli).

V sklopu te naloge sem napisal 4 testne programe, vsak prikazuje delovanje enega sklopa dodanih ukazov (osnovni ALE ukazi, ALE ukazi z takojšnjim operandom, skok JEQ, ukazi MOVE, NEG, CLEAR). Vsi programi se nahajajo v direktoriju *distribucija/test_programs*.

3.1 Testni program za osnovne ALE ukaze:

```
main:  li    r0, 50     # Load values into registers r0 and r1
       li    r1, 10
       add   r0, r0, r1  # Add value of r1 to value of r0
       neg   r1                # Negate r1
       add   r0, r0, r1
       neg   r1
       sub   r0, r0, r1  # Subtract value of r1 from value r0
       neg   r1
       sub   r0, r0, r1
       neg   r1
       mul   r0, r0, r1  # Multiply values of r1 and r0, store to r0
       neg   r1
       mul   r0, r0, r1
       neg   r0
       neg   r1
       div   r0, r0, r1  # Divide value of r0 by value of r1, store to r0
       rem   r0, r0, r1  # Calculate remainder of division of r0 by r1, store to r0
```

Ta program naloži v register r0 vrednost 50, v register r1 pa vrednost 10, nato pa izvaja po mojem mnenju najpomembnejše ALE ukaze, vmes pa tudi negira vrednost registra r1, kar nam omogoča, da testiramo tudi, ali ukazi delujejo tudi z negativnimi vrednostmi.

Ukaz	Št. ponovitev	Trajanje [UP]	Skupno [UP]
li	2	2	4
add	2	2	4
sub	2	2	4
mul	2	2	4
div	1	2	2
rem	1	2	2
neg	7	3	21

3.2 Testni program za ALE ukaze z takojšnjim operandom:

```
main:   li      r0, 50          # Load value 50 into register r0
        addi    r0, r0, 23      # Add 23 (immediate operand) to r0
        addi    r0, r0, -1      # Add -1 to r0
        subi    r0, r0, 15      # Subtract 15 from r0
        subi    r0, r0, -1      # Subtract -1 from r0
        muli    r0, r0, 6       # Multiply r0 by 6
        muli    r0, r0, -1      # Multiply r0 by -1
        neg     r0              # Negate r0
        divi    r0, r0, 2       # Divide r0 by 2
        remi    r0, r0, 2       # Store remainder of division of r0 and 2 in r0
```

Ta program naloži v register r0 vrednost 50, nato pa nad tem registrom izvaja različne operacije ALE z takojšnjim operandom, ki je v nekaterih primerih tudi negativen.

Ukaz	Št. ponovitev	Trajanje [UP]	Skupno [UP]
li	1	2	2
addi	2	2	4
subi	2	2	4
muli	2	2	4
divi	1	2	2
remi	1	2	2
neg	1	3	3

3.3 Testni program za ukaz JEQ:

```
main:  li    r0, 50          # Load 50 to r0
       sw    r0, 200         # Store value of r0 to address 200
       lw    r1, 200         # Read value from address 200 into r1
loop:  jeq   r1, r0, loop    # Compare values of r1 and r0, if they are
                             # equal jump to address of loop
```

Ta program naloži v register r0 vrednost 50, vrednost iz registra shrani na pomnilniški naslov 200, vrednost iz naslova 200 naloži v register r1, nato pa v zanki z ukazom JEQ preverja, če sta vrednosti registrov r0 in r1 enaki. Ker seveda sta enaki, s tem simuliram neskončno zanko.

Ukaz	Št. ponovitev	Trajanje [UP]	Skupno [UP]
li	1	2	2
sw	1	2	2
lw	1	2	2
jeq	1	2	2

3.4 Testni program za ukaze MOVE, NEG, EQ:

```
main:  li    r0, 50          # Load 50 to r0
       li    r1, 25          # Load 25 to r1
       move  r1, r0          # Move value from r0 to r1
       neg   r1              # Negate r1
       clr   r1              # Clear r1
```

Ta program naloži v register r0 vrednost 50, v register r1 pa 25. Nato vrednost iz registra r0 naloži v register r1, vrednost v registru r1 negira, nato pa ta register tudi izprazni.

Ukaz	Št. ponovitev	Trajanje [UP]	Skupno [UP]
li	2	2	4
move	1	2	2
neg	1	3	3
clr	1	4	4

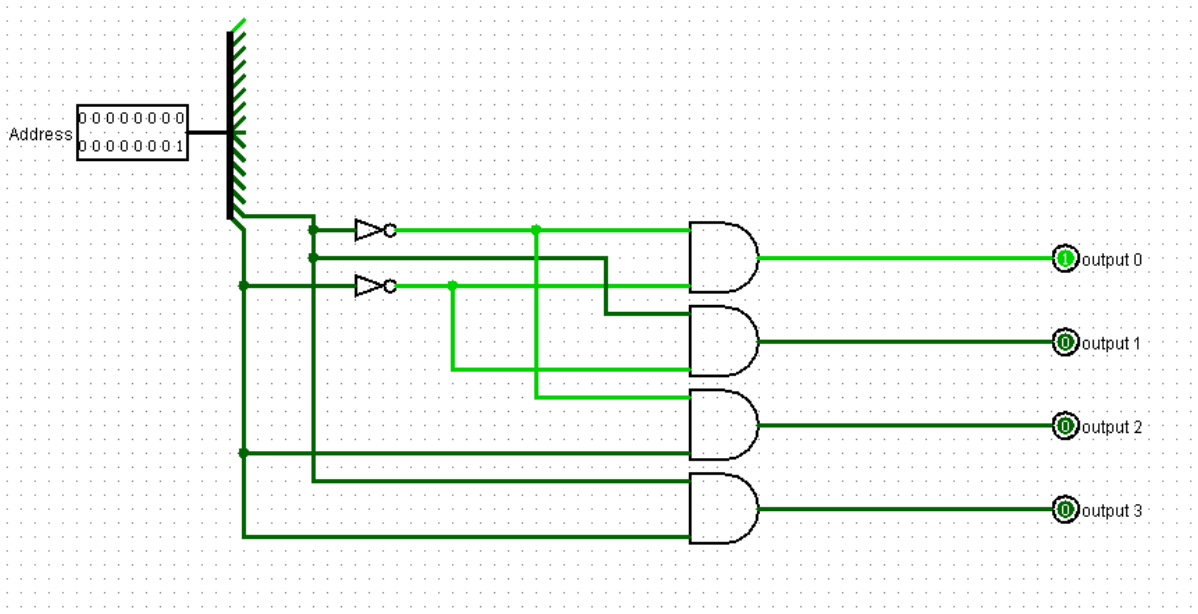
4. NALOGA:

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.

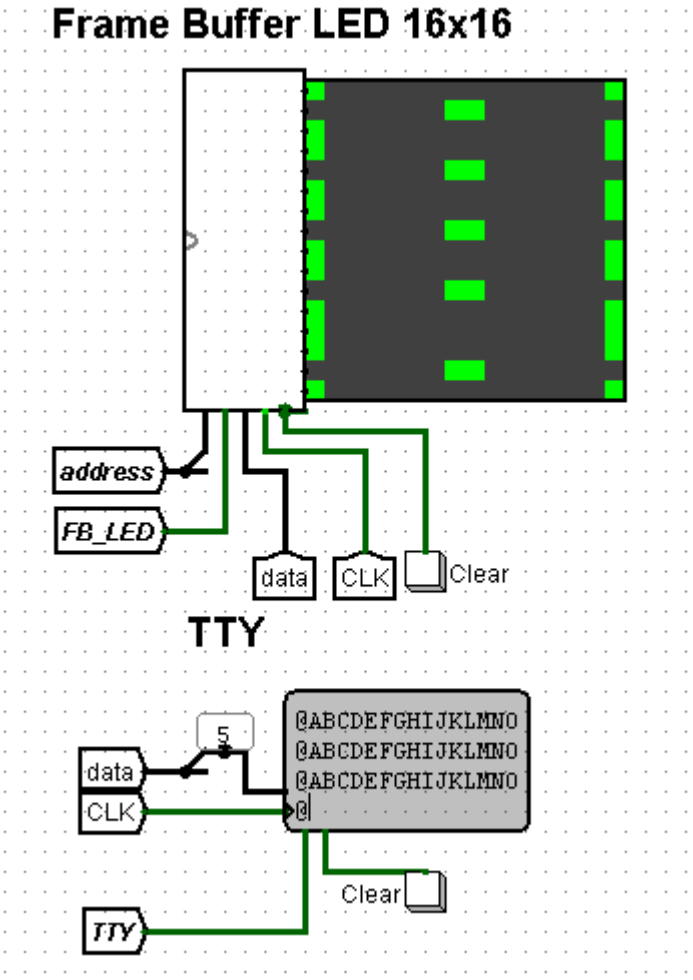
To nalogo sem rešil tako, da sem modificiral podvezje Address Decoder v MiMo CPU modelu. To vezje sem realiziral tako, da sem na vhodni pin velikosti 16 priklopil razdelilec, s katerim sem iz naslova ekstrahiral zadnja 2 bita (bita z največjo vrednostjo). Ta bita sem nato negiral, nato pa sem te povezave povezal na vhod AND vrat, katera so nato izbirala pravilni output, glede na vhode.

Ko sem popravil to vezje, sem prilagodil še testni program, da se je izrisoval nek vzorec. Testni program se nahaja v direktoriju *distribucija/test_program_IO_v05*.

Celotno vezje Address Decoder:



Rezultat delujočega vezja (na FB 16x16 se izrisuje vzorec, ki sem ga definiral sam):

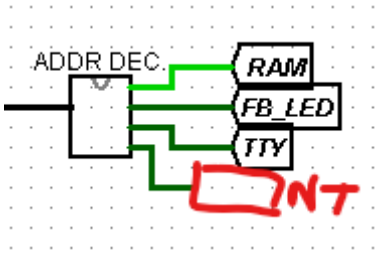


5. NALOGA:

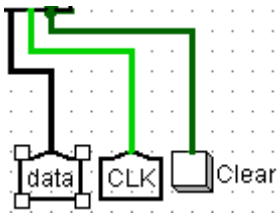
V MiMo modelu je pripravljen prostor za še eno vhodno izhodno napravo. Opišite vsa potrebna opravila za **pravilno vključitev dodatne vhodno izhodne naprave v MiMo model.** Kaj vse bi morali spremeniti in kako bi dosegli, da bi lahko z napravo tudi upravljali iz programov v zbirniku. Opišite karseda natančno in konkretno ter povezavo naprave tudi realizirajte in po možnosti preizkusite.

Da priključimo še eno vhodno/izhodno napravo, moramo vezju dodati kar nekaj component, in sicer:

- V vezje je treba vstaviti novo vhodno/izhodno napravo.
- Pri vezju Address Decoder je potrebno ustvariti novo komponento tunnel (na sliki označena z NT) in jo povezati z Address Decoder-jem.



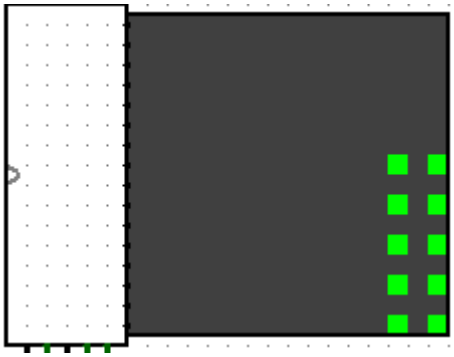
- Na napravo je treba povezati novoustvarjeno tunnel komponento.
- Na napravo je treba povezati tunnel ure (CLK).
- Na napravo je treba povezati tunnel data za pridobivanje podatkov.
- Na napravo je treba povezati gumb Clear.



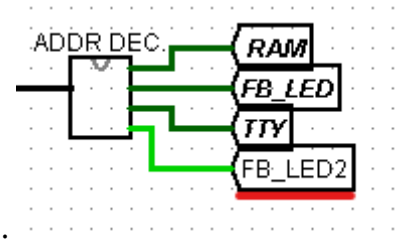
- Ostale povezave so stvar naprave, ki jo povezujemo.
- Ko pišemo kodo za napravo, moramo paziti, da je naslovni prostor naprave med $2^{15} + 2^{14}$ (49152) in $2^{16} - 1$ (65535).

5.1 Dejanska izvedba postopka – dodajanje nove naprave FB 16x16:

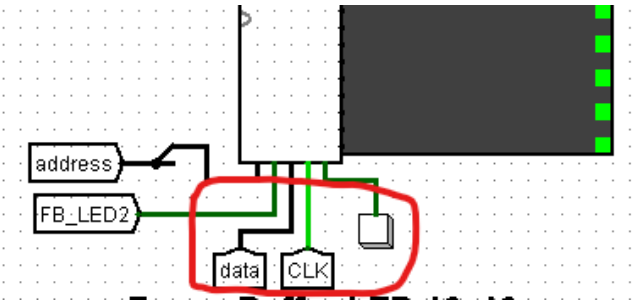
Postopek sem začel tako, da sem v vezje dodal novo napravo FB 16x16, saj se mi je iz možnih naprav zdela še najbolj zanimiva. To napravo sem dodal tako, da sem kombiniral LED matrix velikosti 16x16 z komponento Frame Buffer 16x16.



Nato sem pri vezju Address Decoder dodal novo komponento tunnel, jo poimenoval FB_LED2 in jo povezal z dekoderjem. Prav tako sem pri novi napravi ustvaril enako komponento in jo povezal z le to. Tako sem realiziral povezavo med dekoderjem in vhodno/izhodno napravo.



Nato sem pri napravi kreiral komponenti tunnel data in tunnel CLK (clock) ter komponento gumb, ki se uporablja za brisanje zaslona.



Na koncu sem le še prilagodil datoteko *test_program_IO_v05.s*, v kateri sem napisal par vrstic kode, ki so poskrbele za prikazovanje na zaslon. Tako sem tudi testiral, če je bil moj ocenjen naslovni prostor pravi.