UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Matej Gutman

Izvedba nevronske mreže s programirljivimi vezji FPGA

diplomsko delo na univerzitetnem študiju

mentor: doc. dr. Uroš Lotrič

Ljubljana, 2009

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01568/2009
Datum: 05.04.2009

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MATEJ GUTMAN**

Naslov: **IZVEDBA NEVRONSKE MREŽE S PROGRAMIRLJIVIMI VEZJI FPGA**

IMPLEMENTATION OF NEURAL NETWORK USING FPGA PROGRAMMABLE CIRCUITS

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Na programirljivih vezjih FPGA izdelajte nevronsko mrežo večplastni perceptron. Nevronsko mrežo zasnujte modularno in dovolj splošno, da se bo sposobna učiti iz predstavljenih parov vhodno – izhodnih vzorcev. Prednosti predlaganega pristopa predstavite na primeru razpoznavanja vzorcev.

Mentor:
doc. dr. Uroš Lotrič

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Franc Solina

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Matej Gutman,

z vpisno številko 63010036,

sem avtor diplomskega dela z naslovom

Izvedba nevronske mreže s programirljivimi vezji FPGA.

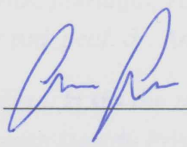
S svojim podpisom zagotavljam:

- da sem diplomsko delo izdelal samostojno pod mentorstvom (naziv, ime in priimek)

  doc. dr. Uroša Lotriča,

- da so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela in

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 2.10.2009 Podpis avtorja: 

Na prvem mestu bi se rad zahvalil mentorju, doc. dr. Urošu Lotriču, ki mi je pomagal kadarkoli sem pri diplomi potreboval pomoč. Zahvala gre tudi doc. dr. Patriciu Buliću, ki mi je s svojimi napotki glede jezika VHDL pomagal, ko se mi je zataknilo. Prav tako bi se zahvalil doc. dr. Branku Šteru, kakor tudi prof. dr. Andreju Dobnikarju za številne ideje.

Rad bi se zahvalil tudi mami in očetu, ki sta me tekom študija podpirala in mi študij omogočila. Zahvala velja tudi kolegoma Davidu Pristovniku ter Roku Skutniku, ki sta si vzela čas in pregledala diplomsko delo za tiskarske napake.

Posebna zahvala pa gre moji punci Andreji, ki mi je vsa ta leta študija stala ob strani ter me spodbujala.

Povzetek

Za živa bitja je značilna velika sposobnost učenja, ki jo raziskovalci že nekaj časa poskušajo prenesti tudi na stroje. Med modele, ki bolj ali manj uspešno posnemajo delovanje živčnega sistema, predvsem možganov živih bitij, sodijo nevronske mreže. Eden najbolj uporabljenih modelov nevronskih mrež je večplastni perceptron, pri katerem je učenje zaradi vzvratnega postopka popravljanja prostih parametrov dokaj hitro in učinkovito.

Tako za večplastni perceptron, kot tudi za ostale tipe nevronskih mrež, je značilna visoko paralelna struktura. Zato se postavlja vprašanje, ali so današnje arhitekture računalniških sistemov res primerne za izvedbo takih modelov. V diplomskem delu poskušamo razviti novo arhitekturo, pravzaprav čip, ki bi v največji možni meri izkoriščal paralelno naravo nevronskih mrež.

Primerno tehnologijo za razvoj novih arhitektur predstavljajo čipi FPGA, ki jih programiramo z jeziki za opisovanje strojne opreme, med katere sodi tudi jezik VHDL. Pri snovanju arhitekture smo se močno oprli na orodje Matlab, v katerem smo simulirali vse omejitve, ki nam jih vsiljuje tehnologija čipov FPGA. Na podlagi simulacije smo se odločili, da je novo arhitekturo smiselno zasnovati okrog visoko paralelne aritmetično logične enote, ki je v eni urini periodi sposobna izračunati izhod iz posameznega nevrona. V bližnji prihodnosti, ko bodo programirljivi čipi še zmogljivejši, bo smiselno razmišljati o izvedbi z več aritmetično logičnimi enotami, ki bodo delovale paralelno.

Obnašanje nevronske mreže v čipu FPGA pri aplikaciji razpoznavanja vzorcev potrjuje, da je izvedba uspela. S tem se odpirajo mnoge možnosti za nadaljnje raziskave, predvsem na področju razširitve ter optimizacije obstoječe arhitekture.

Ključne besede:

- nevronske mreže,
- jezik VHDL,
- čip Xilinx Spartan FPGA,
- razpoznavanje znakov

Abstract

Implementation of neural network using FPGA programmable circuits

Living creatures pose amazing ability to learn and adapt, therefore researchers are trying to apply this ability to machines. There are many mathematical models that mimic the behaviour of the central neural system, especially the brain, with neural networks being one of them. One of the most widely used neural networks is a multilayer perceptron, which gained its popularity with discovery of the back propagation learning algorithm.

A high degree of parallelism is inherently present in all types of neural networks. Since computers are not inherently parallel, the question arises, whether the existing architectures are appropriate for the implementation of such structures. Therefore, in the presented work we are trying to develop a chip with a new architecture, capable of vastly exploiting a parallelism present in the multilayered perceptron.

Programmable devices based on the FPGA technology enable us to quickly and efficiently develop and test new architectures. The behaviour of these devices can be specified in design-entry languages like the VHDL. In the developing cycle we have heavily relied on the Matlab simulations that enabled us to quickly solve restrictions and limitations posed by the FPGA technology. Following the successful completion of these simulations, a decision was made to create a powerful arithmetic logic unit, which is able to process a neuron in only one clock cycle. Perhaps in the future, when

the density of the gates in the FPGA devices becomes higher, a number of parallel arithmetic logic units might be applied.

The practical application in the field of character recognition confirms the suitability of the proposed architecture for the target FPGA devices. There are many possibilities for future work, especially in the area of optimization and expansion of the presented architecture.

Keywords:

- Neural networks,
- Design entry language VHDL,
- Xilinx Spartan FPGA device,
- Character recognition

Kazalo

1.

Uvod

Živa bitja imajo sposobnost prilagajanja okolici, predvsem pa so sposobna učenja, kar jim omogoča in daje prednost pred ostalimi pri boju za preživetje. Ljudje so leta preučevali, kako se bitja prilagajajo in učijo. Če je staro egipčansko ljudstvo verjelo, da leži duša in esenca človeka v srcu, je moderna znanost dognala, da v resnici leži v možganih. Kako pomemben in zapleten organ so možgani kaže tudi dejstvo, da je med letoma 1901 in 1991 prejelo nagrado na področju fiziologije in medicine približno deset odstotkov tistih, ki so prispevali k boljšemu razumevanju delovanja centralnega žičnega sistema [1]. Prav nič ne pretiravamo, če rečemo, da smo se v zadnjih petdesetih letih o delovanju možganov naučili več kot kdajkoli prej.

Centralni živčni sistem živih bitij ima poleg drugih pomembnih lastnosti tudi to, da se lahko dinamično uči zapletenih korelacij med vhodnimi in izhodnimi podatki glede na primere, ki mu jih podamo. To lastnost so poskušali posnemati tudi z matematičnimi modeli. Področje modeliranja živčnih sistemov živih bitij se je začelo razvijati potem, ko sta McCulloch in Pitts leta 1943 [1] predstavila prvi model umetnega nevrona. Področje je doživelo preporod šele po letu 1986, ko so Rumelhart, Hinton in Williams odkrili algoritem za vzvratno učenje umetnih nevronskih mrež [1].

Nevronske mreže so matematični modeli z izrazito paralelno strukturo, ki pa ji današnji računalniki v večini primerov, kljub tehnikam kot so cevovodi, super-skalarno procesiranje, večjedrnost in podobne, ne morejo popolnoma slediti. Kakšna bi bila torej optimalna arhitektura za implementacijo umetne nevronske mreže? Na voljo imamo

precej možnosti – od večjedrnega procesorja, do tiskanega vezja z več procesorji, ki jih povežemo med seboj preko vodila, do procesorjev DSP (*ang*. Digital Signal Processor). Ena od možnosti je tudi zasnova in izvedba modelov nevronskih mrež na posebnih vezjih, katerih arhitektura je v kar največji meri prilagojena potrebam po paralelnosti in načinu procesiranja, značilnih za nevronske mreže.

Področje sinteze nevronskih mrež na čipu je še precej novo. Veliko poskusov na tem področju je bilo opravljenih med letoma 1980 in 1990, vendar se ocenjuje, da so bili večinoma neuspešni, saj ni prišlo do širše uporabe nevro-računalnikov [2]. Velik del neuspeha pripisujejo dejstvu, da je bila večina teh nevro-računalnikov razvitih v tehnologiji ASIC (*ang*. Application Specific Integrated Circuit), katerih razvoj se takrat ni mogel kosati z razvojem procesorjev. Poleg tega tudi sorazmerno velika poraba silicija za tako vezje ni upravičevala njihove funkcionalnosti. Prav tako je potrebno vzeti v zakup, da tedanja programirljiva vezja tehnološko niso bila kos zahtevam, ki jih morajo izpolnjevati modeli nevronskih mrež, da postanejo tudi praktično uporabni. S prihodom vse bolj zmogljivih integriranih vezij FPGA (*ang*. Field Programmable Gate Arrays) ali čipov FPGA je področje znova zaživelo in sledilo je kar nekaj teoretičnih in praktičnih poskusov implementacij [2].

Posebne arhitekture je danes najhitreje in tudi najceneje razvijati na čipih FPGA (*ang*. Field Programmable Gate Array). V literaturi lahko zasledimo le malo idej za izvedbo posebnih arhitektur, ki bi bile prilagojene modelom nevronskih mrež, še manj pa je takih, ki imajo vgrajen algoritem učenja. V zadnjem času je moč opaziti kar nekaj premikov na tem področju [2]. Vsekakor lahko pričakujemo zanimive rešitve, saj obstaja veliko aplikacij, kjer se tak čip lahko uporabi, na primer na področju prepoznavanja objektov v sliki.

V nadaljevanju bodo najprej predstavljeni programirljivi čipi FPGA z vsemi prednostmi in omejitvami, ki jih moramo upoštevati pri razvoju posebne arhitekture. V tretjem poglavju si bomo ogledali večplastni perceptron, eno od najpogosteje uporabljenih nevronskih mrež, s posebnim poudarkom na prilagoditvi modela za računanje v celoštevilčni aritmetiki, ki jo zahteva uporabljena tehnologija. V četrtem poglavju, ki predstavlja osrednji del diplomske naloge, bo prikazana modularna zasnova posebne arhitekture za nevronsko mrežo, ki vključuje tako procesiranje kot tudi učenje. Nato bomo v petem poglavju prikazali pravilnost delovanja zasnovane arhitekture na primeru razpoznavanja znakov. Nazadnje bomo v sklepu povzeli vse najpomembnejše ugotovitve in ideje ter podali napotke za nadaljnje delo.

2.

Programirljiva vezja

Prvi programirljivi čipi so prišli na trg v sedemdesetih letih prejšnjega stoletja in so počasi pridobivali na svoji popularnosti, sedaj pa je to ena najhitreje rastočih vej v polprevodniški industriji [3]. Konec sedemdesetih so bila tiskana vezja polna čipov, ki so izvajala logične funkcije. Zato je podjetje Sinergetics, izdelalo čipe PLA (*ang*. Programmable Logic Array), ki so imeli dve programirljivi ravnini sestavljeni iz vrat AND in vrat OR. V tistem času je podjetje MMI izdelalo čip PAL (*ang*. Programmable Array Logic), pri katerem je bila programirljiva samo ravnina z vrati AND. Sledila so vezja SPLD (*ang.* Simple Programmable Logic Devices), ki so imela mrežo vertikalnih in horizontalnih povezav. Na vozlišču vsake povezave se je nahajala varovalka; z uničenjem le-te na posebnem programatorju je bila povezava v vozlišču prekinjena. Kasneje so vezjem dodali še sinhrone pomnilne celice (*ang*. flip-flop), kar je razširilo področje uporabe iz odločitvenih na popolnoma splošna sekvenčna vezja. Razvoj se je nadaljeval in na tržišče so prispela vezja CPLD (*ang.* Complex Programmable Logic Device), sestavljena iz več vezij SPLD.

Leta 1985 so v podjetju Xilinx izdelali čip, ki so ga poimenovali FPGA (*ang.* Field Programmable Gate Array). Sestavljen je iz blokov logičnih vrat, ki jih uporabnik lahko programira. Pri čipih FPGA so povezave, logične funkcije in podobno, shranjene v celicah SRAM (*ang.* Static Random Access Memory), zato je takšno konfiguracijo mogoče vsakič znova spremeniti. Ker celice SRAM ob izklopu napajanja izgubijo vso vsebino, mora čip FPGA ob vsakem vklopu ponovno naložiti ustrezno konfiguracijo ki je

običajno shranjena v zunanjem pomnilniku tipa FLASH ali EEPROM. Najnovejši čipi, med njimi je tudi družina Xilinx Spartan 3AN, tak pomnilnik že vsebujejo.

Trenutno podjetje Xilnx ponuja cenovno ugodno družino čipov Spartan, za zahtevnejše aplikacije pa so primernejši zmogljivejši a dražji čipi iz družine Virtex. Seveda Xilinx ni edini proizvajalec čipov FPGA. Trenutno je na trgu prisotnih več kot pet podjetij, ki se ukvarjajo z izdelavo in trženjem čipov FPGA. Med njimi so najprepoznavnejša Xilinx, Altera, Lattice, Actel in QuickLogic. V diplomskem delu smo se predvsem zaradi ugodne cene in brezplačnih programskih orodij ter dobre podpore v obliki dokumentacije na njihovi spletni strani odločili za čipe iz družine Xilinx Spartan.

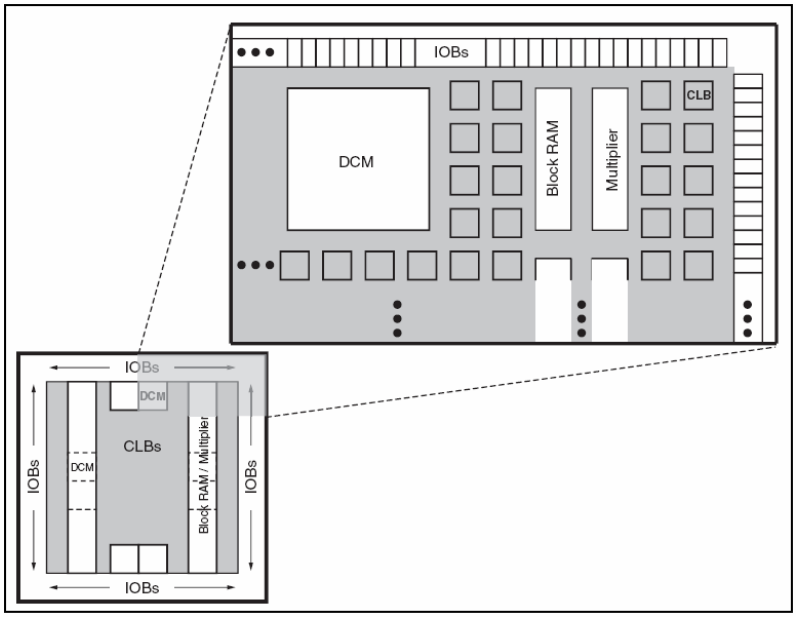
2.1. Zgradba vezja Xilinx Spartan

Vezja družine Xilinx Spartan so sestavljena iz pet glavnih skupin elementov [4]:

- konfigurabilnih logičnih blokov CLB (*ang*. Configurable Logic Blocks), ki vsebujejo programirljive tabele LUT (*ang*. Lookup Table), tabele in elemente za pomnjenje, ki se lahko uporabijo kot zatiči ali sinhrone pomnilne celice,
- vhodno-izhodnih blokov IOB (*ang*. Input/Output Blocks), preko katerih poteka prenos podatkov med notranjim delom čipa in vhodno-izhodnimi priključki ter omogočajo prenos podatkov v obe smeri, kakor tudi visoko impedančno stanje ter priklop pomnilnika DDR,
- dvo-kanalnega blokovnega pomnilnika RAM, imenovanega BRAM (*ang.* Block Random Access Memory) za shranjevanje podatkov z dvema neodvisnima naslovnima in podatkovnima vodilima, zaradi katerih je možno hkratno pisanje in branje podatkov,
- množilnikov (*ang*. Multipliers), ki sprejmejo dve 18 bitni števili ter jih zmnožijo v 36-bitni zmnožek in
- vezja za upravljanje z uro DCM (*ang.* Digital Clock Manager), ki skrbi za pripravo in distribucijo urinega signala, za zakasnitev ter fazni zamik ure, za množenje in deljenje urinega signala ter odpravljanje zdrsa ure (*ang*. clock skew).

Na sliki 2.1 je prikazana organizacija zgoraj naštetih elementov na silicijevi rezini. Obroč vhodno-izhodnih blokov IOB obdaja vse ostale elemente. Vsak blokovni RAM vsebuje več blokov velikosti 18 kbit, tik ob njem pa je postavljen množilnik. Dve vezji za upravljanje z uro se nahajata na vrhnji strani silicijeve rezine, dve pa na spodnji strani.

Čipi z večjo kapaciteto imajo dodano še eno vezje za upravljanje z uro na levi in eno na desni strani silicijeve rezine. Družina Spartan 3AN ima na voljo veliko povezovalnih linij, ki povezujejo glavne skupine elementov med seboj, s prenašanjem signalov iz enega elementa do drugega. Vsakemu od teh glavnih elementov pripada stikalna matrika, ki omogoča pestro izbiro povezav in s tem optimalno povezuje elemente med seboj.

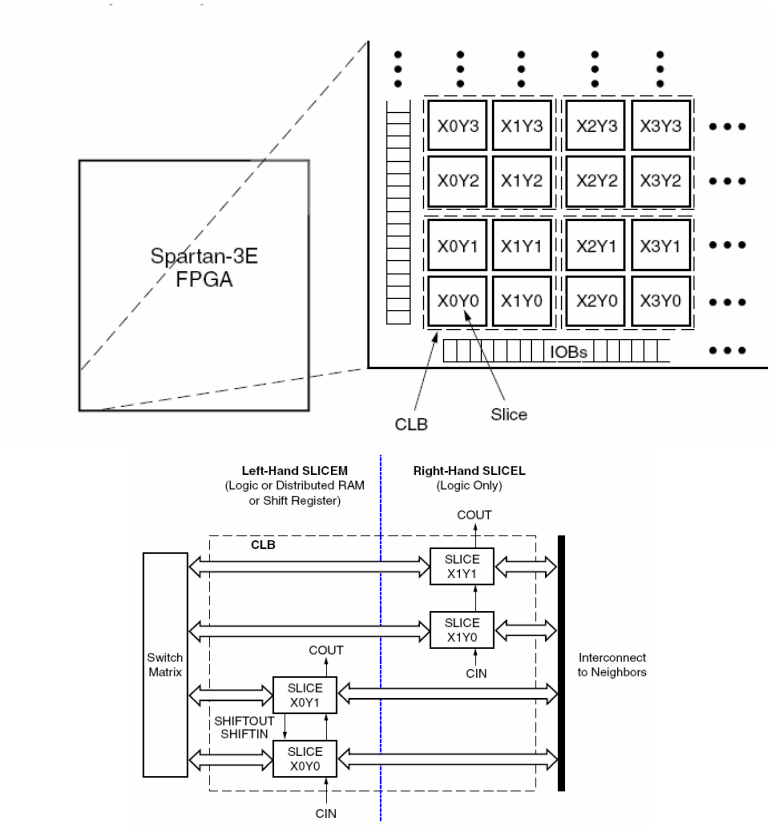


Slika 2.1: Zgradba čipa Xilinx Spartan.

2.1.1. Konfigurabilni logični bloki

Konfigurabilni logični bloki ali bloki CLB vsebujejo gradnike, ki so potrebni za izvedbo odločitvenih in pomnilnih logičnih funkcij v čipih FPGA. Lahko rečemo, da se bodo praktično vse logične funkcije, ki jih opišemo v nekem jeziku HDL (*ang.* High Definition Language) udejanjile v blokih CLB. Izjema so stavki, ki uporabijo druge gradnike, kot so preslikava v vhodno-izhodne bloke, ali pa uporaba vezja za upravljanje ure. Bloki CLB se nahajajo na sredini silicijeve rezine in so obdani z vhodno-izhodnimi bloki.

Kot smo videli na sliki 2.1 so bloki CLB razporejeni v obliki vrstic in stolpcev. Na izbranem čipu FPGA so vsi bloki CLB med seboj enaki. Vsak blok CLB je sestavljen iz štirih logičnih rezin (*ang*. slice) kot je to prikazano na sliki 2.2. Črki 'X' sledi številka, ki nam pove v katerem stolpcu se nahaja opazovana logična rezina, črki 'Y' pa sledi številka, ki predstavlja vrstico, v kateri se nahaja logična rezina na silicijevi rezini.

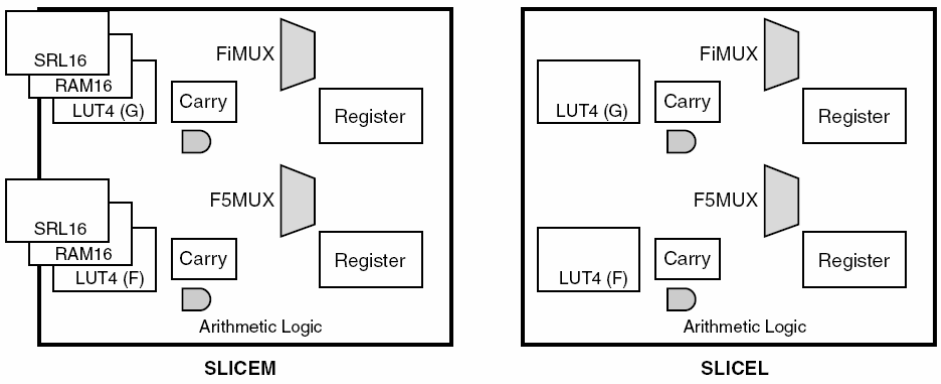


Slika 2.2 Zgradba blokov CLB v čipu Xilnx Spartan.

Logične rezine se med seboj razlikujejo, vendar so paroma enake. Enaki sta si tako levi kot tudi desni rezini. Levi par rezin nosi oznako SLICEM (*ang*. SLICE with Memory capability) desni pa SLICEL (*ang*. SLICE with Logic-only capability). Desni rezini sta zelo podobni levima, razlika je le v tem, da tabel LUT desnih rezin (SLICEL) ne moremo uporabiti za tvorbo pomnilnika RAM ali pomikalnih registrov. Na sliki 2.2 lahko opazimo, da sta levi rezini označeni z X0Y0 ter X0Y1, desni pa z X1Y0 ter X1Y1. Na ta način bomo od sedaj naprej ločevali logične rezine v bloku CLB. Razlika med logičnima rezinama SLICEL in SLICEM je lepo vidna na sliki 2.3 Kot vidimo na omenjeni sliki je vsaka logična rezina sestavljena iz:

dveh logičnih celic,

- dveh 4 vhodnih tabel LUT in tabel F-LUT in G-LUT,
- dveh elementov za hranjenje informacij,
- dveh izbiralnikov FiMUX in F5MUX, ki omogočata izbiro poti signalov ter tvorbo več vhodnih funkcij in
- aritmetične in prenosne logike.



Slika 2.3: Zgradba leve (SLICEM) in desne (SLICEL) logične celice v bloku CLB.

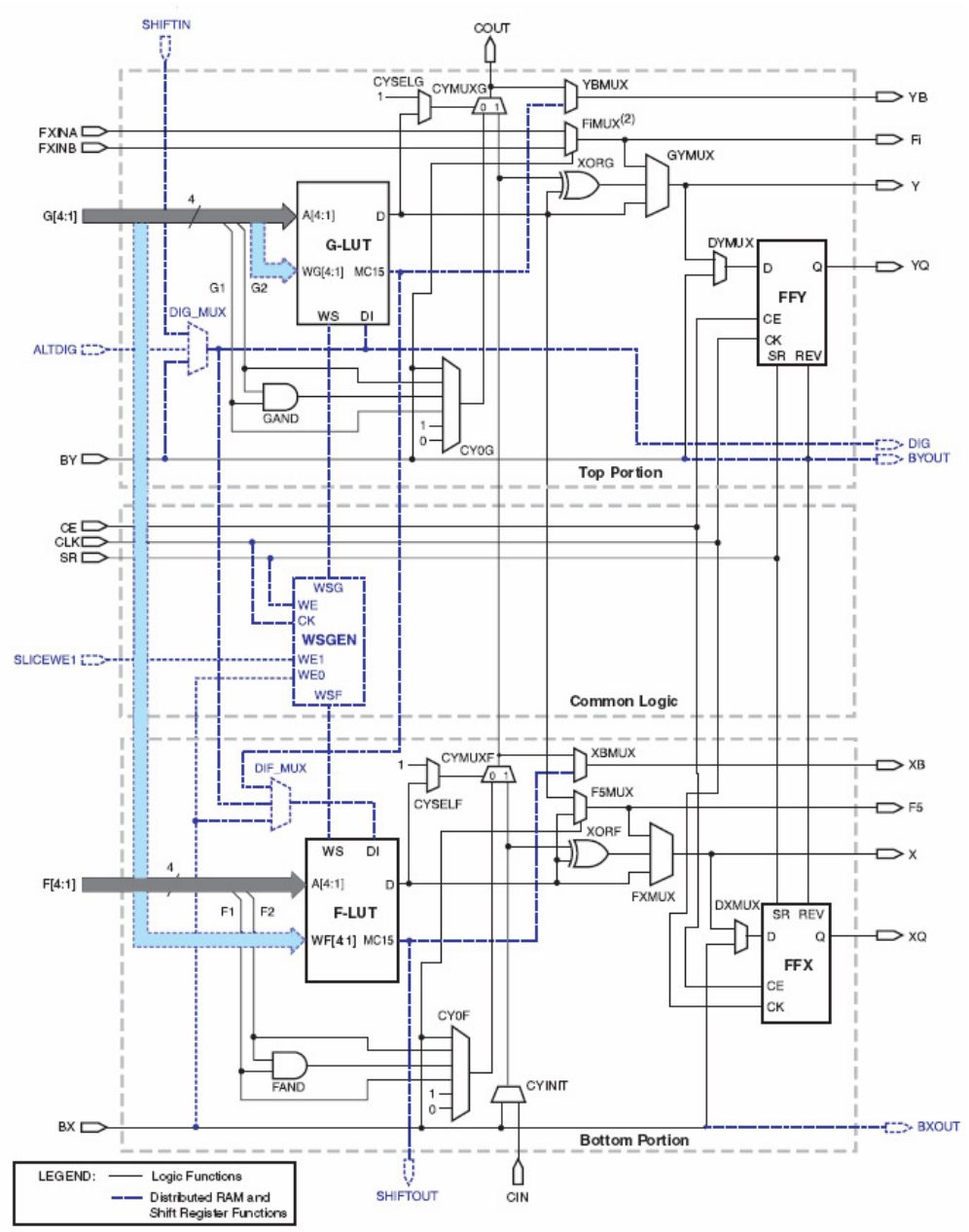
V logični rezini SLICEM lahko uporabimo še:

- dva 16x1 bloka RAM, imenovana RAM16 in
- dva 16 bitna pomikalna registra, imenovana SRL16.

Podrobnejša zgradba logične rezine SLICEM je podana na sliki 2.4. Ker sta logični rezini SLICEM in SLICEL praktično enaki, so z modro črtkano črto označeni elementi, ki so dodani logični rezini tipa SLICEL, da dobimo rezino tipa SLICEM. Vsaka logična rezina vsebuje dve polovici, ki jih imenujemo tudi logični celici, ti pa vsebujeta vse potrebne osnovne logične gradnike za implementacijo kombinatoričnega ali sinhronega logičnega vezja. Logična rezina na sliki 2.4 vsebuje dve polovici imenovani zgornji del (*ang*. top portion) in spodnji del (*ang*. bottom portion). Obe polovici sta med seboj povezani preko skupne povezovalne logike (*ang*. common logic) in imata skupne naslednje kontrolne signale [5]:

- urin signal CLK (*ang*. CLocK),
- signal CE (*ang*. Clock Enable),
- signal SLICEWE1, ki omogoča pisanje v pomnilnik RAM, kadar se v tabeli LUT izdela pomnilnik (na voljo samo v logični rezini tipa SLICEM), in
- signal SR (*ang*. Set/Reset) za spominski celici tipa D.

Tabeli LUT imata oznaki G-LUT v zgornji logični celici ter F-LUT v spodnji logični celici. Pomnilni celici D imata oznaki FFY v zgornji logični celici ter FFX v spodnji logični celici. Vsaka logična rezina ima dva izbiralnika (*ang*. multiplexors) za določanje poti signalov, poimenovana FiMUX v zgornji logični celici ter F5MUX v spodnji logični celici.



Slika 2.4: Zgradba logičnih rezin SLICEL in SLICEM.

Signali začnejo svojo pot pri vhodih F1 do F4. Do teh vhodov pridejo preko stikalnih matrik, ki povezujejo glavne elemente čipa FPGA. Najprej potujejo na vhod tabele LUT, kjer se tvori poljubna 4-vhodna logična funkcija. Pot nadaljujejo iz izhoda D tabele LUT po eni izmed petih možnih poti:

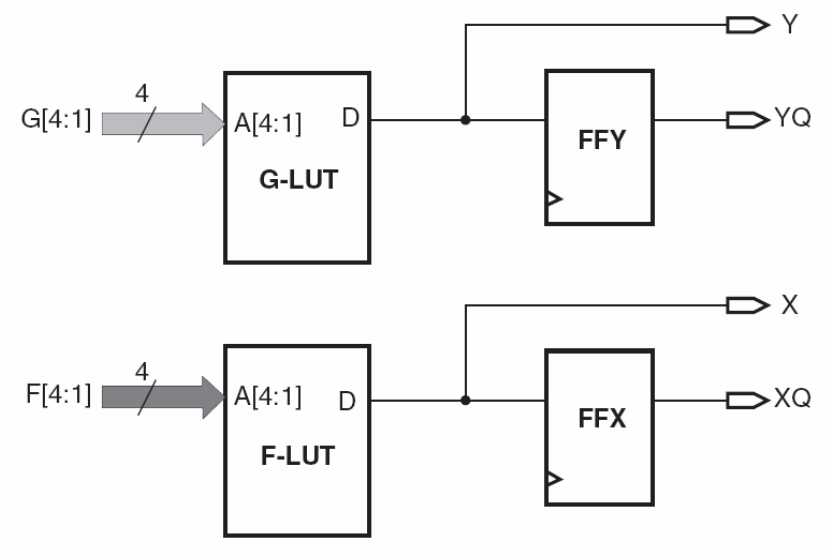
- skozi izbiralnik FXMUX na izhod X ven iz logične rezine,

- skozi izbiralnik FXMUX na izbiralnik DXMUX skozi pomnilno celico FFX na izhod XQ ven iz rezine,
- lahko ga uporabimo kot izbiralni vhod izbiralnika CXMUXF, ta izbiralnik pa se uporabi v aritmetični in prenosni logiki,
- lahko se uporabi kot vhod v vrata XORF, ki se prav tako uporabljajo v aritmetični in prenosni logiki ali
- kot vhod v izbiralnik F5MUX kjer lahko z njim tvorimo več kot 4-vhodne logične funkcije.

Poleg opisane glavne podatkovne poti, pri kateri signali vstopijo na vhodih F1 do F4, pa lahko signali v logično rezino vstopijo tudi na vhodih BX in BY, ter potujejo skozi rezino po naslednjih poteh:

- nespremenjeni lahko izstopijo na izhodu BXOUT,
- potujejo lahko v pomnilno celico FFX in izstopijo na izhodu XQ,
- se uporabijo kot izbiralni vhod za izbiralnik F5MUX ali
- se uporabijo kot vhod v aritmetični in prenosni logiki.

Tabele LUT čipa FPGA so glavni elementi, s katerimi se izvedejo vse logične funkcije v čipu. Pri logičnih rezinah SLICEM pa lahko tabele LUT služijo tudi kot porazdeljen pomnilnik RAM (*ang*. distributed RAM) ali kot 16-bitni pomikalni registri.



Slika 2.5: Tabeli LUT v logičnih rezinah.

Kot vidimo na sliki 2.5 ima vsaka tabela LUT štiri logične vhode (A1 – A4) in en izhod (D). S tabelami LUT lahko izvedemo katerokoli logično funkcijo s štirimi spremenljivkami. Če je potrebno izdelati funkcijo več spremenljivk, je možno tabele LUT vezati kaskadno. Kot je razvidno na slikah 2.4 in 2.5 je izhod tabele LUT povezan na izbiralnik, aritmetično in prenosno logiko, ali pa na enega od pomnilnih elementov FFX oziroma FFY.

2.1.2. Lastnosti vezij Xilinx Spartan

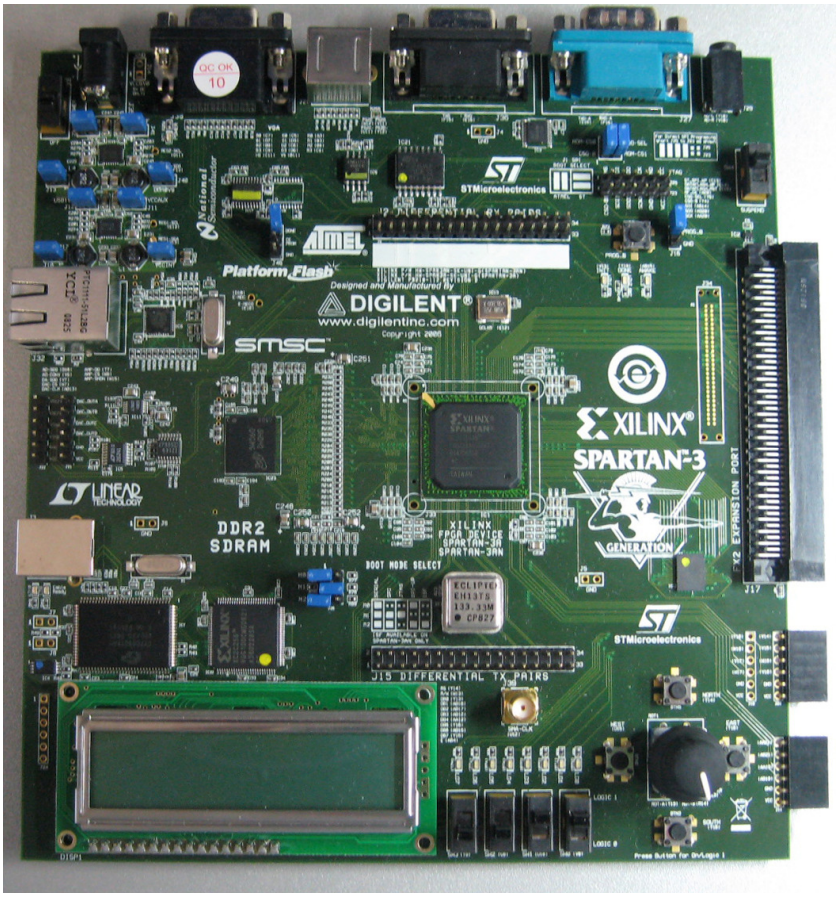
V tabeli 2.1 so prikazana vezja družine Spartan 3AN, z rumeno pa je poudarjena vrstica, ki opisuje izbrani čip. Kot bomo videli kasneje, sta za našo arhitekturo najbolj pomembna podatka o številu logičnih rezin ter o številu množilnikov. Če arhitektura potrebuje večje število množilnikov, kot je vgrajenih, jih je potrebno sintetizirati s pomočjo logičnih rezin, tako kot vso ostalo odločitveno in pomnilno logiko.

Tabela 2.1: Osnovne značilnosti čipov Xilinx Spartan.

Oznaka čipa	Število vrat	Ekvivalentnih logičnih celic	Vseh CLB	Vseh logičnih rezin	Št. bitov dist. RAM	Št. bitov blok RAM	Množilnikov (18x18)	DCM vezij
XC3S50AN	50.000	1.584	176	704	11k	54k	3	2
XC3S200AN	200.000	4.032	448	1.792	28k	288k	16	4
XC3S400AN	400.000	8.064	896	3.854	56k	360k	20	4
XC3S700AN	700.000	13.248	1472	5.888	92k	360k	20	8
XC3S1400AN	1.400.000	25.344	2816	11.264	176k	576k	32	8

2.2. Razvojna plošča

Za testiranje predlagane arhitekture smo uporabili razvojno ploščo Spartan 3AN starter Kit, ki je predstavljena na sliki 2.6.



Slika 2.6: Razvojna plošča Xilinx Spartan 3AN starter kit.

Na razvojni plošči imamo na voljo [6]:

- čip FPGA XC3S700AN,
- 4 Mbit pomnilnika Xilinx FLASH, namenjenega shranjevanju konfiguracije,
- 64 MB pomnilnika DDR2 (v organizaciji 32x16),
- 4 MB pomnilnika NOR FLASH,
- 2 x 16 Mbit serijski pomnilnik FLASH, ki podpira serijski protokola (SPI),
- dvovrstični prikazovalnik LCD,
- vrata PS/2 za priklop tipkovnice ali miške,
- vrata VGA,
- čip PHY za podporo omrežju Ethernet 10/100 Mbit,

- dva serijska vmesnika RS232,
- priključek USB namenjen prenosu bitne datoteke na vezje FPGA in razhroščevanju,
- 50 MHz oscilator,
- 8 pinsko podnožje DIP za zunanji oscilator,
- priključek SMA, ki lahko služi kot vhod za urin takt ali pa kot izhod,
- 100 polni priključek Hirose ,
- visokoprepustne diferencialne priključke,
- 4-kanalni digitalno analogni pretvornik,
- 2-kanalni analogno-digitalni pretvornik,
- rotacijski kodirnik,
- 8 diod LED,
- 4 drsne gumbe in
- 4 gumbe na stik.

2.3. Orodje za načrtovanje vezij FPGA

Izbrani čip Spartan 3AN je popolnoma podprto v brezplačnem integriranem programskem okolju Xilinx ISE Webpack (*ang*. Integrated Software Enviroment). Eden od pogledov na programsko okolje je predstavljen na sliki 2.7.

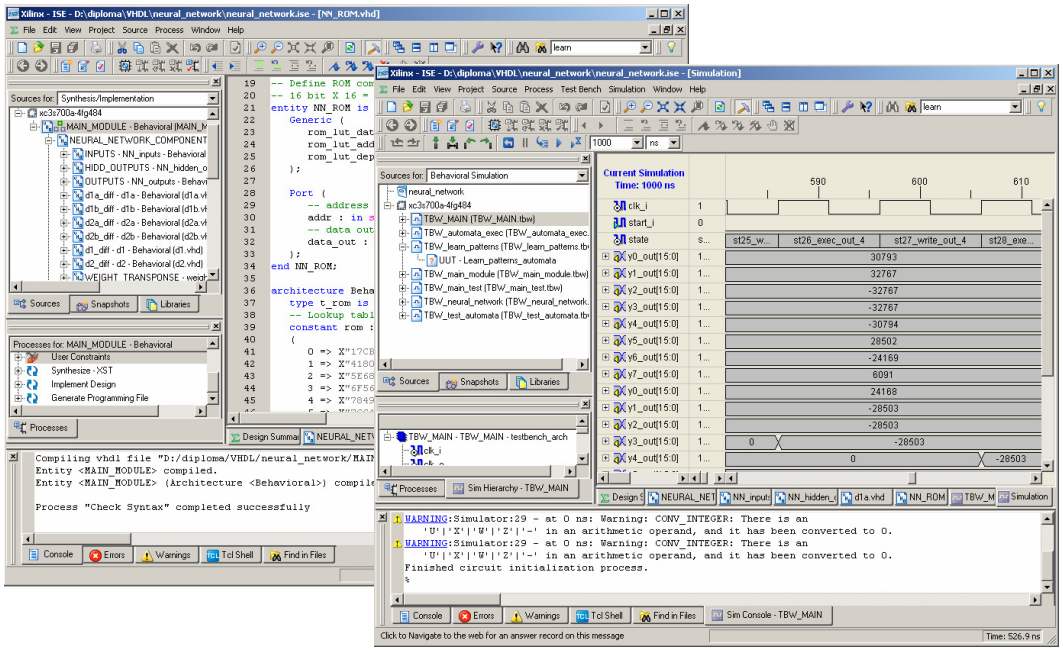
Delo v programskem okolju Xilinx ISE Webpack poteka v naslednjem zaporedju:

- opis vezja: Vezje opišemo v poljubnem programskem jeziku (Verilog, VHDL) ali pa ga narišemo v grafičnem urejevalniku (schematic editor). Rezultat prevajanja na tem koraku je datoteka EDIF (*ang*. Electronic Data Interchange Format), ki ustreza industrijskim standardom in jo je možno kasneje uporabiti v poljubnem programskem paketu, ki podpira tak standard. Do te točke je opis vezja neodvisen od tehnologije.
- izvedba vezja: Tu se vsebina datoteke EDIF preslika v izbrano tehnologijo, v našem primeru Spartan 3AN. Rezultat je datoteka NCD (*ang*. Native Circuit Description), ki vsebuje opis vezja z gradniki, ki so podprti v tehnologiji. Če smo uporabili kakšen vir, ki ne obstaja, nas na tem koraku orodje na to opozori, saj drugače ne moremo

nadaljevati. Rezultat tega koraka je konfiguracijska datoteka (bitstream), ki se lahko prenese in zažene na čipu FPGA s pomočjo programskega orodja, ki je del razvojnega okolja ISE Webpack. Implementirano datoteko si lahko v razvojnem okolju ISE Webpack ogledamo kot shemo RTL (*ang*. Register Transfer Level), ki grafično prikaže rezultat opisa v datoteki VHDL, ali pa kot tehnološko shemo (*ang*. Technology Schematic), v kateri si lahko ogledamo dejansko porabljene vire na čipu FPGA.

- verifikacija vezja: Na tej točki se uporabi simulator, ki deluje na nivoju logičnih vrat. S pomočjo simulatorja se preveri ali vezje ustreza časovnim zahtevam ter ali vezje res deluje tako kot je predvideno. Simulator ISIM, ki je vključen v programsko okolje ISE Webpack je brezplačen in za projekte z do 50.000 vrsticami kode popolnoma funkcionalen, pri večjih projektih se simulacije občutno upočasnijo.

- funkcionalna verifikacija vezja: Ta lahko poteka na že sprogramiranem čipu s pomočjo osciloskopa ali logičnega analizatorja. Pri tem smo precej omejeni, saj signalov, ki jih nismo usmerili na vhodno-izhodne bloke, na vezju FPGA ne moremo pomeriti.



Slika 2.7: Integrirano programsko okolje Xilinx ISE Webpack.

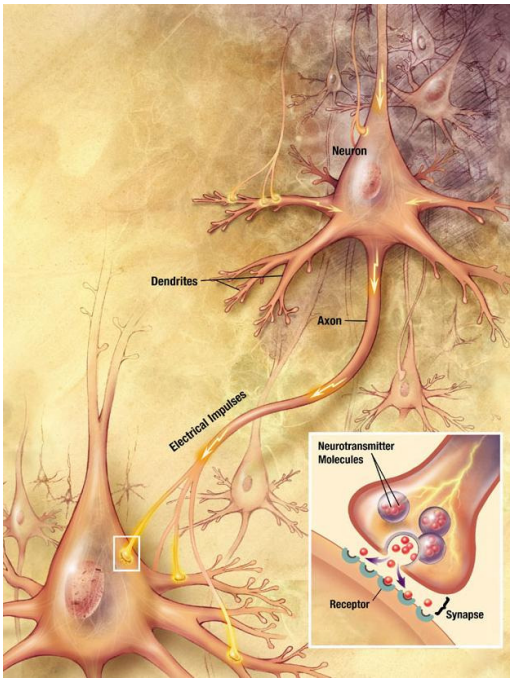
3.

Nevronske mreže

Z umetnimi nevronskimi mrežami poskušamo modelirati in oponašati delovanje bioloških nevronskih mrež. Da bi lahko uspešno postavili in analizirali model nevronske mreže, moramo poznati vsaj njene osnovne značilnosti. Čeprav so nevroni med seboj povezani z milijoni povezav in je bilo predstavljenih že veliko abstraktnih modelov [1], ki jih poskušajo opisati, pa je zgradba osnovne celice (nevrona) znana in splošno sprejeta.

Prvi, ki je prepoznal nevron kot osnovno funkcijsko enoto živčnega sistema je bil Cajal [7]. Nevron, kot je prikazan na sliki 3.1, sestoji iz jedra, dendritov, aksona in sinaptičnih povezav [7]. Dendriti nevrona sprejemajo informacije iz okolice ali drugih nevronov preko sinaptičnih povezav. Jedro nato v odvisnosti od stanja sinaptičnih povezav, kemičnih in električnih procesov signal bodisi ojača ali pa zaduši. Ta signal nato potuje po aksonu, ki se običajno razveji do več drugih nevronov.

Ocenjujejo, da je v možganski skorji človeka približno 10^9 nevronov, ki so med seboj povezani s 60 x 10^{12} sinaptičnimi povezavami. Za svoje delovanje naj bi porabili 10^{-10} J energije na sekundo [7]. Čeprav obdelava informacij v računalniških sistemih poteka v nanosekundah, v možganih pa v milisekundah, je ravno način in število povezav razlog, da so možgani dosti bolj uspešni pri opravljanju nalog, kot so iskanje vzorcev v sliki, razpoznavanje govora in vodenje. Centralni živčni sistem je tudi visoko redundanten - odrasel človek izgubi okoli 50.000 nevronov na dan, oziroma okoli 10 % nevronov do 75. leta starosti, pa še vedno brez večjih težav opravlja večino funkcij [8].



Slika 3.1: Shema biološkega nevrona.

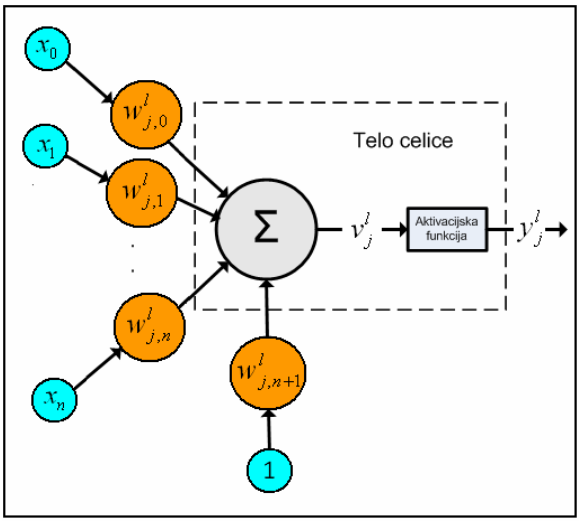
3.1. Matematični model nevrona

Z matematičnim modelom nevrona poskušamo zajeti najpomembnejše lastnosti živčne celice in s tem posnemati njeno delovanje. Enega izmed prvih modelov, ki je še danes najbolj uveljavljen, sta predstavila McCulloch in Pitts leta 1943 [7], katerega shema modela nevrona je predstavljena na sliki 3.2.

Funkcijo sinaptičnih povezav v modelu opravljajo uteži $w_{j,1}^l$, ... , $w_{j,n+1}^l$, ki vhode v nevron $x_1,...,x_{n+1}$ ustrezno utežijo. Pri označevanju uteži smo uporabili indeksa j in l, s katerima bo v nadaljevanju določen položaj nevrona v nevronski mreži. Pozitivno vrednost uteži si lahko predstavljamo kot vzbujeno sinaptično povezavo, ki vhodni signal ojači, negativno pa kot nevzbujeno sinaptično povezavo, ki vhodni signal zaduši. V jedru celice se, podobno kot pri biološki celici, uteženi signali najprej seštejejo v aktivacijski potencial

$$v_j^l = \sum_{i=1}^{n+1} w_{j,i}^l x_i \quad . \tag{3.1}$$

V modelu je dodan prag, s katerim določamo aktivno območje nevrona, v obliki uteži $w_{j,n+1}^l$, s fiksnim vhodom $x_{n+1} = 1$.

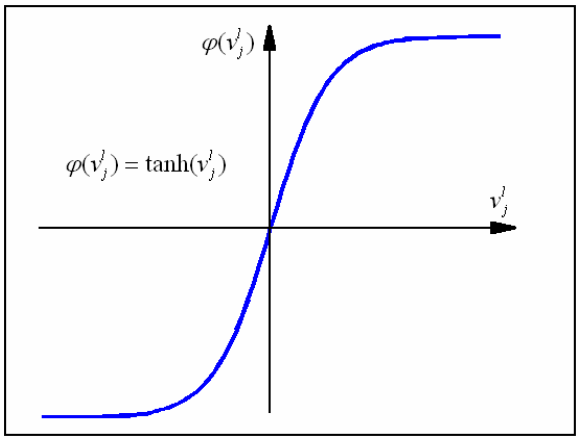


Slika 3.2: Matematični model nevrona.

Za aktivacijsko funkcijo običajno izberemo sigmoidno funkcijo. Gre za družino gladkih monotono naraščajočih funkcij v obliki črke S, ki imajo natanko en prevoj. V našem primeru smo se odločili za funkcijo

$$\varphi(v_j^l) = \tanh(v_j^l) = \frac{e^{v_j^l} - e^{-v_j^l}}{e^{v_j^l} + e^{-v_j^l}} \quad , \tag{3.2}$$

ki je predstavljena na sliki 3.3.



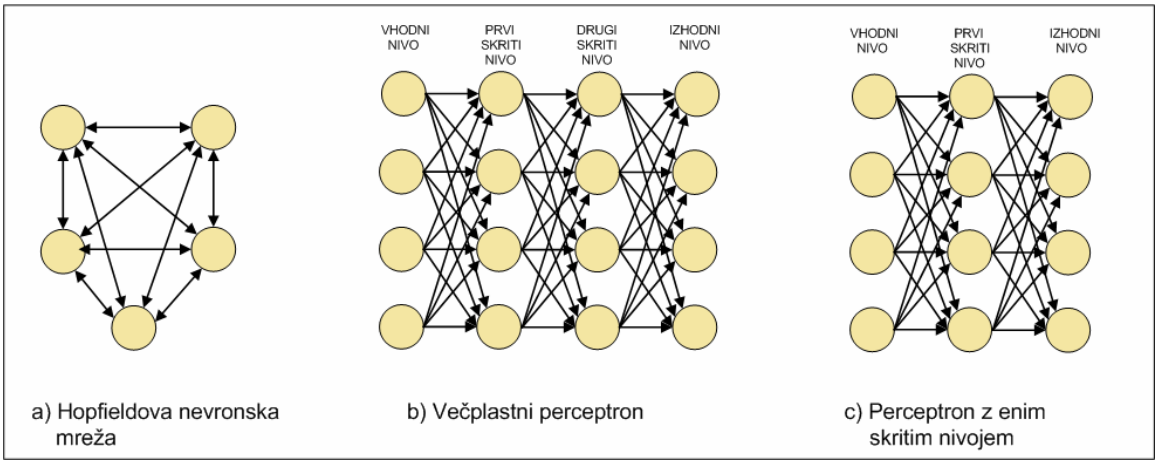
Slika 3.3: Aktivacijska funkcija.

Izhod nevrona je nazadnje določen z enačbo :

$$y_j^l = \varphi(v_j^l) = \tanh(\sum_{i=1}^{n+1} w_{j,i}^l x_i) \tag{3.3}$$

3.2. Modeli nevronskih mrež

Načinov, kako se nevroni povezujejo v mrežo, je veliko. Najbolj splošen, pa tudi najbolj kompleksen način povezave predstavlja Hopfieldova nevronska mreža [7], kjer je vsak nevron povezan z vsakim (slika 3.4a). Velik razcvet nevronskih mrež je sledil po letu 1986, ko so Rumelhart, Hinton in Williams predstavili večplastni perceptron (slika 3.4b) z vzvratnim postopkom prilagajanja prostih parametrov modela [7]. V nadaljevanju smo se omejili na večplastni perceptron (slika 3.4c) z enim skritim nivojem, ki je kljub majhnemu številu plasti še vedno predstavlja popolnoma splošno nevronsko mrežo.

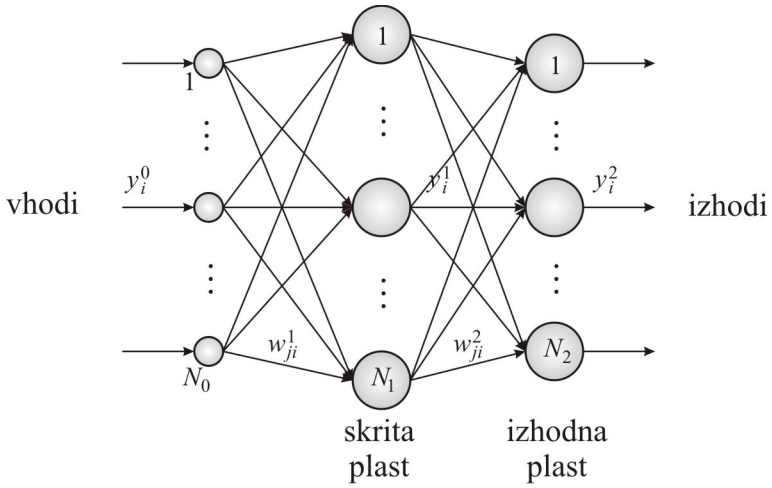


Slika 3.4: Nekaj tipov nevronskih mrež.

3.2.1. Večplastni perceptron

Večplastni perceptron z vsaj eno skrito plastjo nevronov in nelinearnimi aktivacijskimi funkcijami je univerzalni aproksimator, saj lahko z njim poljubno natančno aproksimiramo katerokoli funkcijo [7]. Zaradi nelinearnosti, vpetih v sam model, je večplastni perceptron zelo primeren tudi za modeliranje nelinearnih dinamičnih sistemov. Topologija večplastnega perceptrona omogoča, da se informacija med procesiranjem porazdeli na vse nevrone, ki so vključeni v model [7]. Ker prav vsak nevron v nekem trenutku nosi del informacije, napaka na enem nevronu ali povezavi ne povzroči hude napake na izhodu. Ta neobčutljivost modela na manjše napake je pomembna lastnost pri modeliranju realnih sistemov. Druga zelo pomembna lastnost modelov nevronskih mrež je posploševanje znanja, ki ga pridobijo med prilagajanjem prostih primerov, oziroma učenje iz primerov [7].

Večplastni perceptron na sliki 3.5 ima N_0 vhodov v nevronsko mrežo, N_1 nevronov v skriti plasti in N_2 nevronov v izhodni plasti. Tak model ima skupaj $(N_0+1)N_1 + (N_1+1)N_2$ uteži, ki določajo njegovo kapaciteto. Izhod večplastnega perceptrona z enim skritim nivojem, računamo plast za plastjo, od vhoda proti izhodu.



Slika 3.5: Večplastni perceptron z eno skrito plastjo nevronov.

Vzemimo, da vhod v večplastni perceptron predstavimo kot vektor $\mathbf{y}^0 = (y_1^0,..., y_{N_0}^0)^{\mathrm{T}}$, željeni izhod pa kot vektor $\mathbf{t} = (t_1,...,t_{N_2})^{\mathrm{T}}$. Izhodi skritega nivoja so potem enaki

$$y_j^1 = \varphi(v_j^1) \quad , \quad v_j^1 = \sum_{i=1}^{N_0+1} w_{ji}^1 y_i^0 \quad , \quad j = 1,..., N_1 \quad , \tag{3.4}$$

pri čemer smo upoštevali, da utež za vhod $y_{N_0+1}^0 = 1$ nadomešča prag. Na enak način lahko za izhode nevronov izhodne plasti zapišemo

$$y_j^2 = \varphi(v_j^2) \quad , \quad v_j^2 = \sum_{i=1}^{N_1+1} w_{ji}^2 y_i^1 \quad , \quad j = 1,..., N_2 \quad , \tag{3.5}$$

kjer smo postavili $y_{N_1+1}^1 = 1$. Za izhode iz modela, ki jih v obliki vektorja lahko zapišemo kot $\mathbf{y}^2 = (y_1^2,..., y_{N_2}^2)^{\mathrm{T}}$, si želimo, da bi bili kar najbolj podobni želenemu izhodnemu vektorju $\mathbf{t}$.

3.2.2. Učenje večplastnega perceptrona

Učenje je prilagajanje prostih parametrov nevronske mreže glede na vplive okolja [7]. Nadzorovano učenje zahteva od okolice ali učitelja, da za vsak vzorec, ki ga predstavi na vhod v nevronsko mrežo, ve kakšen je pravilen izhod. S postopkom učenja želimo

nevronsko mrežo naučiti tako, da bo na znani vzorec $\mathbf{y}^0$ odgovorila z izhodom iz modela $\mathbf{y}^2$, ki se bo kar najmanj razlikoval od želenega izhoda $\mathbf{t}$, oziroma, da bo napaka med istoležnimi elementi vektorjev $\mathbf{t}$ in $\mathbf{y}^2$,

$$e_j = t_j - y_j^2 \quad , \tag{3.6}$$

čim manjša. Najpogosteje se uspešnost modela nevronske mreže meri s cenilno funkcijo kvadratne napake [7],

$$E = \frac{1}{2}\sum_{j=1}^{N_2} e_j^2 \quad . \tag{3.7}$$

Sledi izračun gradientov po vzvratnemu postopku učenja [7]. Za uteži nevronov in za pragove v izhodni plasti lahko z uporabo verižnega pravila zapišemo:

$$\frac{\partial E}{\partial w_{ji}^O} = \frac{\partial E}{\partial v_j^2}\frac{\partial v_j^2}{\partial w_{ji}^2} = d_j^2 y_i^1 \quad , \tag{3.8}$$

kjer je

$$d_j^2 = \frac{\partial E}{\partial v_j^2} = \frac{\partial E}{\partial e_j}\frac{\partial e_j}{\partial y_j^2}\frac{y_j^2}{v_j^2} = -e_j \varphi'(v_j^2) \quad . \tag{3.9}$$

Na enak način nadaljujemo tudi na skriti plasti,

$$\frac{\partial E}{\partial w_{j,i}^1} = \frac{\partial E}{\partial v_j^1}\frac{\partial v_j^1}{\partial w_{j,i}^1} = d_j^1 y_i^0 \quad , \tag{3.10}$$

pri čemer velja še

$$d_j^1 = \frac{\partial E}{\partial v_j^1} = \left(\sum_{o=1}^{N_2}\frac{\partial E}{\partial v_o^2}\frac{\partial v_o^2}{\partial y_j^1}\right)\frac{\partial y_j^1}{\partial v_j^1} = \left(\sum_{o=1}^{N_2} d_j^2 w_{o,j}^2\right)\varphi'(v_j^1) \quad . \tag{3.11}$$

V zgornjih enačbah smo s $\varphi'(v) = \partial\varphi(v)/\partial v$ označili odvod aktivacijske funkcije po parametru v.

Potem, ko poznamo gradient napake na izhodu po vseh prostih parametrih modela ali utežeh, lahko izračunamo popravke uteži na izhodni plasti po enačbi

$$w_{j,i}^2 \leftarrow w_{j,i}^2 - \eta\frac{\partial E}{\partial w_{j,i}^2} \quad , \tag{3.12}$$

na skriti plasti pa po enačbi

$$w_{j,i}^{1} \leftarrow w_{j,i}^{1} - \eta \frac{\partial E}{\partial w_{j,i}^{1}} \quad . \tag{3.13}$$

3.3. Prilagoditev modela za strojno izvedbo

Da bi si olajšali načrtovanje modela nevronske mreže na čipu FPGA, smo delovanje modela najprej simulirali v okolju Matlab, pri čemer smo se omejili na računanje v celoštevilčni aritmetiki. Hkrati smo vse kompleksne funkcije, ki se pojavljajo v modelu ustrezno tabelirali.

Celotna izvorna programska koda, napisana za okolje Matlab, je predstavljena v Prilogi A. V nadaljevanju poglavja so v dveh ločenih podpoglavjih predstavljeni samo njeni ključni deli.

3.3.1. Računanje izhodov

Izvajanje modela večplastnega perceptrona poteka vedno od vhodne plasti preko skrite plasti na izhodno plast. Ker je večplastni perceptron sam po sebi zelo paralelna struktura, bi lahko v enem koraku obdelali celotno plast. Na žalost so čipi iz družine Spartan trenutno še premalo zmogljivi, da bi bila taka realizacija za smiselne modele večlastnega perceptrona sploh mogoča.

Kljub temu, da paralelno izvajanje celotne plasti zankrat še ne pride v poštev, pa lahko paralelno obdelamo vsak nevron v plasti posebej. To tudi odraža izvorna koda, predstavljena na sliki 3.6, ki za množenje in seštevanje uporablja množilnik *mult*. V kodi:

- konstanta *N1* predstavlja število nevronov na skritem nivoju,
- konstanta *N2* predstavlja število nevronov na izhodnem nivoju,
- *W1* in *W2* sta matriki, ki vsebujeta uteži vseh nevronov skritega oziroma izhodnega nivoja,
- funkcija *mult* zmnoži uteži in vhode, in jih sešteje (glej spodnji opis),
- funkcija *rangeLimit* omeji zapis rezultata na zahtevano število bitov (glej spodnji opis),

- vektorja *v1* in *v2* vsebujeta rezultat množenja in seštevanja (aktivacijski potencial) in
- vektorja *LUT in LUTd* vsebujeta tabelirano sigmoidno funkcijo in njen odvod.
- Ob koncu programa vektorski spremenljivki *y1* in *y2* vsebujeta izhode nevronov na skriti in izhodni plasti, ki so izračunani po enačbah 3.4 in 3.5.

```
% procesiranje
for n = 1: N1
    v1(n) = mult( W1(n,:), y0, precIO-1 + precWeight-1 - precLUT );
    v1(n) = rangeLimit( v1(n), precLUT);
    y1(n) = LUT( v1(n) + maxLUT + 2 );
end
y1(N1+1) = maxIO;
for n = 1: N2
    v2(n) = mult( W2(n,:), y1, precIO-1 + precWeight-1 - precLUT );
    v2(n) = rangeLimit( v2(n), precLUT);
    y2(n) = LUT( v2(n) + maxLUT + 2 );
end
```

Slika 3.6 Izvajanje večnivojskega perceptrona.

Funkcija, ki predstavlja množilnik *mult* vrača dva parametra. Parameter *scalar* je vektor, katerega elementi so produkti istoležnih elementov v vektorjih *w* in *x*, parameter *weightedsum* pa predstavlja skalarni produkt vektorjev *w* in *x*. S parametrom *rshiftbits* določimo, koliko bitov naj funkcija odreže od rezultata. Koda funkcije *mult* je predstavljena na sliki 3.7.

S funkcijo *rangeLimit* preverimo, ali je vrednost na vhodu ustrezna za zapis v spremenljivko, za katero imamo na voljo *prec* bitov. Če je vrednost prevelika ali premajhna, se spremenljivki priredi največje pozitivno oziroma najmanjše negativno število, ki ga lahko zapišemo s *prec* biti.

```
%*********************************************************************
%* Simulacija navronske mreze
%* Datoteka : mult.m
%*********************************************************************
function [weightedsum, scalar] = mult(w, x, rshiftbits)

scalar = w.*x;
%scalar
```

```
weightedsum = sum(sum(scalar));
%weightedsum
scalar = floor( scalar / 2^rshiftbits );
%scalar
weightedsum = floor( weightedsum / 2^rshiftbits );
%weightedsum
```

Slika 3.7 Funkcija *mult* – vzporedni množilnik.

```
%*********************************************************************
%* Simulacija navronske mreze
%* Datoteka : rangeLimit.m
%*********************************************************************
function y = rangeLimit(x, prec)
maxVal =  2^(prec-1)-1;
y = max( min( x, maxVal ), -maxVal-1 );
```

Slika 3.8 Funkcija *rangeLimit*.

3.3.2. Učenje

Potem, ko iz vhodnega vzorca izračunamo izhod iz nevronske mreže, ga primerjamo z željenim izhodom. Na podlagi razlike med izračunanim in željenim izhodom iz modela lahko v fazi učenja ustrezno popravimo proste parametre večplastnega perceptrona. V nadaljevanju je podana skrajšana oblika izvorne kode enega koraka učenja. Celotna koda, ki vsebuje tudi preverjanja rezultatov in izračun napake za kasnejši izris grafa napake, je predstavljena v Prilogi A.

Kot je razvidno iz kode na sliki 3.9, se najprej izračuna vektorska spremenljivka *d2b*, ki po enačbi 3.6 predstavlja razliko med vektorjem željenih izhodov (*t*) ter vektorjem dejanskih odgovorov modela (*y2*). Rezultat se z uporabo funkcije *rangeLimit* omeji na potrebno število bitov. Spremenljivka *d2a* predstavlja desni del produkta v enačbi 3.9 in se določi iz tabeliranih vrednosti odvodov sigmoidne funkcije, zapisanih v tabeli *LUTd*. Sledi izračunavanje vektorja *d2*, ki je po enačbi 3.9 kar vektor, katerega elementi so zmnožki istoležnih elementov v vektorjih *d2b* in *d2a.* Kot vidimo v kodi, je množenje izvedeno paralelno s funkcijo *mult*. Ker se v tem primeru vsota ne uporablja, smo rezultat priredili spremenljivki *dummy*1.

Sledi izračunavanje vektorja *d1a*, odvoda aktivacijske funkcije nevronov na skritem nivoju, ki se izračuna iz tabele odvodov sigmoidne funkcije *LUTd*. Komponente vektorja *d1b* se izračunajo v zanki *for*, in sicer iterativno, za vsak nevron izhodne plasti posebej. Posamezni elementi se sestavijo v vektor, ki ga predstavlja vsota v enačbi 3.11. Z

množenjem vektorjev *d1a* in *d1b* dobimo vektor *d1*, ki ga predstavlja celotna enačba 3.11.

```
% napaka
d2b = t - y2;
d2b = rangeLimit(d2b, precIO);
% delte
d2a = LUTd( v2 + maxLUT + 2);
[dummy1, d2] = mult(  d2a, d2b, precIO-1 );
d1a = LUTd( v1 + maxLUT + 2);
for n = 1: N1
    d1b(n) = mult( W2(:,n)', d2, precIO-1 );
    d1b(n) = rangeLimit( d1b(n), precWeight );
end
[dummy2, d1] = mult( d1a, d1b, precIO-1 );

% popravljanje uteži
for n=1: N2
    [dummy3, dW2] = mult( d2(n)*ones(size(y1)),y1, precIO- ...
                          1+log2(1/eta)-(precWeight-precIO)...
                          +log2(rangeWeight));
    W2(n,:) = rangeLimit( W2(n,:) + dW2, precWeight);
end
for n=1: N1
    [dummy4, dW1] = mult( d1(n)*ones(size(y0)), y0, precIO- ...
                          1+log2(1/eta) );
    W1(n,:) = rangeLimit( W1(n,:) + dW1, precWeight);
end
```

Slika 3.9 Korak učenja večplastnega perceptrona.

Korak učenja se zaključi s popravljanjem uteži po enačbah 3.12 in 3.13. Tako na izhodni kot na skriti plasti poteka popravljanje uteži iterativno, v zanki *for*, za vsak nevron posebej. Ponovno uporabljamo funkcijo *mult*, ki nam vrne rezultat (odvod napake po utežeh nevrona) v obliki vektorja, katerega elementi so produkti istoležnih elementov v obeh vhodnih vektorjih.

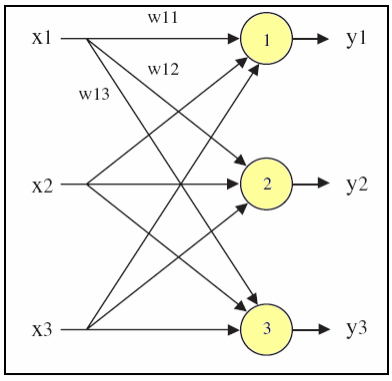
Pri zasnovi algoritma smo predpostavili, da je učni parameter potenca števila 2, tako, da pravo spremembo uteži lahko določimo z zamikanjem v desno za ustrezno število bitov.

4.

Izvedba nevronske mreže v jeziku VHDL

4.1. Pregled področja

V literaturi zasledimo kar nekaj izvedb nevronskih mrež na programirljivih logičnih vezjih [9]. Zelo pogoste so izvedbe, ki uporabljajo krožne registre. V nadaljevanju sta predstavljeni dve različici izvedbe nevronske mreže na sliki 4.1, katere izhodi so

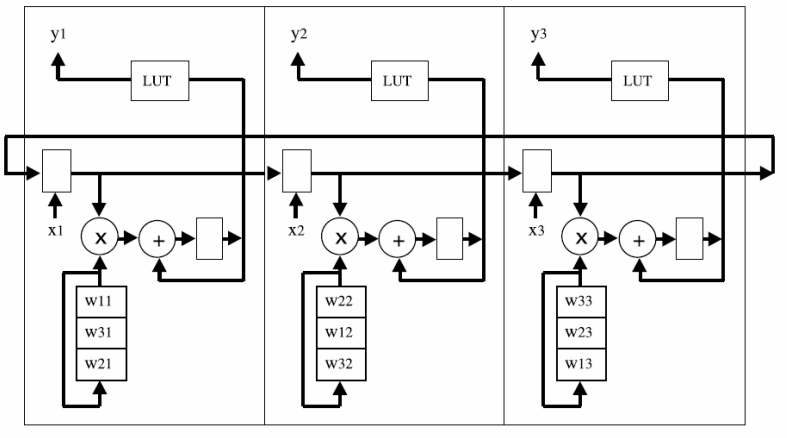


Slika 4.1: Nevronska mreža z eno plastjo nevronov.

določeni z enačbo

$$y_j = \varphi(\sum_{i=1}^{3} w_{ji} x_i) \quad , \quad j = 1,...,3 \quad . \tag{4.1}$$

Pri izvedbi, prikazani na sliki 4.2, je sigmoidna funkcija predstavljena v obliki tabele LUT (*ang*. LookUp Table), saj je izračunavanje aktivacijske funkcije tako prostorsko kot časovno neprimerno bolj potratno kot njeno tabeliranje. Izvedba predvideva krožni register za vhode in krožni register za uteži za vsak nevron posebej. Po treh ciklih izvajanja operacij s krožnimi registri dobimo izračunane vrednosti izhodov y1, y2 in y3.



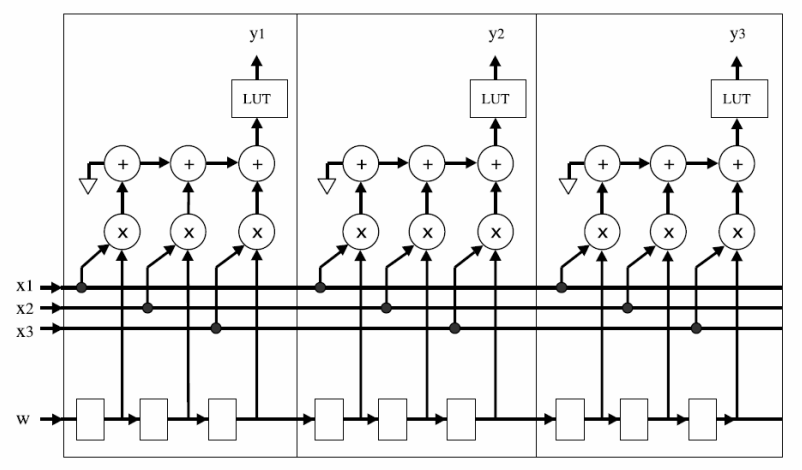
Slika 4.2: Izvedba nevronske mreže s krožnimi registri.

Računanje izhodov poteka na naslednji način:

- izračunajo se produkti $w_{11}x_1$, $w_{22}x_2$ in $w_{33}x_3$,
- ti produkti se zapišejo v registre delnih vsot,
- izračunajo se produkti $w_{12}x_1$, $w_{23}x_2$ in $w_{31}x_3$,
- ti produkti se prištejejo v registre delnih vsot,
- izračunajo se produkti $w_{13}x_1$, $w_{21}x_2$ in $w_{32}x_3$,
- ti produkti se prištejejo v registre delnih vsot,
- izhode nevronov dobimo na izhodih tabel LUT, v katere vstopajo registri delnih vsot.

Vidimo, da potrebujemo vsaj šest korakov za izračun novega stanja mreže. Tak pristop k implementaciji umetne nevronske mreže zahteva en množilnik in en seštevalnik za vsak nevron.

Drugačen pristop je prikazan na sliki 4.3. Tu so uteži tiste, ki vstopajo v sistem zaporedno, izračun izhodov pa se izvrši v enem koraku. Prednost tega sistema je nedvomno hitrost, vendar lahko opazimo, da so tokrat uporabljeni kar trije seštevalniki in trije množilniki za vsak nevron.



Slika 4.3: Izvedba nevronske mreže brez krožnih registrov.

Obe opisani izvedbi sta samo izvajali preslikavo vhodnih vzorcev na izhod, nobena pa ni upoštevala učenja mreže. Poleg tega je pri omenjenih izvedbah kar nekaj težav.

- Vsakič, ko naredimo en nevron, s tem naredimo tudi primerek seštevalnika in množilnika. Ti dve komponenti pa sta, kot bomo videli kasneje, precej potratni.
- Predstavljeni izvedbi ne vključujeta učenja, ki je mnogo bolj kompleksno od same preslikave vhoda na izhod tako v smislu matematičnih operacij kot tudi porabe komponent kot so množilniki in seštevalniki.

Iz zgornjih ugotovitev je razvidno, zakaj sem se odločil za lastno izvedbo nevronske mreže. Nobena od predlaganih izvedb namreč ni vključevala učenja nevronske mreže, kar pa je bilo ključno pri načrtovanju naše rešitve. Vsekakor bi lahko ob lepši in bolj potratni izvedbi hitreje in učinkoviteje načrtovali rešitev, vendar bi po vsej verjetnosti morali uporabiti večje in dražje čipe od čipov iz družine Xilinx Spartan. To pa so tudi razlogi, zaradi katerih sem se odločil, da bo osrednji element naše izvedbe nevronske mreže zmogljiva aritmetično logična enota ali enota ALE, ki je uporabljena za večino matematičnih operacij, ki se izvajajo v nevronski mreži.

4.2. Izvedba večplastnega perceptrona z enoto ALE

Cilj diplomske naloge je bila izvedba nevronske mreže z vzvratnim učenjem na čipu FPGA. Nevronska mreža pozna dva načina delovanja:

- računanje izhodov na podlagi predstavljenega vhodnega vzorca ter
- nadzorovano učenje na podlagi razlike med izračunanimi in želenimi izhodi.

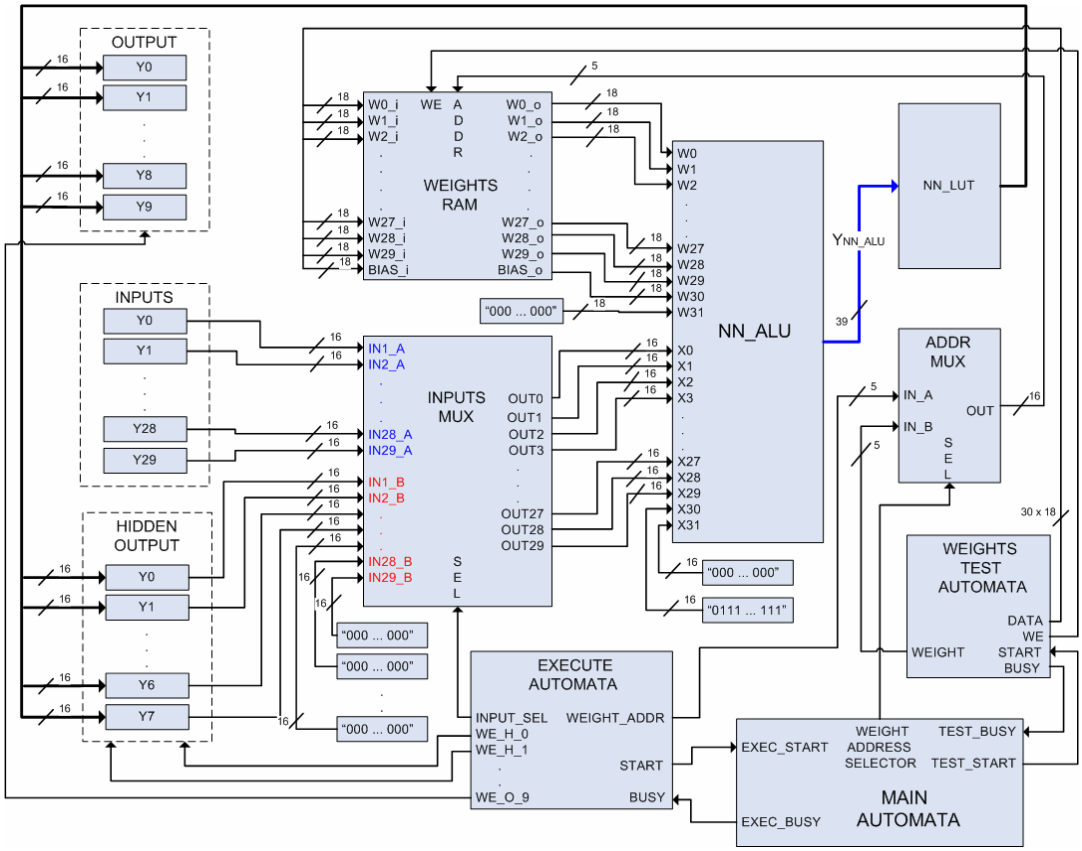
V naslednjem razdelku bo opisan prvi način delovanja, sledil pa bo razdelek v katerem bodo podane spremembe in dopolnitve osnovne sheme, potrebne za vključitev učenja v arhitekturo.

Predlagana izvedba nevronske mreže je zapisana v jeziku VHDL (*ang*. Very high speed integrated circuit Hardware Description Language). Ker je izvorna koda zelo obsežna, vsebuje namreč 14.793 vrstic kode v 36 datotekah, se bomo v nadaljevanju omejili na opis najpomembnejših delov, potrebnih za razumevanje predlaganega koncepta.

Naša izvedba nevronske mreže večplastni perceptron z enoto ALE je zasnovana popolnoma modularno in je v obliki sheme RTL (*ang*. Register Transfer Logic) predstavljena na sliki 4.4. V izvedbi večplastnega perceptrona z enoto ALE so uporabljene naslednje glavne enote, ki bodo podrobneje opisane v nadaljevanju:

- enota ALE (NN_ALU): naloga te enote je izvajanje vseh računskih operacij, ki se dogajajo v nevronski mreži,
- pomnilnik RAM (WEIGHTS RAM): pomnilnik, ki vsebuje vse uteži nevronske mreže,
- registri (INPUTS, OUTPUT, HIDDEN OUTPUT): ti registri vsebujejo vse izhode nevronov,
- izbiralnik za vhodne podatke v enoto ALE (INPUTS MUX): naloga tega sklopa je pripeljati ustrezne podatke v enoto ALE,
- tabela LUT (LUT): pomnilnik ROM, ki vsebuje tabelirane vrednosti sigmoidne funkcije,
- izvajalni avtomat (EXECUTE AUTOMATA): ta avtomat je, kot že ime pove, zadolžen za računanje izhodov iz nevronske mreže; glavna funkcija tega avtomata je preklapljanje potrebnih signalov v pravilnem zaporedju,

- testni avtomat (WEIGHTS TEST AUTOMATA): avtomat, ki je bil narejen v testne namene, saj naloži testne uteži v pomnilnik RAM; kasneje, ob dodanem algoritmu učenja, bo prevzel funkcijo inicializacije vseh uteži,
- izbiralnik naslovov za pomnilnik RAM (ADDR MUX): ta izbiralnik izbere izvor naslova, pomnilnika RAM, saj naslov generirata tako izvajalni avtomat, kot testni avtomat in
- glavni avtomat (MAIN AUTOMATA): njegova naloga je krmiljenje izvajalnega avtomata, testnega avtomata ter naslovnega izbiralnika.

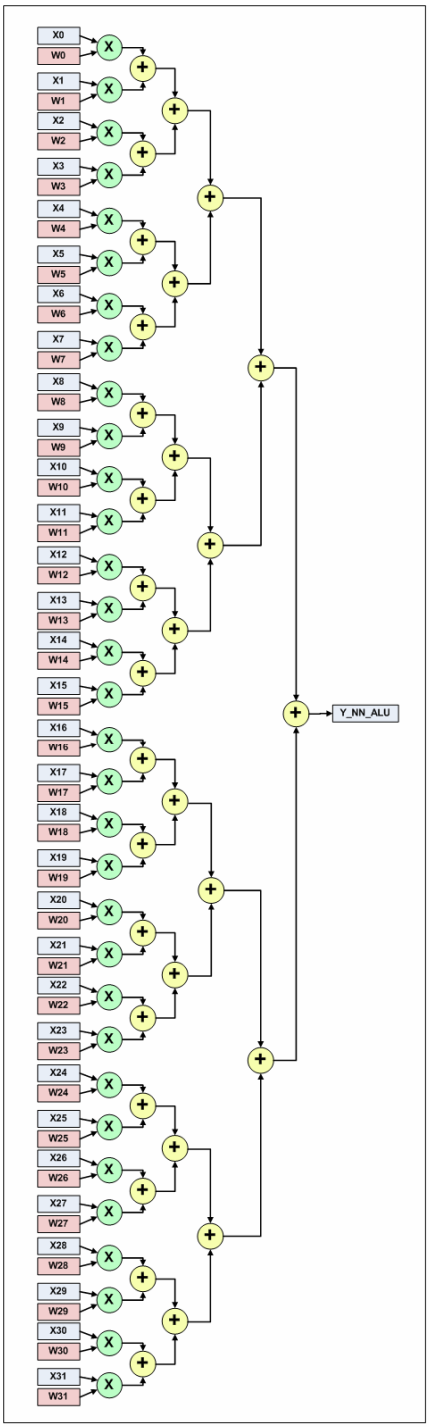


Slika 4.4: Shema RTL večnivojskega perceptrona z enoto ALE.

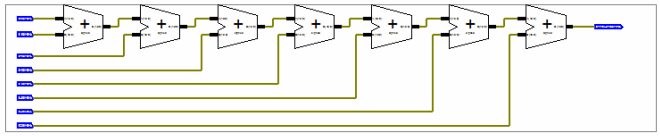
Pri nadaljnji obravnavi smo se zaradi zasnove množilnikov v vezju Xilinix Spartan 3AN omejili na 18 bitni zapis uteži. Za potrebe aplikacije, ki bo podrobno predstavljena v naslednjem poglavju smo zasnovali nevronsko mrežo večplastni perceptron s 30 vhodi, 8 nevroni na skriti plasti ter 10 nevroni na izhodni plasti. Zadovoljivo natančnost smo dosegli s 16 bitnim zapisom vhodov in ostalih signalov v nevronski mreži. Pri tem je potrebno poudariti, da je zasnova sistema popolnoma splošna.

4.2.1. Enota ALE

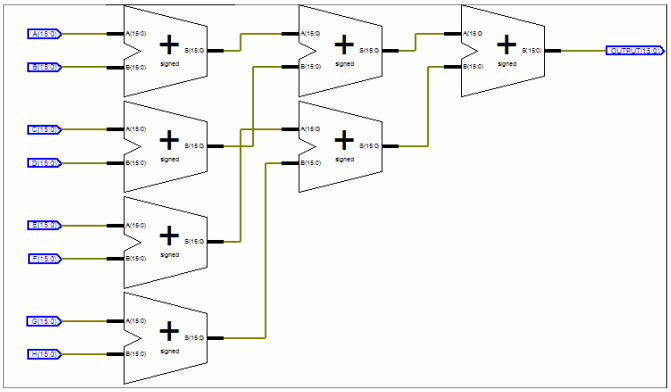
Najpomembnejša enota v naši izvedbi nevronske mreže večplastni perceptron je enota ALE, predstavljena na sliki 4.5.



a) Drevesna struktura seštevalnikov ALE enote



b) Sekvenčna implementacija seštevalnikov



c) Drevesna implementacija seštevalnikov

Slika 4.5: Shema enote ALE.

Enota ALE je zadolžena za množenje in seštevanje vhodov in uteži nevrona. Enoto ALE, opisano v nadaljevanju, sestavlja 32 množilnikov in 31 seštevalnikov, zaradi omejitev izbranega ciljnega čipa iz družine Xilinx Spartan 3AN. Ker je enota ALE izvedena kombinatorično, pomeni, da nimamo praktično nobenih dostopov do pomnilnika. Izjema so le zapisi izhoda iz tabele LUT v ciljne registre.

Prihranek časa pri procesiranju je glede na sekvenčna vezja očiten. Če bi enoto ALE izvedli z mikrokrmilnikom, bi za računske operacije porabili 32 množenj ter 31 seštevanj, skupaj kar 63 računskih operacij, ki bi se izvajale na hipotetičnem mikro-krmilniku v naslednjem zaporedju:

- postavi register R3 na 0,
- naloži utež v register R1,
- naloži vhod v register R2,
- zmnoži registra R1 in R2,
- prištej rezultat v register R3,
- če še niso obdelani vsi pari, skoči na drugi korak,
- shrani register R3.

Kot vidimo, smo porabili 32 množenj, 31 seštevanj ter 64 dostopov do pomnilnika, kar pomeni da lahko celotno nalogo taka CPE opravi v 127 urinih periodah. V primeru, da ima mikrokrmilnik več registrov, se to število seveda lahko drastično zmanjša. Vse to pa seveda velja ob predpostavki, da procesor zna delati s 64 bitnimi registri, saj je rezultat take operacije 39 bitno število. V primeru 32 bitnega procesorja se ti časi podaljšajo zaradi preverjanja preliva pri seštevanju, v primeru 16 bitnega procesorja pa se stvari še bolj zapletejo.

Čeprav vse omenjene računske operacije predstavljena enota ALE izvede v eni urini periodi, pa ima predlagana rešitev tudi svoje slabosti.

- Zakasnitveni časi čez tako vezje so neprimerno večji kot čez en množilnik in seštevalnik. Enota ALE tako na izbranem čipu Xilinx Spartan 3AN deluje ob maksimalni frekvenci 28,5 MHz. Vezje, ki vključuje samo en možilnik 18 x 18 dveh 18 bitnih števil na čipu deluje s frekvenco 89,2 MHz.
- Taka implementacija porabi vse množilnike 18 x 18 na čipu. Kot je razvidno v tabeli 1, jih je na čipu na voljo samo 20, ostalih 12 pa mora razvojno okolje FPGA sintetizirati samo, kar pomeni veliko porabo logičnih rezin.

- Sintetizirati je potrebno tudi vseh 31 seštevalnikov. Seštevalniki so z vsakim dodanim nivojem za 1 bit večji. Na zadnjem nivoju je seštevalnik 38 biten.

Poraba virov čipa FPGA za enoto ALE je prikazana v tabeli 4.1. Ob tem je potrebno povedati, da je slabost predlagane izvedbe tudi odvečen množilnik in seštevalnik. Da bi namreč lahko naredili drevesno strukturo, lahko kot število vhodnih parov vzamemo le potenco števila 2, kar je v našem primeru 32. Implementirana je bila torej ALE enota, ki zmnoži in sešteje 32 parov števil, čeprav bi bilo dovolj 31 (vse uteži in prag). Ker se zadnja dva vhoda ne uporabljata, sta, kot je to razvidno na sliki 4.5, vhoda *X31* in *W31* povezana na konstantni vrednosti nič, saj tako zagotovimo, da ne prispevata nič k končnemu rezultatu. Prag, ki ga predstavljata vhoda *W30* in *X30*, pa je na strani *X30* povezan na $7FFF_{Hex}$, kar pomeni maksimalno 16 bitno vrednost zapisano v dvojiškem komplementu.

Tabela 4.1: Poraba virov pri izvedbi enote ALE.

Vir	Porabljeno	Na voljo	Delež
Logične rezine	2673	5888	45%
Množilniki 18 x 18	20	20	100%

Če želimo doseči drevesno razporeditev seštevalnikov, moramo paziti, da kodo v jeziku VHDL pravilno zapišemo. Preprosto seštevanje osmih števil, pri katerem se izgubljajo biti prenosa, lahko v jeziku VHDL zapišemo tako, kot je prikazano na sliki 4.7.

```
entity adderSeq is
port(
      A,B,C,D,E,F,G,H : in signed (15 downto 0);
      OUTPUT : out signed (15 downto 0)
);
end adder;

architecture Behavioral of adderSeq is
begin
      OUTPUT <= A + B + C + D + E + F + G + H;
end Behavioral;
```

Slika 4.6: Sekvenčni seštevalnik osmih števil v jeziku VHDL.

Rezultat sinteze zgornje kode je shema, ki je prikazana na sliki 4.5b. Iz slike je razvidno, da iz te kode orodje ni naredilo drevesnega seštevalnika, ampak sekvenčnega. Da bi postavili drevesno vezavo, je potrebno najprej narediti delne seštevke, te pa nato sešteti na naslednjem nivoju, kot to prikazuje koda na sliki 4.7. Kot je razvidno iz sheme na sliki 4.5c, je tokrat sinteza drevesne strukture uspela.

```
entity adderTree is
port(
     A,B,C,D,E,F,G,H : in signed (15 downto 0);
     OUTPUT : out signed (15 downto 0)
);
end adder1;

architecture Behavioral of adderTree is
signal summ_AB, summ_CD, summ_EF, summ_GH : signed (15 downto 0);
signal summ_ABCD, summ_EFGH : signed (15 downto 0);
begin
     summ_AB <= A + B;
     summ_CD <= C + D;
     summ_EF <= E + F;
     summ_GH <= G + H;
     summ_ABCD <= summ_AB + summ_CD;
     summ_EFGH <= summ_EF + summ_GH;
     OUTPUT <= summ_ABCD + summ_EFGH;
end Behavioral;
```

Slika 4.7: Drevesni seštevalnik osmih števil v jeziku VHDL.

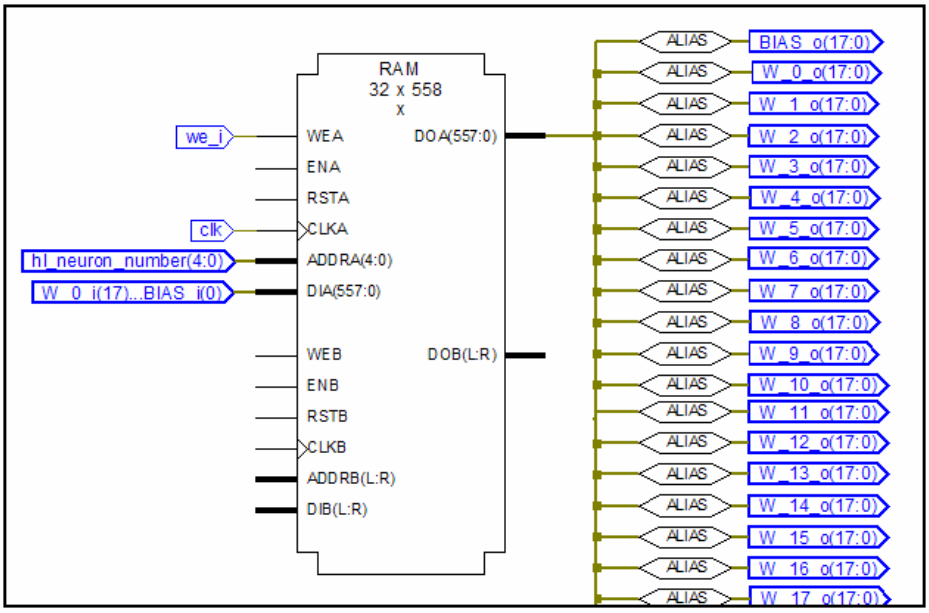
4.2.2. Pomnilnik RAM za uteži

Za večplastni perceptron, ki ima 30 vhodov, 8 nevronov na skriti plasti in 10 nevronov na izhodni plasti potrebujemo (30+1) x 8 = 248 uteži na skriti plasti in (8+1)x10 = 90 uteži na izhodni plasti. Če bi želeli tako strukturo opisati z registri, bi potrebovali 248 + 90 = 338 18-bitnih registrov za uteži, kar skupaj znese 6.084 sinhronih pomnilnih celic (*ang*. flip-flop). Izbran čip ima na voljo 11.776 sinhronih pomnilnih celic, kar je dovolj, vendar pa praktična uporaba tako velikega števila registrov ni smiselna, saj bi bile preklopne matrike za izbiranje izhodov iz registrov ogromne.

Da bi se izognili tej oviri, je bil potreben kompromis. Namesto registrov smo uporabili pomnilnik RAM za vse uteži. Pomnilnik RAM ima dovolj široko organizacijo, da se lahko naenkrat naslovijo vse uteži izbranega nevrona. Pri izvajanju večplastnega perceptrona je to dovolj za popolnoma paralelno računanje posameznega nevrona, kot bomo videli

kasneje, pa težava nastopi pri učenju, kjer so v enem od korakov zahtevani sekvenčni dostopi do uteži v pomnilniku RAM.

Struktura uporabljenega pomnilnika RAM je prikazana na sliki 4.8. Ker imamo skupaj 18 nevronov, od tega 8 na skriti plasti in 10 na izhodni plasti, rabimo za shranjevanje vseh uteži 18 pomnilniških naslovov. Ker je lahko število naslovov v pomnilniku RAM le potenca števila 2, smo prisiljeni rezervirati 32 naslovov. Iz slike 4.8 je razvidno, da bo razvojno okolje poskušalo sintetizirati pomnilnik RAM z globino 32 ter širino 31 x 18 = 558 bitov. Izhodi iz pomnilnika (*DOA*) so nato strnjeni v 18 bitne signale, ki so primerno poimenovani. To so prag *BIAS_o*, ter vse uteži *W_0_o* do *W_29_o.* Zaradi pomanjkanja prostora so na sliki označene samo uteži do vključno *W_17_o*.



Slika 4.8: Uporabljeni pomnilnik RAM.

Izkaže se, da pri sintezi takega pomnilnika pride do situacije, zaradi katere zasedemo več virov kot je to potrebno. Vezje na sliki 4.8 se v čipu Xilinx Spartan 3AN izvede kot pomnilnik BRAM, ki je razdeljen na 20 kosov. Vsak kos pomnilnika BRAM ima kapaciteto 18 kBit, širina vodila pa je 32 bitov, dodani pa so še dodatni štirje biti namenjeni preverjanju paritete [4]. Ker se tudi biti, namenjeni preverjanju paritete uporabljajo kot dodatni naslovi je končna širina organizacije pomnilnika 36 bitov.

Orodje za sintezo zasede 16 BRAM blokov organizacije 512x36, te pa nato poveže v 576 bitno besedo. Ker nevronska mreža potrebuje le 558 bitov za en nevron, je 18 bitov ostalo neizkoriščenih. Do še večje izgube pride zaradi globine pomnilnika BRAM. Od 512

lokacij je izkoriščenih le 18, torej samo 3,5 %. Zadnja izguba pa se zgodi zaradi izhodne plasti nevronske mreže. Ta je namreč povezana s skrito plastjo preko osmih nevronov, zaradi poenostavitev pa smo, tako kot na vhodnem nivoju, rezervirali pomnilniški prostor za 32 nevronov. Če upoštevamo vse te izgube, ugotovimo da se uporablja le 6.084 bitov od rezerviranih 294.912 bitov, kar znese dobra 2 % uporabe dejanskega rezerviranega prostora. To izgubo se da razpoloviti s tehniko rezervacije pomnilnika BRAM, do katerega lahko vršimo pisalni in bralni dostop hkrati. V tem primeru se pomnilnik razdeli na dva dela, ki omogočata samo en dostop hkrati, obenem pa imata polovično kapaciteto takega, ki lahko vrši oba dostopa hkrati [4].

4.2.3. Tabela LUT za računanje aktivacijske funkcije

V tabelo LUT smo shranili vnaprej izračunane vrednosti nelinearne aktivacijske funkcije, saj je matematično izračunavanje preveč zapleteno. Kot bomo videli kasneje, je 16 vrednosti v tabeli LUT dovolj za naše potrebe. Izhod enote ALE, ki predstavlja aktivacijski potencial, je 39 bitno število, ki ga je pred uporabo v tabeli LUT potrebno najprej skrajšati na 4 bite. Krajšanje izvedemo na najmanj pomembnih bitih tako kot zahteva funkcija *rangeLimit* v okolju Matlab. Pri krajšanju namreč ne smemo kar zavreči nepomembnih bitov, saj lahko pri tem naredimo hudo napako. Vzemimo za primer, da je v 11 bitnem številu zapisana vrednost +15, ki je v dvojiškem komplementu predstavljena z 00000001111. V tem primeru štirje najmanj pomembni biti 1111 predstavljajo desetiško vrednost -1, ki pa je popolnoma napačna. Glede na to, da vrednost +15 presega največjo tabelirano vrednost, bi bila prava izbira +7 ali binarno 0111.

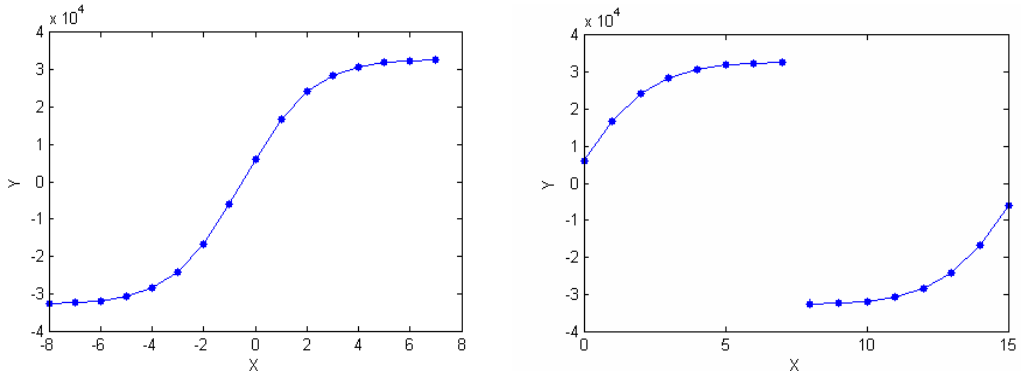
Modul v jeziku VHDL, ki nadomešča funkcijo *rangeLimit* v okolju Matlab, vidimo na sliki 4.9. S prvim stavkom *if* najprej ugotovimo ali je število pozitivno ali negativno. Če so na naslednjih sedmih bitih same ničle za pozitivna števila ali same enke za negativna števila, število skrajšamo, v nasprotnem primeru pa priredimo največjo pozitivno vrednost (+7) ali najmanjšo negativno vrednost (-8). Rezultat je štiri bitno število, ki predstavlja indeks tabele LUT.

Na sliki 4.10a je prikazan potek izbrane aktivacijske funkcije, v tabeli 4.2a pa so predstavljene vrednosti, ki so zapisane v tabelo LUT v 16-bitni natančnosti. Razvidno je, da je indeks tabele število na intervalu [-8, 7]. Ker je lahko naslov pomnilnika ROM le pozitivno celo število, bi bilo potrebno vrednosti tabele premakniti na interval [0, 15] ter pri izhodu nevrona prišteti odmik 8, da bi dobili željen naslov v tabeli ROM. Taka rešitev bi zahtevala dodatni seštevalnik na koncu izhoda iz enote ALE, ki bi prinesel dodatno urino periodo zakasnitve.

```
signal input : in std_logic_vector(38 downto 0);
process(input)
	begin
	if (input(38) = '0') then
		if (input(37 downto 31) = "0000000") then
			addr <= input(31 downto 28);
		else
			addr <= "0111";
		end if;
	else
		if (input(37 downto 31) = "1111111") then
			addr <= input(31 downto 28);
		else
			addr <= "1000";
		end if;
	end if;
end process;
```

Slika 4.9: Modul v jeziku VHDL, s katerim je izvedena funkcija *rangeLimit*.

Namesto tega se vrednosti v tabelo LUT lahko zapiše tako, da si števila na vhodu, zapisana v dvojiškem komplementu, predstavljamo kot nepredznačena števila. Tako si število na vhodu x = -2, ki je v 4-bitnem dvojiškem komplementu zapisano z znaki 1110 lahko predstavljamo kot nepredznačeno število 14 ali E_{Hex}. Če tabelo preuredimo z upoštevanjem omenjenih zvez, dobimo pravilen rezultat na izhodu brez prištevanja odmika. Slika 4.10b prikazuje potek preoblikovane funkcije, v tabeli 4.2b pa so zajete številčne vrednosti.



Slika 4.10: Aktivacijska funkcija v osnovni obliki a) in obliki, uporabljeni v tabeli LUT b).

Tabela 4.2: Tabelarični zapis nelinearne aktivacijske funkcije y = tanh(x) v osnovni obliki a) in obliki uporabljeni v tabeli LUT b).

a)

x (Dec)	x (Hex)	y (Dec)
-8	8	-32767
-7	9	-32500
-6	A	-31941
-5	B	-30794
-4	C	-28503
-3	D	-24169
-2	E	-16769
-1	F	-6092
0	0	6091
1	1	16768
2	2	24168
3	3	28502
4	4	30793
5	5	31940
6	6	32499
7	7	32767

b)

x (Hex)	y (Dec)
0	6091
1	16768
2	24168
3	28502
4	30793
5	31940
6	32499
7	32767
8	-32767
9	-32500
A	-31941
B	-30794
C	-28503
D	-24169
E	-16769
F	-6092

4.2.4. Registri in izbiralniki

Registri, prikazani na sliki 4.4, so zbrani v bloke OUTPUT, HIDDEN OUTPUT ter INPUTS in predstavljajo vhodne podatke v mrežo ter izhode vseh nevronov na skriti plasti ter na izhodni plasti. Število registrov na izhodni plasti je 10, na skriti plasti 8 ter na vhodni plasti 30. Vse skupaj torej 48 registrov s 16 bitno širino.

Izbiralnik registrov INPUTS MUX na sliki 4.4 mora na vhode v enoto ALE pripeljati željene podatke. Za izračun stanja nevronov na skriti plasti je potrebno pripeljati na vhod vse vhodne registre (INPUTS), za izračun stanja izhodne plasti pa je potrebno pripeljati na vhod vse registre skritega nevrona (HIDDEN_OUTPUTS). Ker se na skritem nivoju uporabi le 8 nevronov, so neuporabljeni vhodi izbiralnika povezani na nič, da ne vplivajo na izračun vrednosti stanja nevrona.

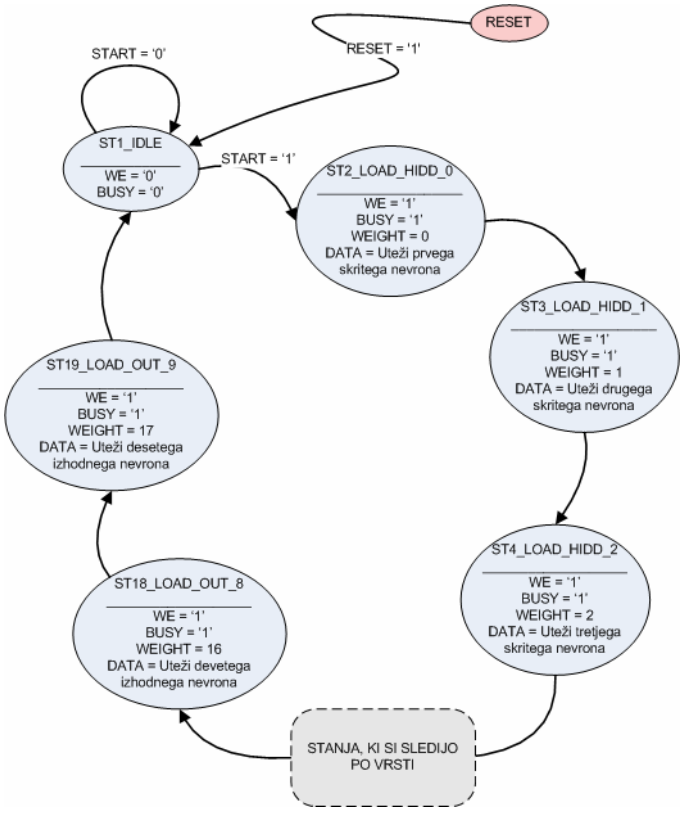
Naslove uteži potrebujemo ob pisanju nove uteži v pomnilnik in pri branju uteži iz pomnilnika. Ti dve operaciji nikoli ne potekata hkrati, zato je dovolj, če se uporabi

izbiralnik pred naslovnim vodilom ADDR MUX na sliki 4.4, ki se ob pravem trenutku preklopi. Za pravilne preklope skrbijo avtomati, ki so predstavljeni v nadaljevanju.

4.2.5. Avtomat za inicializacijo uteži

Avtomat za inicializacijo uteži je označen na sliki 4.4 kot WEIGHTS TEST AUTOMATA. Ta avtomat poskrbi za nalaganje vnaprej preračunanih uteži. Ker v prvo fazo načrtovanja arhitekture nismo vključili učenja smo s tem avtomatom naložili uteži, dobljene s simulacijo v okolju Matlab. Kasneje se je ta avtomat obdržal, pri čemer smo ga predelali tako, da inicializira uteži na naključne vrednosti.

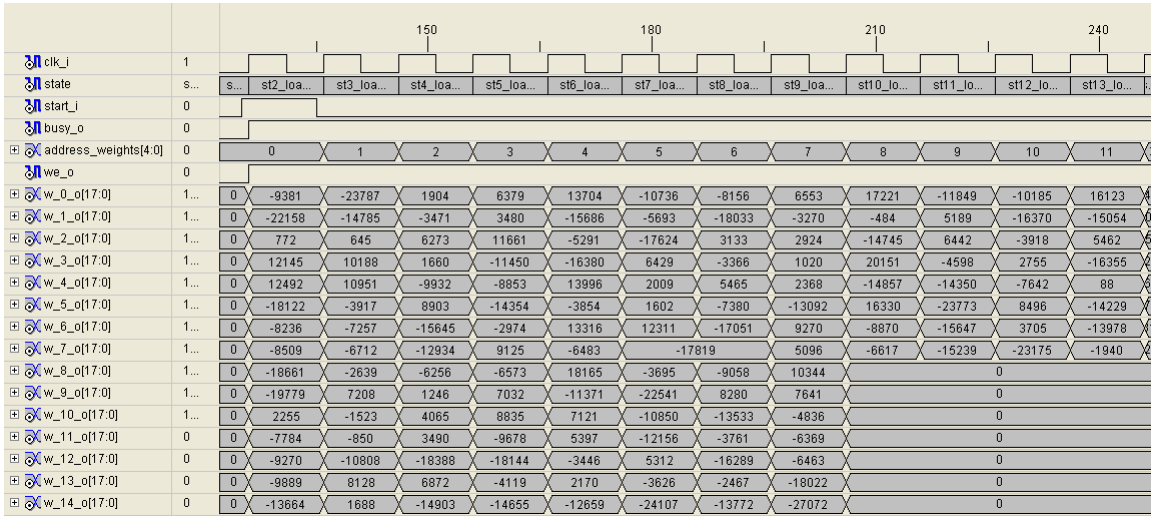
Avtomat je sila enostaven, kakor tudi njegovo izvajanje. Kot je razvidno iz slike 4.4, avtomat vsebuje vhod *START*, s katerim sprožimo začetek izvajanja in izhod *BUSY*, preko katerega sporoča, da je izvajanje v teku. Avtomat nato ustrezno spreminja naslov nevrona (*WEIGHT*), trideset 18 bitnih uteži (*DATA*) ter signal za vpis podatkov v pomnilnik RAM (*WE*).



Slika 4.11: Stanja avtomata za inicializacijo uteži.

Na sliki 4.11 so prikazana stanja avtomata za inicializacijo uteži. Avtomat ima toliko stanj, kot ima mreža nevronov ter enega dodatnega, za mirujoče stanje avtomata

(ST1_IDLE). Potem, ko s signalom *START* sprožimo avtomat, se avtomat iz mirujočega stanja premakne v prvo stanje, ki naloži podatke za prvi nevron, nato pa avtomat brezpogojno preide čez vseh 18 stanj in se nazadnje vrne v mirujoče stanje. V vsakem stanju se na izhodu pojavijo ustrezni podatki uteži na izhodih (*DATA*). Slika 4.12 prikazuje simulacijo časovnega poteka avtomata za inicializacijo uteži. Avtomat se izvede v 18 urinih periodah, po desetem stanju pa je tudi na simulaciji razvidno, da se uteži na skritem nivoju od osme dalje naložijo s samimi ničlami.



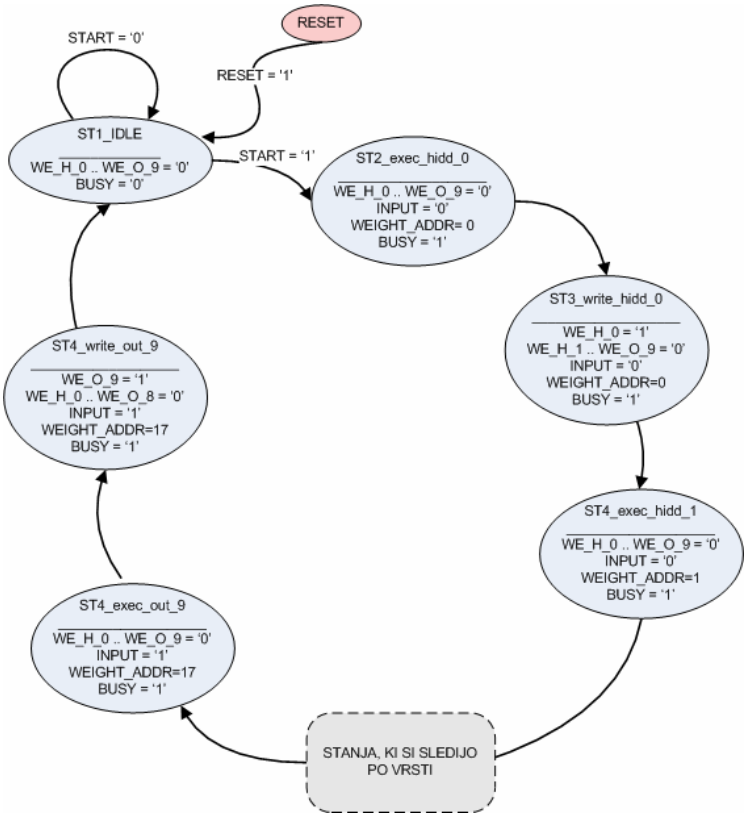
Slika 4.12: Časovni potek izvajanja avtomata za inicializacijo uteži.

4.2.6. Izvajalni avtomat

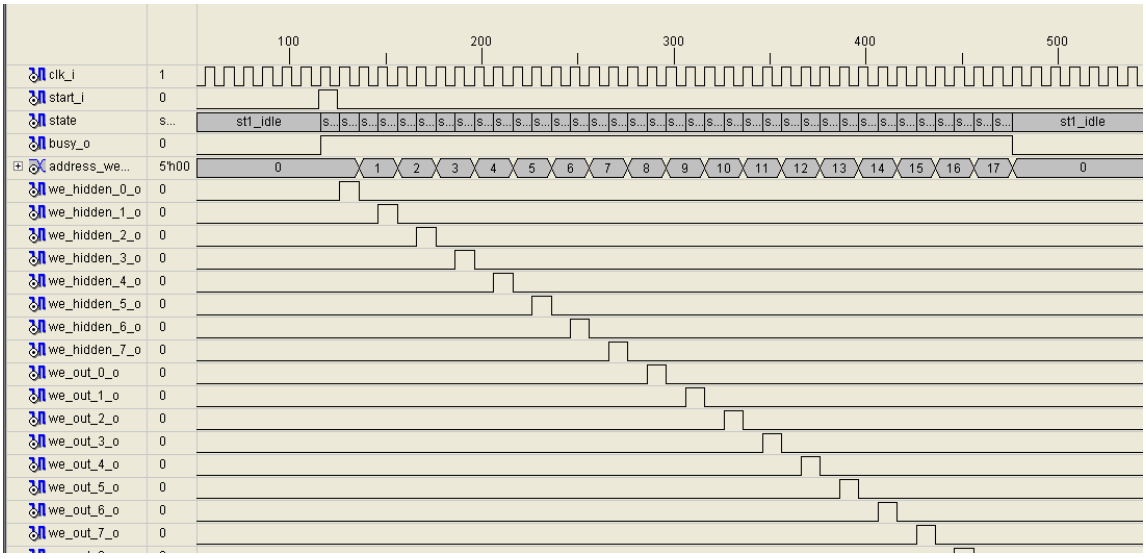
Izvajalni avtomat skrbi za pravilni potek računanja izhodov iz nevronske mreže. Njegova naloga je krmiljenje signalov za naslavljanje uteži vsakega nevrona posebej (*WEIGHT_ADDR*), krmiljenje signalov za vpis izhodne vrednosti v ustrezne registre (*WE_H_0, ..., WE_O_9)* ter krmiljenje signala za izbiralnik vhodnih podatkov v enoto ALE (*INPUT_SEL*). Na sliki 4.4 je označen kot EXECUTE AUTOMATA. Avtomat sprožimo tako, da na signal *START* postavimo logično '1'. Med izvajanje avtomata le-ta drži signal *BUSY* na logični '1'.

Na sliki 4.13 so označena stanja izvajalnega avtomata, na sliki 4.14 pa je predstavljen potek simulacije. Iz slik je razvidno, da se računanje izhoda vsakega nevrona posebej izvede v dveh urinih periodah. V prvi se izračuna novo stanje nevrona, v drugi pa se izračunana vrednost zapiše v primeren register tako, da se za eno urino periodo dvigne signal za zapis v register (*WE*). Signali *WE* si sledijo zaporedno v odvisnosti od tega, katero stanje nevrona se je v prejšnjem stanju izračunalo. Celotni avtomat in s tem

celotni izračun novih izhodov nevronske mreže z 18 nevroni, se izvede v 36 urinih periodah.



Slika 4.13: Stanja izvajalnega avtomata.

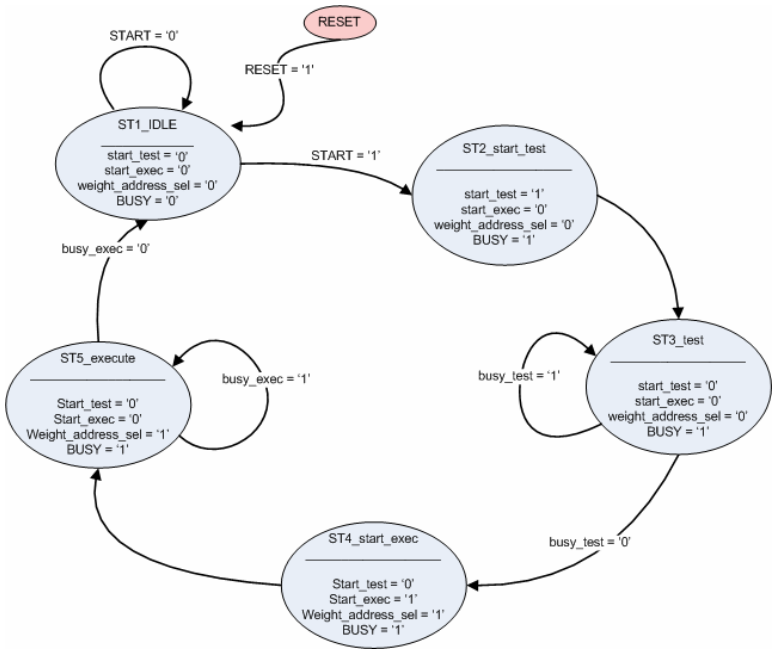


Slika 4.14: Časovni potek pri izvajalnem avtomatu.

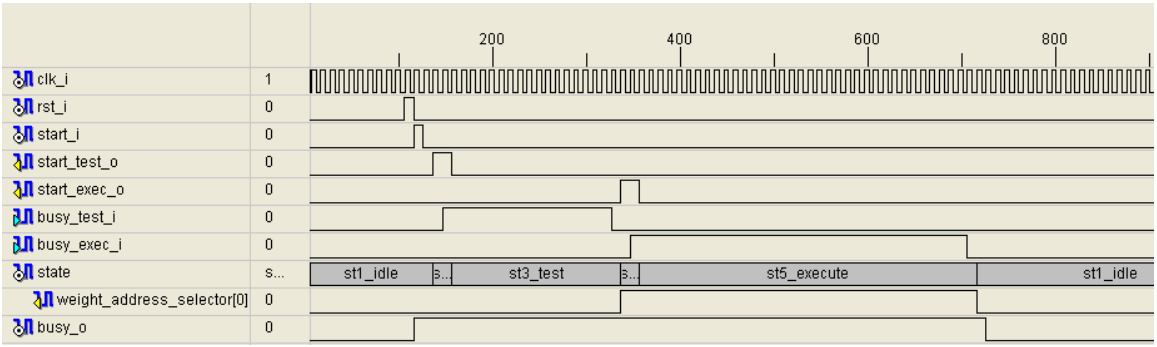
4.2.7. Glavni avtomat

Glavni avtomat je na sliki 4.4 označen kot MAIN AUTOMATA. Z njim najprej sprožimo avtomat za inicializacijo uteži, nato pa še izvajalni avtomat. Ko se izvaja avtomat za inicializacijo uteži, mora glavni avtomat prestaviti izbiralni vhod izbiralnika naslovov uteži ADDR MUX na '0'. Kasneje, ko teče izvajalni avtomat, pa mora biti omenjeni izbiralni vhod postavljen na logično '1'. S tem dosežemo, da je izvor naslovov za uteži enkrat avtomat za inicializacijo uteži, drugič pa izvajalni avtomat.

Slika 4.15 prikazuje stanja glavnega avtomata, slika 4.16 pa simulacijo delovanja, pri kateri vidimo, da avtomat čaka v stanju 3 (st_3_test), oziroma v stanju 5 (st_5_execute), dokler avtomat za inicializacijo uteži ali izvajalni avtomat ne končata z delom.



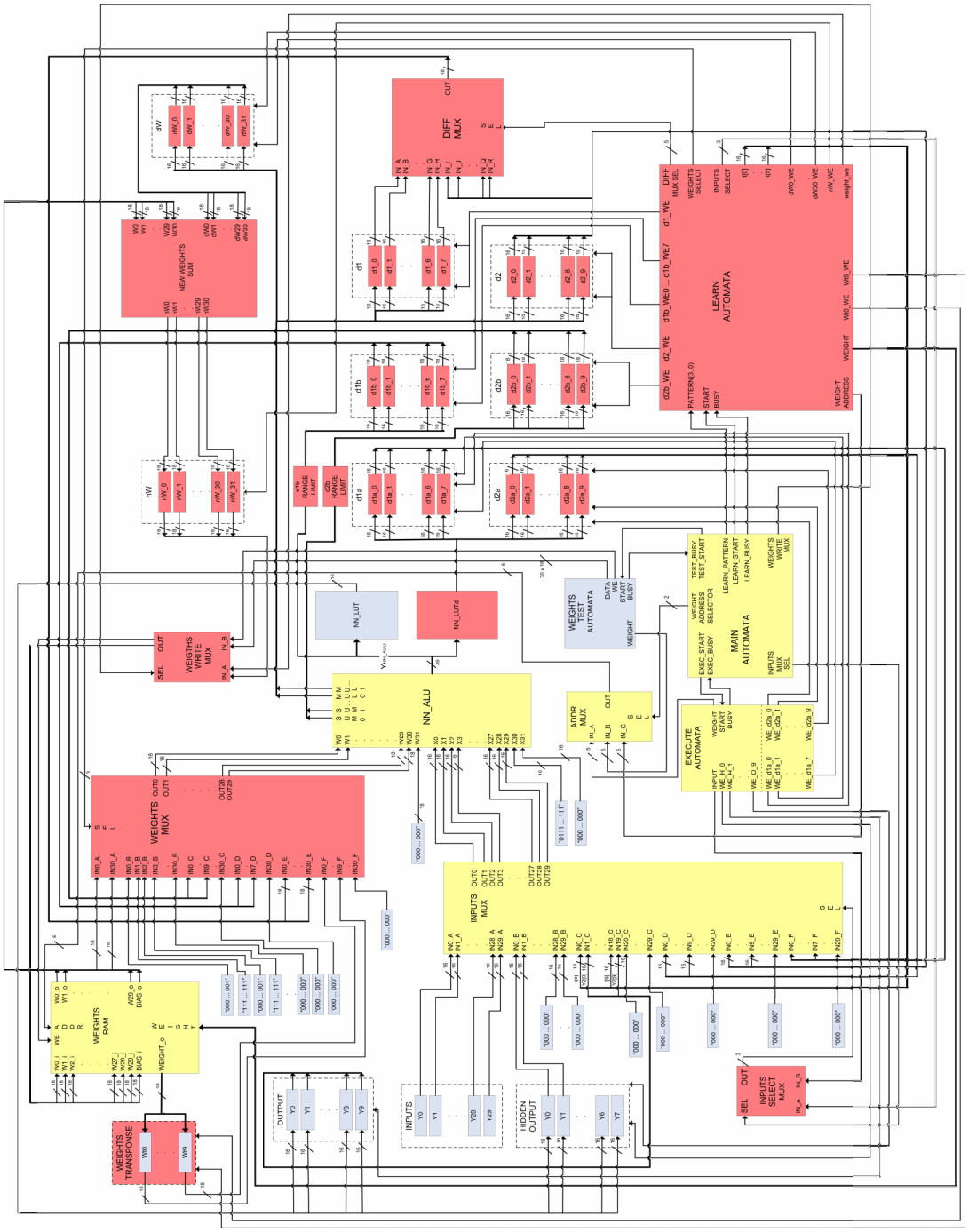
Slika 4.15: Stanja glavnega avtomata.



Slika 4.16: Časovni potek izvajanja glavnega avtomata.

4.3. Učenje večplastnega perceptrona

Vezje za nevronsko mrežo večplastni perceptron z vključenim učenjem, je predstavljeno na sliki 4.17.



Slika 4.17: Shema RTL za vezje večnivojski perceptron z učenjem.

Na sliki so z rumeno barvo označene enote, ki se od osnovnega vezja za večplastni perceptron z enoto ALE (slika 4.4) razlikujejo po dodani funkcionalnosti, z rdečo barvo pa so označene enote, ki so bile dodane na novo. Pri nadgrajevanju osnovnega vezja z moduli za učenje smo se v največji možni meri opirali na enoto ALE, ki smo jo predstavili v poglavju 4.2.1.

V nadaljevanju bodo predstavljene popravljene in nove enote. Vrstni red opisovanja enot bo sledil poteku simulacije v okolju Matlab, ki je predstavljena v poglavju 3.5. Zaradi lažjega sledenja predstavitvi bodo pomembni deli kode v okolju Matlab, ki je predstavljena na sliki 3.9, prikazani še enkrat.

4.3.1. Računanje vektorja napake

Učenje se začne z računanjem vektorja napake, ki ga predstavljata vrstici na sliki 4.18a. Konstanta *precIO* ima vrednost 16, kar pomeni, da je napaka predstavljena kot 16 bitno število.

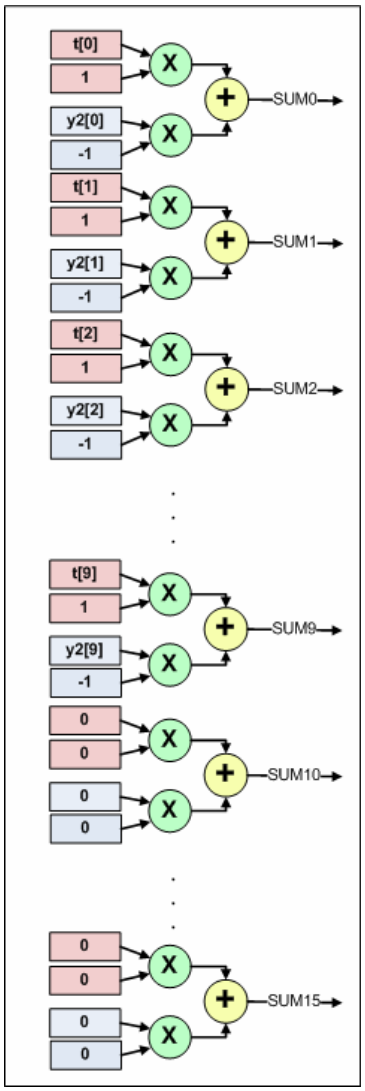
a)
```
% napaka
d2b = t - y2;
d2b = rangeLimit(d2b, precIO);
```

b)
```
% napaka
d2b = (1 * t) + (-1 * y2);
d2b = rangeLimit(d2b, precIO);
```

Slika 4.18: Računanje napake v okolju Matlab: a) osnovna oblika enačb in b) spremenjena oblika primernejša za izvedbo z enoto ALE.

Čeprav v prvi vrstici nastopa odštevanje, lahko vrstico zapišemo z uporabo množenja in seštevanja tako, kot prikazuje slika 4.18b. Taka oblika zapisa je zelo primerna za računanje s predlagano enoto ALE. Na sliki 4.19 vidimo, kako se vse komponente vektorja izhodov nevronske mreže (y2) pomnožijo z -1, vse komponente vektorja pričakovanega izhoda (t) pa se pomnožijo s +1. Izhodi iz sosednjih množilnikov se nato paroma seštejejo. Rezultat na izhodu seštevalnikov prvega nivoja predstavlja vektor napake izhoda (d2b).

Ker ima enota ALE v izvedbi nevronske mreže brez učenja samo en izhod (seštevek zmnožkov istoležnih elementov v vektorjih), smo ji morali dodati izhode iz seštevalnikov na prvem nivoju.



Slika 4.19: Ideja za uporabo enote ALE pri računanju napak na izhodu.

Za izvedbo te vrstice izvorne kode v okolju Matlab smo morali v izvorni kodi jezika VHDL spremeniti več stvari.

- Dodali smo izhode za delne vsote na prvem nivoju seštevalnikov v enoti ALE, označene kot *SUM0*, ..., *SUM15*. Ta dodatek ne porabi dodatnih virov, saj so delne vsote že implementirane, potrebno je bilo samo dodati signale, vidne navzven.
- Dodali smo nov vhod v izbiralnik INPUTS MUX za vhode v enoto ALE in povezali ustrezne signale. Dodali smo vhod *C*, na njegovih prvih 20 vhodov pa smo povezali izhode registrov OUTPUT, ki predstavljajo izhode nevronske mreže, ter pričakovan izhod *VEKTOR T*. Neuporabljeni vhodi so povezani na '0'.

- Dodali smo avtomat za učenje LEARN AUTOMATA. Kot vhod v avtomat podamo številko vzorca, za katerega bi radi, da se ga mreža nauči. Izhod tega avtomata je med drugim tudi želen izhod mreže, torej *VEKTOR T*. Ta avtomat prav tako krmili signal *d2b_WE* za vpis rezultata v novo množico registrov d2b. Ker izračun poteka paralelno, je en signal za vse registre dovolj.
- Dodali smo izbiralnik WEIGHTS MUX za vhodne uteži za enoto ALE. Na vhod *A* tega izbiralnika je sedaj povezan tudi pomnilnik RAM z utežmi WEIGHTS RAM, na vhod *B* pa so povezane konstante, ki so potrebne za izračun napake. Konstante so zapisane v obliki dvojiškega komplementa (1, -1, 1, -1, ..., 1, -1, 0, 0, ..., 0).
- Dodali smo funkcijski blok d2b RANGE LIMIT, ki upravlja funkcijo rezanja in prilagajanja rezultatov seštevalnikov na 16 bitov, kot smo to zasledili že pri tabeli LUT v poglavju 4.2.3.

Ker je funkcijski blok d2b RANGE LIMIT implementiran kombinatorično in ker izračun napake poteka paralelno, se le-ta izvrši v eni urini periodi. Pri izbiralniku za vhode INPUTS MUX v enoto ALE je sedaj potrebno upoštevati tudi to, da je enkrat izvor signala za izbiro vhoda *SEL* izvajalni avtomat EXECUTE AUTOMATA, drugič pa avtomat za učenje LEARN AUTOMATA. Ta problem reši dodaten izbiralnik INPUTS SELECT MUX, ki mu vhod *SEL* krmili glavni avtomat MAIN AUTOMATA glede na to ali je trenutno aktiven izvajalni avtomat ali avtomat za učenje.

4.3.2. Izračun odvodov aktivacijske funkcije

Kot smo videli v poglavju 3.5, se odvoda aktivacijske funkcije *d1a* in *d2a*, predstavljena na sliki 4.20, izračunata iz tabele LUTd, podobno kot se izhod aktivacijske funkcije izračuna iz tabele LUT. Izračun vrednosti se prav tako izvrši v isti urini periodi.

```
d1a = LUTd( v1 + maxLUT + 2);
d2a = LUTd( v2 + maxLUT + 2);
```

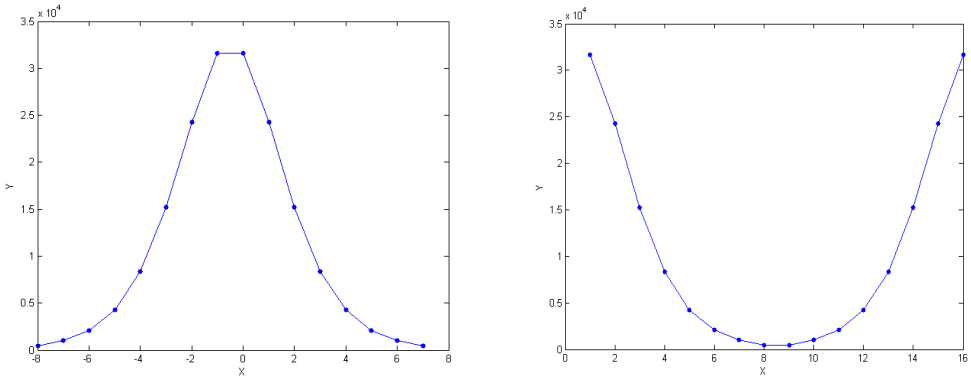
Slika 4.20: Računanje odvodov aktivacijske funkcije v okolju Matlab.

Tabela LUTd je realizirana na enak način kot tabela LUT, ki je bila opisana v poglavju 4.2.3. Osnovni potek tabelirane krivulje odvoda aktivacijske funkcije in potek, ki je prirejen za strojno izvedbo s tabelo LUTd, sta prikazana na sliki 4.21.

Za računanje odvoda aktivacijske funkcije je bilo potrebno:

- dodati tabelo LUTd na izhod seštevalnika in

- dodati osem registrov za vektor odvodov d1a ter deset registrov za vektor odvodov d2a in
- nadgraditi izvajalni avtomat EXECUTE AUTOMATA s signali *WE_d1a_0,…, WE_d2a_9* za pisanje v blok registrov d1a ter d2a, s katerimi lahko izhodne vrednosti tabele LUTd zapišemo v registre d1a in d2a.



Slika 4.21: Odvod aktivacijske funkcije v a) osnovni obliki in b) obliki vpisani v tabelo LUTd.

Ker se odvodi izračunajo v isti urini periodi kot izhodi iz nevronov, v tem primeru ne pride do dodatnih zakasnitev povezanih z učenjem nevronske mreže.

4.3.3. Računanje odvoda napake po aktivacijskem potencialu

Sledi izračun odvoda napake po aktivacijskem potencialu na izhodni plasti večplastnega perceptrona, ki smo ga v izvorni kodi okolja Matlab na sliki 3.9 označili kot vektor *d2*. Kot vidimo na sliki 4.22 ga dobimo kot zmnožek istoležnih elementov v vektorjih *d2a* in *d2b*.

```
[dummy1, d2] = mult( d2a, d2b, precIO-1 );
```

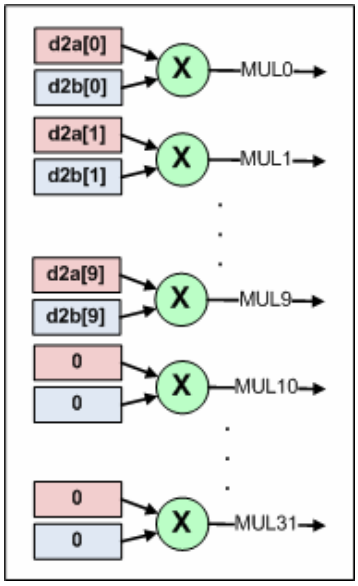
Slika 4.22: Računanje vektorja d2 v okolju Matlab.

Na sliki 4.23 je prikazana množica množilnikov, ki so uporabljeni za izračun vektorja *d2*. Primerjava s sliko 4.5 pokaže, da imamo te množilnike že na voljo v enoti ALE, vse kar moramo storiti je, da potrebne signale speljemo na njen izhod.

Za računanje odvoda napake po aktivacijskem potencialu smo zato prilagodili več stvari.

- Signale *MUL0* … *MUL31*, ki vodijo iz množilnikov v enoti ALE smo povezali na njen izhod.

- Dodati smo 10 registrov, ki predstavljajo vektor *d2*. Njihovi vhodi so povezani na ustrezne izhode enote ALE (*MUL0, ..., MUL9*).
- V avtomat za učenje LEARN AUTOMATA smo dodali signal za pisanje v registre vektorja d2 (*d2_WE*). Ker se računanje vektorja izvede paralelno, je en signal za pisanje, ki pelje v vse registre dovolj.
- Dodali smo nov vhod v izbiralnik za vhode enote ALE (INPUTS MUX), ki je označen s črko *D* in je povezan z izhodi vektorja d2a. Tudi v tem primeru so neuporabljeni vhodi povezani na '0'.
- Dodali smo nov vhod v izbiralnik za uteži (WEIGHTS MUX), ki je označen s črko *C*. Povezan je z izhodi vektorja d2b. Neuporabljeni vhodi so povezani na '0'.



Slika 4.23: Shema za množenje istoležnih elementov v dveh vektorjih.

Potrebno je dodati še, da so registri vektorja d2b drugačni od vseh do sedaj uporabljenih. Registri so sicer 16 bitni, vendar je izhod teh registrov 18 bitno število, saj to zahteva enota ALE, ki na vhodu pričakuje 18 bitno utež. Da bi pravilno razširili vrednost v registru, ni dovolj, da damo na zgornja dva bita kombinacijo '00', saj je vrednost lahko negativna. V slednjem primeru jo moramo razširiti s kombinacijo '11'. Rešitev v jeziku VHDL je predstavljena na sliki 4.24.

Ker je vektor d2 izračunan s pomočjo enote ALE se tudi tokrat vse komponente vektorja izračunajo v eni urini periodi.

```
process (q)
begin
        if (q(15) = '1') then
                data_out <= "11" & q;
        else
                data_out <= "00" & q;
        end if;
end process;
```

Slika 4.24: Razširjanje 16 bitnega registra *q* v 18-bitni register data_out.

Najzahtevnejši korak učenja je računanje odvoda napake po aktivacijskem potencialu za nevrone na skriti plasti, ki smo ga v kodi na sliki 3.9 označili kot vektor d1, ki ga dobimo kot produkt istoležnih elementov v vektorjih d1a in d1b. Vektor d1a smo izračunali že med računanjem izhoda iz nevronske mreže, manjka nam samo še izračun vektorja d1b, ki ga predstavlja koda na sliki 4.25.

```
for n = 1: N1
    d1b(n) = mult( W2(:,n)', d2, precIO-1 );
    d1b(n) = rangeLimit( d1b(n), precWeight );
end
```

Slika 4.25: Računanje vektorja d1b v okolju Matlab.

Največjo težavo predstavlja prvi stavek v zanki *for*. V njem nastopa transponiran vektor uteži na izhodnem nivoju W2(:,n)'. Ker so uteži v naši nevronski mreži shranjene v pomnilniku RAM (WEIGHTS RAM), z enim bralnim dostopom ni mogoče prebrati tega vektorja. Zato moramo komponente vektorja W2(:,n)' brati zaporedno, šele za tem sledi množenje z vektorjem d2. Ker z vsakim dostopom pomnilnika RAM, kjer so shranjene uteži, dobimo na izhodu pomnilnika vseh 29 uteži in prag, je bilo potrebno nadgraditi pomnilnik RAM tako, da je možno prebrati samo eno utež iz pomnilnika in ne vseh naenkrat.

Za to potrebne nadgradnje modela so predstavljene v spodnjih točkah.

- Pomnilnik RAM smo nadgradili tako, da sedaj omogoča dostop do posamezne uteži, ki jo je možno izbrati z vhodom *WEIGHT*. Izbrana 18-bitna utež se pojavi na izhodu *WEIGHT_o*.
- Dodali smo registre WEIGHTS TRANSPONSE z oznakami Wt0, …, Wt9, ki predstavljajo tansponiran vektor uteži, ki ga sestavljamo s sekvenčnim branjem posamičnih uteži iz pomnilnika.
- Avtomatu za učenja (LEARN AUTOMATA), smo dodali signale za izbiranje uteži (*WEIGHT ADDRESS*), signale za izbiranje posmezne uteži (*WEIGHT*) ter signale, ki

so potrebni za nadzor nad zapisovanjem komponent transponiranega vektorja (*Wt0_WE, ..., Wt9_WE*). Tudi izbiralnik za naslov uteži (ADDR MUX) smo nadgradili z novim vhodom *C*, ki je povezan na signal *WEIGHT ADDRESS* avtomata za učenje.

- Ko je vektor uteži transponiran, ga moramo pomnožiti z vektorjem d2. Zato je bilo potrebno razširiti izbiralnik za uteži (WEIGHTS MUX) z novim vhodom *F*, na katerega smo povezali transponiran vektor uteži (WEIGHTS TRANSPONSE). Prav tako se je razširil izbiralnik vhodov (INPUTS MUX) z novim vhodom *E*, na katerega je povezan vektor d2.
- Z množenjem in seštevanjem vektorjev d2 in Wt dobimo en element vektorja d1b. Tega najprej po že znanem postopku krajšanja obdelamo v enoti d1b RANGE LIMIT, nato pa ga zapišemo v ustrezen element vektorja d1b. Ker je ta izračun zopet sekvenčen, smo morali avtomat za učenje dopolniti s signali za pisanje vsakega elementa vektorja d1b posebej (*d1b_WE0, ..., d1b_WE9*).

To je najdaljši korak učenja, saj zahteva 10 urinih period za transponiranje izbranega vektorja uteži, kar pa je treba pomnožiti še z 8 koraki za izračun vektorja d1b. Skupno število urinih period potrebnih za to operacijo je torej 80.

Potem, ko sta znana vektorja d1a in d1b lahko izračunamo vektor d1. Ta izračun poteka na enak način kot izračun vektorja odvoda napake po aktivacijskem potencialu (d2). Elemente vektorja d1 dobimo z množenjem istoležnih komponent vektorjev d1a in d1b tako, kot prikazuje slika 4.26.

```
[dummy2, d1] = mult( d1a, d1b, precIO-1 );
```

Slika 4.26: Računanje vektorja d1 v okolju Matlab.

Za izračun vektorja d1 smo morali dodati več stvari, navedenih v spodnjih točkah.

- Izbiralniku uteži (WEIGHTS MUX) smo dodali nov vhod z oznako *D*, kamor se priklopijo izhodi vektorja d1b. Ti registri so podobni tistim, ki smo jih srečali pri vektorju d2b. Vhod je 16 biten, izhod pa 18 biten.
- Izbiralniku vhodov (INPUTS MUX) smo dodali nov vhod, označen z *F*. Nanj smo priklopili izhode vektorja d1a.
- Dodali smo novo množico registrov z imenom d1, ki predstavljajo izračunani vektor d1.
- Avtomatu za učenje smo dodali izhod, ki nadzira pisanje v registre vektorja d1.

Ker gre zopet za paralelno izvedbo množenja, je ta korak končan v eni urini periodi.

4.3.4. Izračun novih uteži

Ogledali si bomo izračun novih uteži na izhodnem nivoju, saj je iz slike 3.9 očitno, da je korak za izračun novih uteži na skritem nivoju praktično enak. Računanje novih uteži na izhodnem nivoju je še enkrat predstavljeno na sliki 4.27.

```
% popravljanje uteži
for n=1: N2
    [dummy3, dW2] = mult( d2(n)*ones(size(y1)), y1, precIO-...
                          1+log2(1/eta) - (precWeight-precIO)...
                          + log2(rangeWeight));
    W2(n,:) = rangeLimit( W2(n,:) + dW2, precWeight);
end
```

Slika 4.27: Računanje novih uteži v okolju Matlab.

Najprej moramo izračunati spremembe uteži dW2. To storimo tako, da pomnožimo vektor y1 z vektorjem, ki vsebuje samo izbrani element vektorja d2. To dosežemo tako, da izbrani element vektorja d2 večkrat povežemo na vhode enote ALE.

Za pravilno izvedbo popravljanja uteži smo v shemi dopolnili več komponent.

- Dodali smo izbiralnik za elemente vektorja d2 (DIFF MUX). Izbirni signal (*SEL*) novega izbiralnika je povezan na avtomat za učenje, kjer smo ustrezen signal označili kot *DIFF MUX SEL*.
- Dodali smo nov vhod v izbiralnik za uteži (WEIGHTS MUX), ki je označen s črko *E*. Nanj je povezan izhod novega izbiralnika za elemente vektorja d2.
- Dodali smo novo množico registrov z imenom dW, ki po množenju vektorjev vsebujejo spremembo uteži.
- Potem, ko izračunamo spremembe uteži, jih moramo prišteti k starim utežem. Zato smo dodali novo enoto, ki se ukvarja samo s seštevanjem uteži.

Za izvedbo zadnje od zgornjih točk bi lahko uporabili enoto ALE, ki seštevalnike že vsebuje, vendar je uporaba enote ALE v tem primeru neustrezna saj

- enota ALE vsebuje samo 16 seštevalnikov na prvem nivoju, ki bi bili za to uporabni,
- kot je razvidno iz slike 4.17, sta vhoda *X30* in *X31* enote ALE fiksno povezana na 1, oziroma 0, kar pomeni, da imamo na voljo samo 15 delnih vsot,
- ker moramo popraviti 30 uteži in 1 prag, bi z uporabo enote ALE izgubili 3 urine periode,
- če bi želeli uporabiti enoto ALE, bi morali dodati še 3 vhode v izbiralnik za uteži in enega v izbiralnik za vhode.

Testi so pokazali, da bi v primeru če bi za prištevanje popravkov uteži uporabili enoto ALE, porabili več logičnih elementov kot za izvedbo dodatnih 31 seštevalnikov. Za to potrebne dopolnitve so strnjene v naslednjih točkah.

- Dodali smo seštevalnike NEW WEIGHTS SUM z vgrajenim modulom RANGELIMIT. Seštevalnik na vhodu dobi stare uteži (*W*) in spremembe uteži (*dW*), na izhod pa postavi nove uteži (*nW*).
- Dodali smo množico novih registrov (nW) v katere shranjujemo popravljene uteži.
- Izhode registrov popravljenih uteži (nW) moramo zapisati v pomnilnik RAM. Ker v pomnilnik RAM lahko piše tudi avtomat za inicializacijo uteži, smo za nadzor pisanja v pomnilnik dodali še izbiralnik WEIGHTS WRITE MUX. Izbirni signal (*SEL*) ustrezno nastavlja glavni avtomat, glede na to, kateri od podrejenih avtomatov je aktiven.

Za vsak izračun sprememb uteži nevrona (dW) ter popravek uteži (nW) in vpis novih uteži v pomnilnik BRAM, je potrebnih šest urinih period. Za popravek vseh uteži v nevronski mreži je tako potrebnih 108 urinih period.

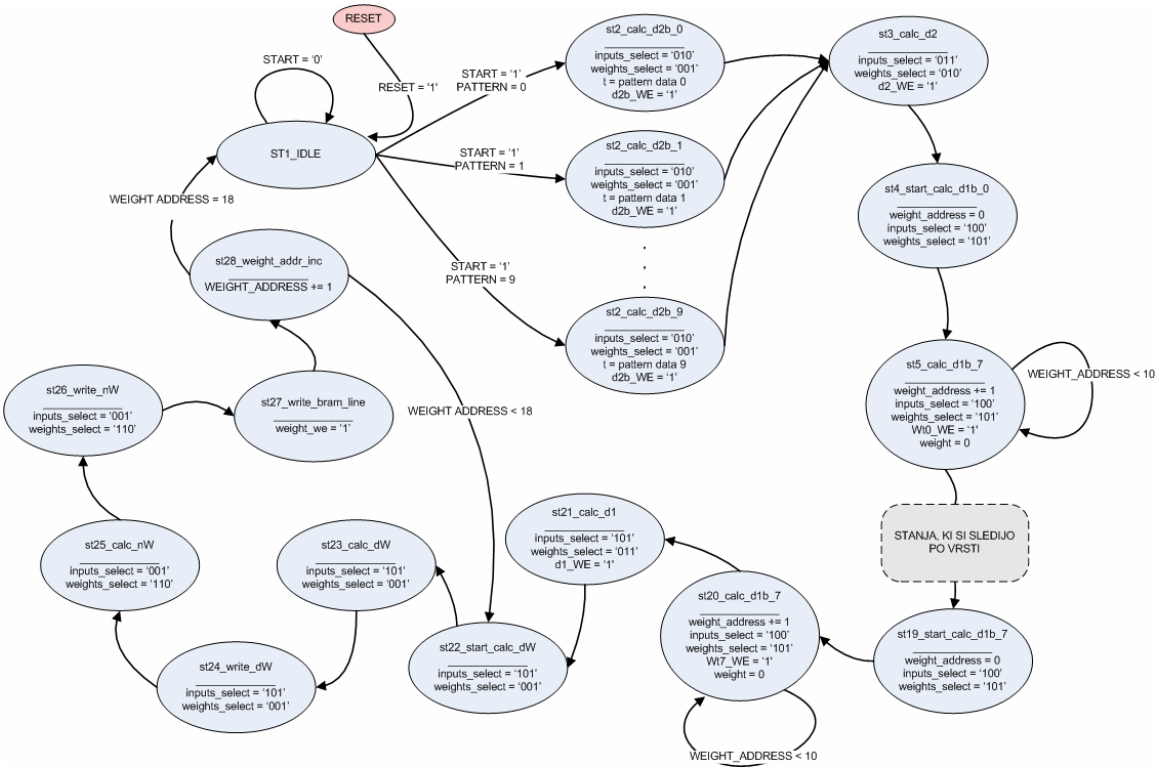
4.3.5. Avtomat za učenje

Pomembnejše spremembe na obstoječih avtomatih so zbrane v spodnjih točkah.

- Izvajalni avtomat (EXECUTE AUTOMATA) smo dopolnili z novimi signali, ki pomagajo pri izračunu odvoda aktivacijske funkcije (*WE_d1a_0, ..., WE_d2a_9*). Ti signali so podobni že opisanim signalom za izračun aktivacijske funkcije (*WE_H_0, ..., WE_O_0*), ki smo jih opisali v poglavju 4.2.6.

- Z izvajalnim avtomatom sedaj nadziramo tudi izbiralnik WEIGHTS WRITE MUX namenjen zapisovanju uteži v pomnilnik RAM.
- Za en korak učenja moramo najprej sprožiti izvajalni avtomat, ko zaključi z izvajanjem pa še avtomat za učenje. Nadzor nad delovanjem obeh avtomatov opravlja glavni avtomat, ki smo mu v ta namen dodali signale *LEARN_START*, *LEARN_BUSY* in *LEARN_PATTERN*.

Avtomat za učenje LEARN AUTOMATA ima v fazi učenja najpomembnejšo funkcijo. Ker je avtomat za učenje zelo obsežen in smo naloge, ki jih mora opraviti opisali že med obravnavanjem posameznih korakov v postopku učenja, je na sliki 4.28 podana samo poenostavljena shema stanj.



Slika 4.28: Shema stanj v avtomatu za učenje.

Izvajanje avtomata za učenje lahko v grobem razdelimo na 3 sklope. V prvem sklopu avtomat poda pričakovan vektor na izhod *t* in izračuna vektor napake. Ustrezna stanja so označena z st2_calc_d2b_0 do st2_calc_d2b_9. V drugem sklopu sledi izračun vektorja d2 (st3_calc_d2), nato pa izračun elementov vektorja d1b (st4_start_calc_d1b_0 do st20_calc_d1b_7). Temu sledi izračun vektorja d1 (st21_calc_d1). V tretjem sklopu se v zanki izračunajo še vse spremembe uteži dW ter vektorji novih uteži nW, nato pa se vektor nW zapiše v ustrezno lokacijo v pomnilniku RAM. Za to zanko so zadolžena stanja od st22_start_calc_dW do st28_weight_addr_inc.

Ko avtomat obhodi vsa stanja, so v pomnilniku RAM shranjene nove vrednosti uteži. Avtomat opravi en obhod v 281 urinih periodah. Ker je za vsako iteracijo učenja potrebno pognati tudi izvajalni avtomat, ki porabi 36 urinih period, lahko zaključimo, da ena iteracija učenja traja 317 urinih period.

5.

Razpoznavanje vzorcev z nevronsko mrežo izvedeno v vezju FPGA

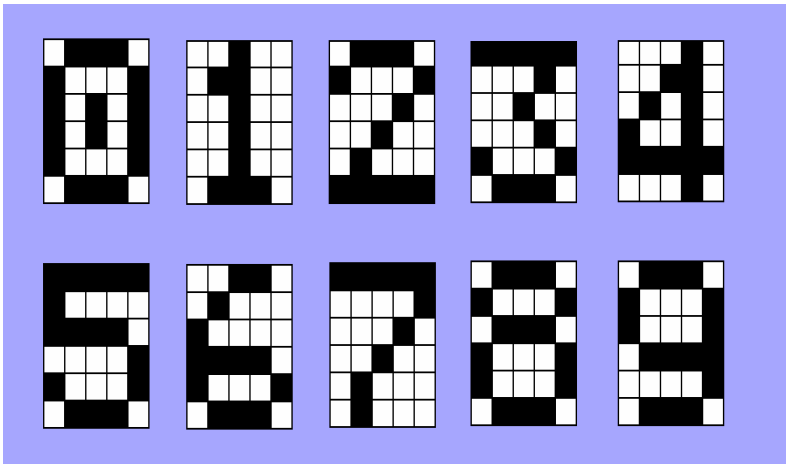
Področij uporabe nevronske mreže večplastni perceptron je veliko – od iskanja povezav v velikih množicah podatkov, podatkovnega rudarjenja, napovedovanja potekov funkcij, do razpoznavanja vizualnih vzorcev, govora in vodenja [7]. V tem poglavju se bomo omejili na razpoznavanje vizualnih vzorcev, natančneje na problem razpoznavanja znakov (*ang*. Character Recognition).

Večplastni perceptron izveden v čipu FPGA, ki smo ga predstavili v prejšnjem poglavju, bo na vhodu sprejemal slike števk v matrikah velikosti 6 x 5 točk in jih poskušal prepoznati tako, da bo glede na vhodno sliko števke aktiviral enega od desetih nevronov na izhodni plasti. Uporabljene slike števk so predstavljena na sliki 5.1.

Za to nalogo je bila primerna nevronska mreža, predstavljena na sliki 5.2, ki je sestavljena iz

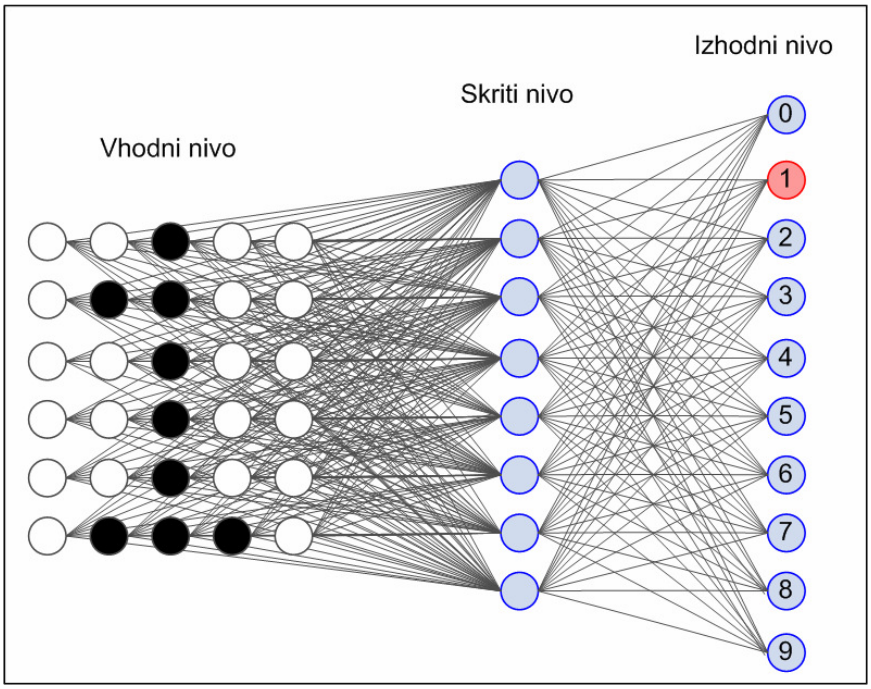
- 30 vhodov na vhodni plasti, ki ustrezajo predstavitvi števke v matriki 6 x 5 točk,

- 8 nevronov na skriti plasti in
- 10 nevronov na izhodni plasti, s katerimi mreža predstavi razpoznani vzorec.



Slika 5.1: Slike števk, ki predstavljajo vhode v nevronsko mrežo.

Čeprav se nevronska mreža uči na enobarvnih vzorcih, pa to ni nikakršna ovira za prehod na barvne globine, saj je model mreže postavljen splošno in ima 16 bitne vhode. V nadaljevanju obravnave bomo najprej utemeljili izbiro ključnih parametrov naše izvedbe, nato pa pokazali kako nevronska mreža deluje na čipu FPGA.



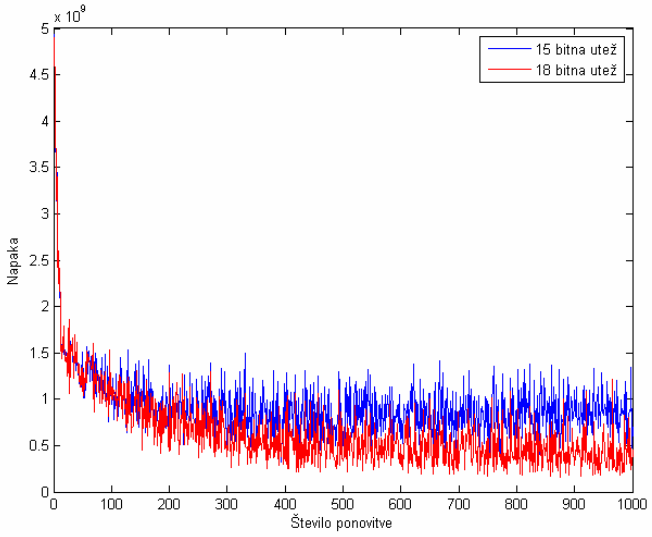
Slika 5.2: Večplastni perceptron, uporabljen pri razpoznavanju števk.

5.1. Določitev ključnih parametrov večplastnega perceptrona

Namen simulacije v okolju Matlab ni bil zgolj preverjanje delovanja in učenja mreže, ampak tudi določitev vseh ključnih parametrov, ki so potrebni za izvedbo večplastnega perceptrona v VHDL, predvsem bitne širine uteži, velikosti učnega parametra in bitne širine tabel LUT in LUTd. Za zagotovitev večje robustnosti metode smo v tej fazi vhodne slike števk med učenjem naključno pokvarili. Z verjetnostjo 12,5 % smo vsako slikovno točko spremenili iz črne v belo ali obratno.

Ker družina čipov Xilinx Spartan 3AN nima vgrajenih enot za delo s plavajočo vejico, sinteza takih enot pa je zelo potratna, se je treba zadovoljiti s celoštevilsko aritmetiko. Najprej smo določili bitno širino vhodov in uteži, s katerimi večplastni perceptron še zadovoljivo deluje. Za vhode smo izbrali 16 bitna števila, bitno širino uteži pa smo določil s pomočjo simulacije.

Graf na sliki 5.3 in tabela 5.1 kažeta, da z 18-bitnimi utežmi dosežemo zadovoljivo 90% uspešnost razpoznavanja vzorcev. Spodnji graf prikazuje potek napake modela (enačba 3.7) v odvisnosti od števila iteracij učenja. Simulacija prikazuje 10.000 iteracij učenja pri 16-bitnih vhodih ter 15 in 18-bitnih utežeh. Tabela 5.1 prikazuje uspešnost razpoznavanja vzorcev po končanem postopku učenja. Vidimo, da prehod iz 18 na 22-bitne uteži bistveno ne prispeva k večji uspešnosti prepoznavanja vzorcev.

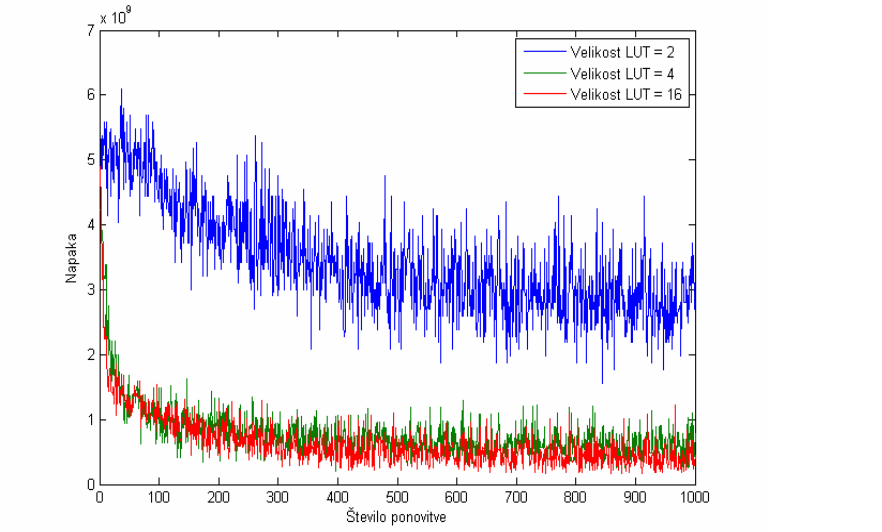


Slika 5.3: Potek učenja pri različnih bitnih širinah uteži.

Tabela 5.1: Uspešnost razpoznavanja v odvisnosti od bitne širine uteži.

Bitna širina uteži	Uspešnost razpoznavanja
15	64,84%
16	81,47%
17	88,82%
18	90,67%
22	91,25%

Naslednji pomemben parameter je velikost tabele LUT in LUTd. Ker vemo, da aktivacijske funkcije in njenega odvoda ne bomo izračunavali sproti, jo je potrebno tabelirati. Seveda se takoj postavlja vprašanje, koliko vrednosti potrebujemo v tabelah. Ker bodo vrednosti tabel zapisane v pomnilniku, je lahko število vrednosti le potenca števila 2. Na sliki 5.4 so izrisani poteki napake pri različnih velikostih tabel, v tabeli 5.2 pa so zbrani podatki o uspešnosti razpoznavanja vzorcev za različne velikostih tabel. Vidimo, da zadovoljive rezultate dosežemo že s samo 16 elementi v tabelah LUT in LUTd. Dovolj je torej pomnilnik s štirimi naslovnimi biti, saj je $16 = 2^4$.

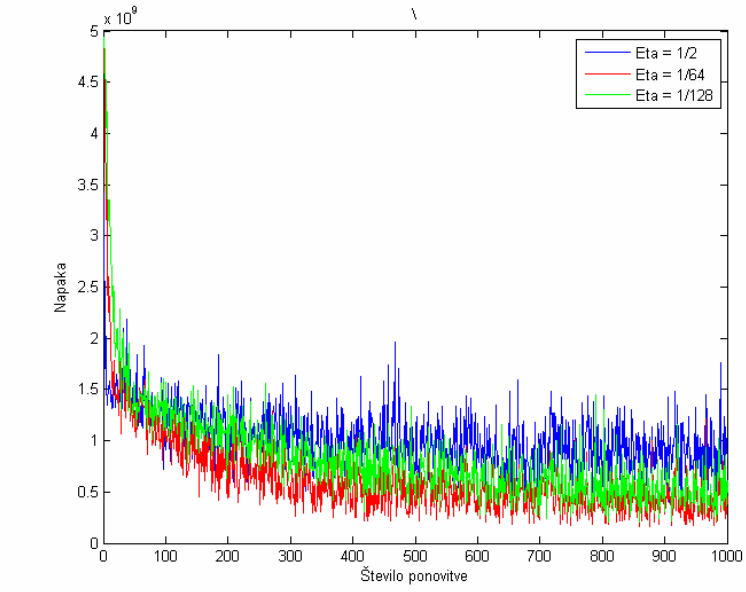


Slika 5.4: Potek učenja pri različnih velikostih tabel LUT in LUTd.

Tabela 5.2: Uspešnost razpoznavanja v odvisnosti od velikosti tabele LUT.

Velikost tabel LUT in LUTd	Uspešnost razpoznavanja
2	6,44%
4	73,26%
8	89,85%
16	90,07%
32	90,12%
64	90,60%

Zadnji parameter, ki ga je potrebno določiti je učni parameter η. Manjši kot je učni parameter η, počasneje se mreža uči, vendar pa ta parameter ne sme biti prevelik, saj se v temu primeru mreža ne nauči dovolj dobro. Učni parameter običajno leži na intervalu [0, 1]. Pri izvedbi na čipu je zaželeno, da je učni parameter potenca števila 2, saj množenje ali deljenje z učnim parametrom elegantno rešimo s pomikanjem bitov v levo ali desno. Kot kažeta slika 5.5 in tabela 5.3, je najbolj optimalna izbira za učni parameter $\eta = 1/64$.



Slika 5.5: Potek učenja pri različnih vrednostih učnega parametra.

Tabela 5.3: Uspešnost razpoznavanja pri različnih vrednosti učnega parametra.

Učni parameter η	Uspešnost razpoznavanja
1/2	68%
1/4	80,16%
1/32	90,20%
1/64	91,47%
1/128	87,99%

Kadar računanje in učenje poteka z realnimi števili, zaradi natančnosti zapisa nismo omejeni z velikostjo uteži in signalov. Nasprotno, pa moramo pri računanju s celimi števili zagotoviti, da so rezultati dovolj natančni. Zato smo pri gradnji modela v simulaciji in pri izvedbi na čipu omejili velikosti signalov. Velikosti uteži smo omejili na interval, ki je ekvivalent intervalu [-4, +4] pri realnih številih. Podobno smo velikosti signalov na izhodu iz nevronov omejili na interval, ki ustreza intervalu [-2, +2] na realni osi. V tabeli 5.4 so zbrani vsi parametri, ki so bili določeni s pomočjo simulacije v okolju Matlab.

Tabela 5.4: Parametri večplastnega perceptrona, uporabljeni pri razpoznavanju števk.

Parameter	Vrednost
Bitna širina vhodov	16 bit
Bitna širina izhodov	16 bit
Bitna širina uteži	18 bit
Učni parameter	1/64
Velikost tabele LUT	16 elementov
Interval uteži	[-4, +4]
Interval tabele LUT	[-2, +2]

Za realizacijo smo imeli na voljo eno samo razvojno ploščico, zato smo v projekt integrirali tudi aplikacijo. Potrebni dodatni moduli so predstavljeni v obliki sheme RTL na sliki 5.6.

5.2. Avtomat za generiranje učnih vzorcev

Ker je bila do sedaj nevronska mreža zelo splošna, se nismo spraševali kako bomo z zunanjim svetom povezali vse potrebne zunanje priključke. Nevronsko mrežo bi lahko povezali s 16 bitnim zunanjim vodilom, ki ima navzven vidne bralno-pisalne registre, na drugo stran pa bi priklopili mikro-krmilnik, ki bi pisal v registre in jih bral ter sprožil računanje izhodov na podlagi vhodnih vzorcev ali učenje.

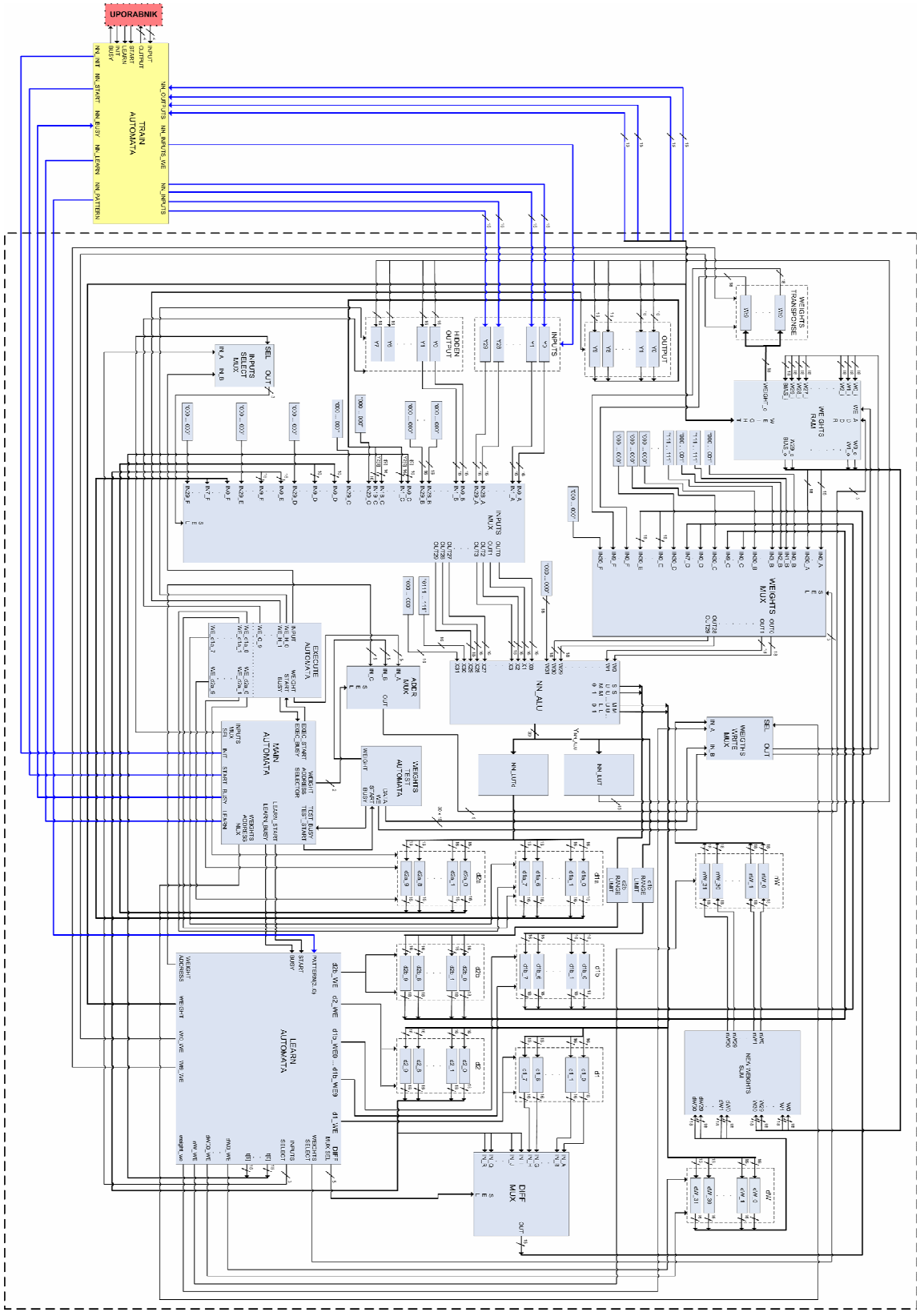
Mikro-krmilnika na uporabljeni razvojni ploščici ni, zato smo izdelali avtomat, ki krmili vezje za nevronsko mrežo. Avtomat za generiranje učnih vzorcev je na sliki 5.6 označen kot TRAIN AUTOMATA in predstavlja vmesnik med stikali in diodami LED na razvojni plošči ter nevronsko mrežo. Vsi signali avtomata, ki so povezani z nevronsko mrežo, se na shemi RTL začnejo z oznako *NN_*. Avtomata za generiranje učnih vzorcev ima več nalog.

- Na vhodu (*INPUT*) sprejme 4-bitno število, ki predstavlja vzorec, s katerim operiramo. Na podlagi tega vhoda nato avtomat izdela matriko števke (*NN_INPUTS, NN_INPUTS_WE*), ki smo si jo izbrali. Če si izberemo števko, ki je višja od 9, jo avtomat interpretira kot 0.

- S signalom *START* se začne avtomat izvajati. Uporabnik si lahko izbere način delovanja avtomata tako, da se sproži inicializacijo uteži (*INIT*), učenje (*LEARN*), ali računanje izhodov. Slednji način je aktiven vedno, kadar sta signala *INIT* in *LEARN* postavljena na '0'. Na vhode v nevronsko mrežo (*NN_INPUTS*) postavi avtomat ustrezno matriko, ki je odvisna od tega, katero številko si je uporabnik izbral za prepoznavanje (*INPUT*).

- Avtomat med delovanjem postavi izhod *BUSY* na logično '1'.

- Avtomat je namenjen tudi interpretaciji rezultata. Če je na izhodnem nevronu (*NN_OUTPUTS*) negativno število, ga interpretira kot logično '0', če pa je na izhodnem nevronu pozitivno število, ga interpretira kot '1'. Za tem pretvori 10



Slika 5.6: Shema RTL z vključenimi moduli za aplikacijo.

bitov v štiri. Za to je uporabljen navaden kodirnik:

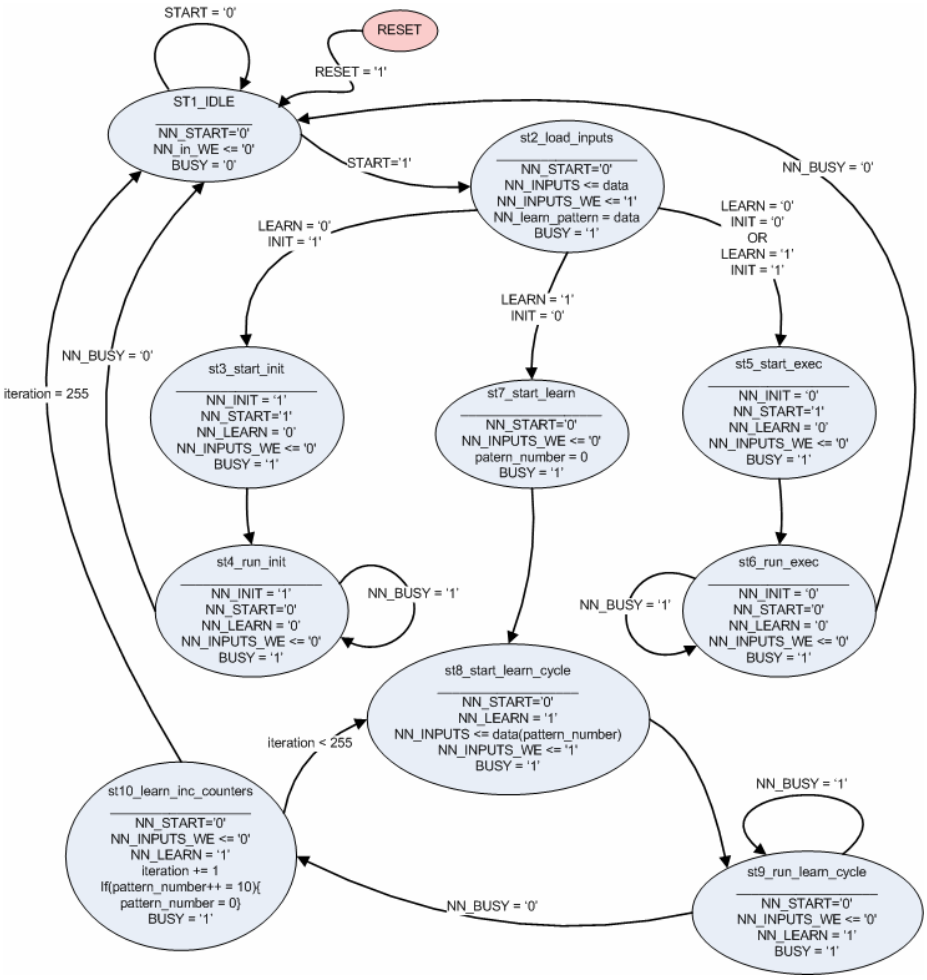
- $1000000000_{Bin} \rightarrow 0_{Hex}$
- $0100000000_{Bin} \rightarrow 1_{Hex}$
- $0010000000_{Bin} \rightarrow 2_{Hex}$

…

- $0000000001_{Bin} \rightarrow 9_{Hex}$

Kombinacije, ki so prepovedane (recimo 1000000100_{Bin}), se interpretirajo kot F_{Hex}.

- Avtomat nadzira tudi signale *NN_INIT*, *NN_START*, *NN_LEARN* glede na to ali uporabnik želi inicializacijo uteži, računanje izhoda ali učenje s popravljanjem uteži.



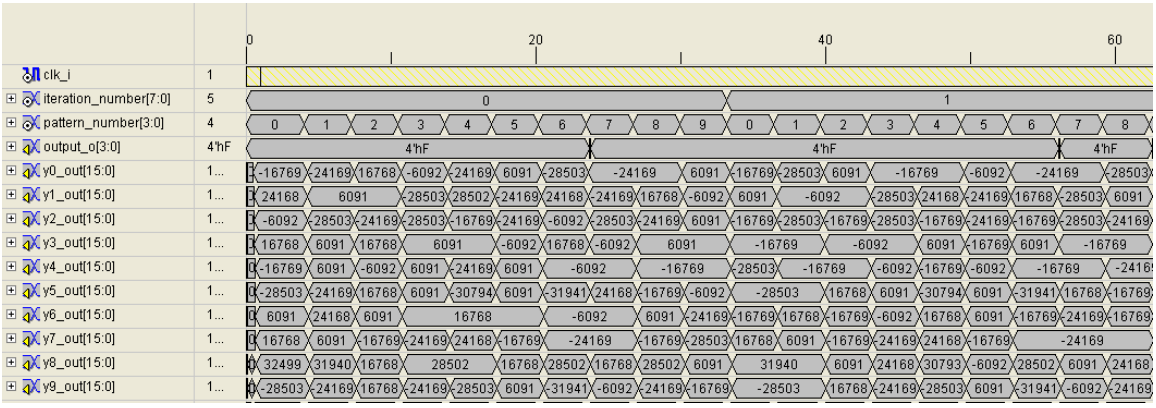
Slika 5.7: Shema avtomata za generiranje učnih vzorcev.

Kadar uporabnik sproži učenje, avtomat izvede 255 učnih ciklov, pri čemer v vsakem učnem ciklu nevronski mreži predstavi vse števke od 0 do 9. V 255 učnih ciklih se tako dejansko izvede 2.550 korakov učenja.

Shema prehajanja stanj v avtomatu je prikazana na sliki 5.7. Kot je razvidno iz sheme, avtomat lahko izvaja inicializacijo, računanje izhoda ali pa učenje. Pri učenju naj spomnimo, da je za njegovo izvajanje najprej potrebno izvesti računanje izhoda nevronske mreže. Ta korak opravi že glavni avtomat (MAIN AUTOMATA), saj se avtomat za generiranje učnih vzorcev, ki predstavlja uporabnika, s tem ne sme ukvarjati.

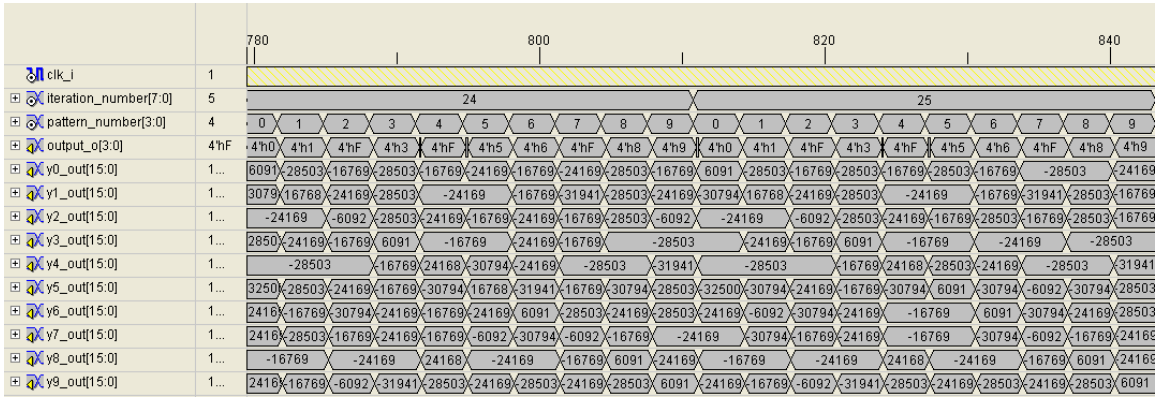
Pred testiranjem modela nevronske mreže na čipu smo opravili simulacijo, s katero smo potrdili pravilno delovanje celotne sheme RTL, ki je prikazana na sliki 5.6. Na sliki 5.8 je prikazan začetni potek simulacije. Na sliki je številka cikla učenja označena kot *iteration_number*, številka vzorca, ki se ga mreža uči pa kot *pattern_number*. Vidimo tudi, da se v vsakem ciklu učenja nevronski mreži predstavi vseh 10 vzorcev, ki so označeni od 0 do 9. Na sliki je kot *output* označen izhod, ki predstavlja prepoznani vzorec.

Ker so na začetku simulacije uteži inicializirane z naključnimi vrednostmi, vidimo, da mreža vzorca 0 ni prepoznala. Kot že vemo, negativna številka na izhodnem nevronu pomeni logično '0', pozitivna pa logično '1'. Na izhodnih nevronih (*y0_out, ..., y9_out*) se po prvem poskusu učenja številke 0 nahaja vektor [-16769, 24168, -6092, 16768, -16769, -28503, 6091, 16768, 32499, -28503], kar se prevede v izhodni vektor [0,1,0,1,0,0,1,1,1,0], ki pa ni veljaven vektor izhoda, zato se na izhodu (*output_o*) pojavi vrednost F_{Hex}.



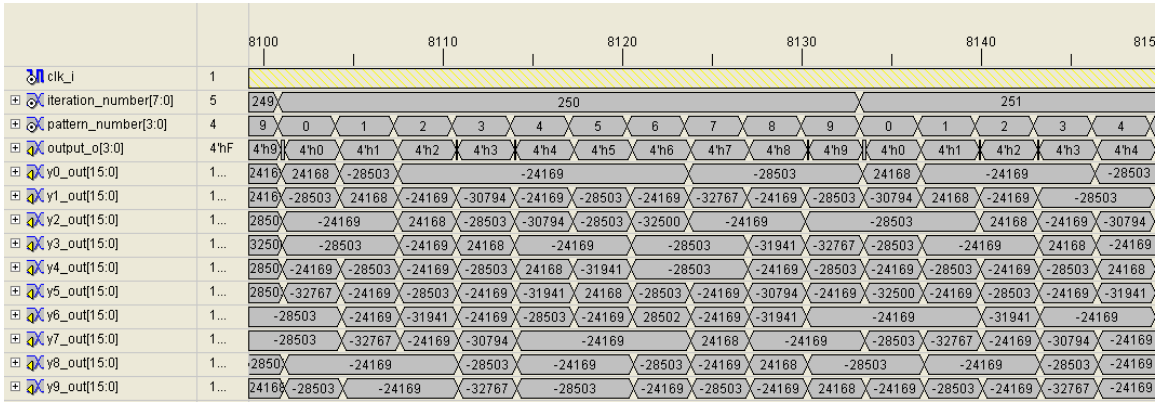
Slika 5.8: Simulacija prvega cikla učenja z avtomatom za generiranje učnih vzorcev.

Slika 5.9 prikazuje simulacijo 24. in 25. Cikla učenja. Jasno se vidi, da se je do sedaj mreža naučila prepoznati precej vzorcev. Težave ji še vedno povzročajo vzorci 2, 4 in 7. Ob tem dodajmo še, da so na izhodu lahko številke precej napačne po velikosti, a bo mreža za nas še vedno zadovoljivo delovala. Če si ogledamo kot primer izhod *y0_out*, opazimo, da je pri učenju vzorca 3 vrednost 6091, ki se precej razlikuje od želene vrednosti 26213, vendar je vseeno že dovolj za logično '1' na izhodu.



Slika 5.9: Simulacija 24. in 25. cikla učenja z avtomatom za generiranje učnih vzorcev.

Zadnji odsek simulacije je prikazan na sliki 5.10. Kot vidimo, je po 250 ciklih učenja mreža zelo uspešna, kar se vidi tudi po vrednostih na izhodu nevronov na izhodni plasti.



Slika 5.10: Simulacija 250. cikla učenja z avtomatom za generiranje učnih vzorcev.

5.3. Izvajanje nevronske mreže na čipu FPGA

Ker je verifikacija delovanja nevronske mreže uspela, jo lahko naložimo na čip FPGA in tudi preizkusimo. Na sliki 5.6 lahko vidimo avtomat generiranja vzorcev (TRAIN AUTOMATA), ki je namenjen tudi komuniciranju z uporabnikom. Za vmesnik med uporabnikom in avtomatom smo uporabili:

- rotacijski kodirnik, na katerega je povezan signal *START,*
- stikali, kamor sta priklopljena signala *LEARN* ter *INIT* in signal *RESET*, ki postavi vse avtomate v začetna stanja in vse registre na 0,
- drsne gumbe, kamor so priklopljeni signali vektorja *INPUT*,
- diode LED, kamor so priklopljeni signali vektorja *OUTPUT* in signala *BUSY*.

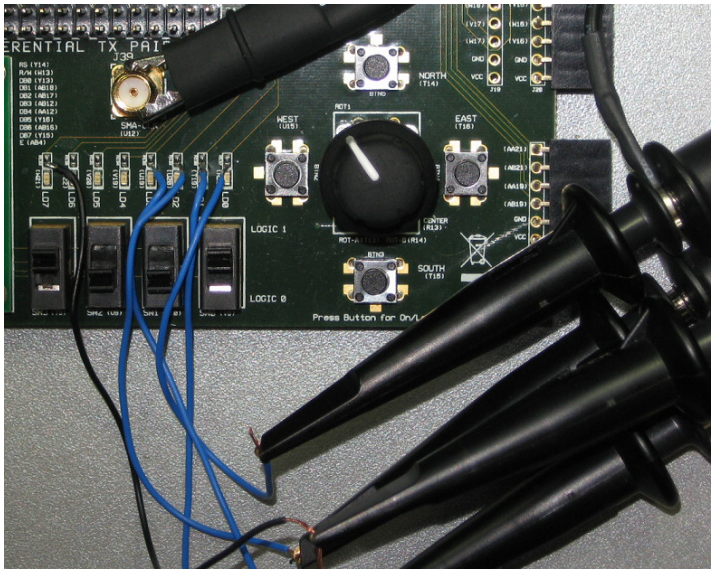
Da bi uspešno priklopili omenjene signale na vhodno-izhodne nožice čipa FPGA, smo morali v projekt v razvojnem okolju Xilinx ISE Webpack dodali še datoteko, ki opisuje priklop notranjih signalov čipa na vhodno-izhodne bloke. Na sliki 5.11 je predstavljena datoteka UCF (*ang.* User Constraint File), ki opisuje

- lokacijo izvora ure in tip ure. Izvor ure se nahaja na 50 MHz oscilatorju, ki je priklopljen na nožico E12, z amplitudo 3.3V [6],
- lokacijo vhodnega vektorja *INPUT*, ki se nahaja na drsnih stikalih SW3 do SW0,
- lokacijo signalov *RESET*, *LEARN* ter *INIT*, ki se nahajajo na stikalih East, South in West v navedenem vrstnem redu,
- proženje signala *START* ob pritisku na rotacijski gumb,
- signal *BUSY*, ki je povezan z diodo LED številka 7 in
- izhodne signale *OUTPUT* na diodah LED od številke 3 do številke 0.

```
NET "clk_i" PERIOD = 20.0ns HIGH 50%;
NET "clk_i" LOC = "E12" | IOSTANDARD = LVCMOS33;
NET "input_i(0)" LOC = "V8" | IOSTANDARD = LVTTL | PULLUP ;
NET "input_i(1)" LOC = "U10" | IOSTANDARD = LVTTL | PULLUP ;
NET "input_i(2)" LOC = "U8" | IOSTANDARD = LVTTL | PULLUP ;
NET "input_i(3)" LOC = "T9" | IOSTANDARD = LVTTL | PULLUP ;
NET "rst_i" LOC = "T16" | IOSTANDARD = LVTTL | PULLDOWN ;
NET "learn_i" LOC = "T15" | IOSTANDARD = LVTTL | PULLDOWN ;
NET "init_i" LOC = "U15" | IOSTANDARD = LVTTL | PULLDOWN ;
NET "start_i" LOC = "R13" | IOSTANDARD = LVTTL | PULLDOWN ;
NET "busy_o" LOC = "W21" | IOSTANDARD = LVTTL | SLEW = QUIETIO | DRIVE = 4 ;
NET "output_o(3)" LOC = "U19" |IOSTANDARD = LVTTL | SLEW = QUIETIO | DRIVE = 4;
NET "output_o(2)" LOC = "U20" |IOSTANDARD = LVTTL | SLEW = QUIETIO | DRIVE = 4;
NET "output_o(1)" LOC = "T19" |IOSTANDARD = LVTTL | SLEW = QUIETIO | DRIVE = 4;
NET "output_o(0)" LOC = "R20" |IOSTANDARD = LVTTL | SLEW = QUIETIO | DRIVE = 4;
```

Slika 5.11: Datoteka UCF.

Vse opisane enote so predstavljene na sliki 5.12. Na diode LED so priklopljene sonde osciloskopa, s katerimi smo naredili v nadaljevanju opisane meritve.



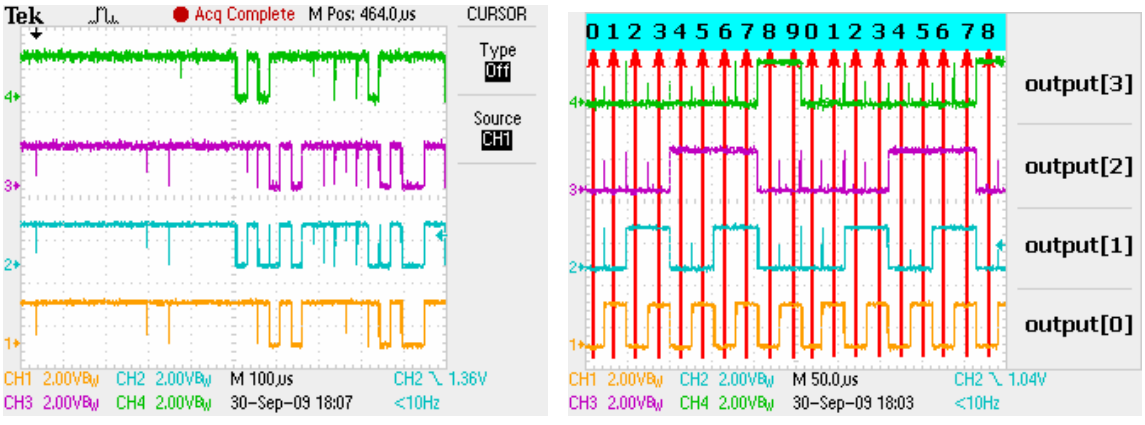
Slika 5.12: Stikala, gumbi in diode LED, ki predstavljajo uporabniški vmesnik.

5.4. Meritve izvajanja nevronske mreže na čipu FPGA

Meritve smo izvajali na štirikanalnem osciloskopu proizvajalca Tektronix, model TDS2024B. Ker je orodje za sintezo javilo, da je maksimalna frekvenca ure 23 MHz, mi pa imamo na razvojni plošči 50 MHz oscilator, smo delili osnovno frekvenco s 4, ter tako dobili najvišjo možno frekvenco delovanja, ki znaša 12,5 MHz.

Vezje se zbudi v neinicializiranem stanju, zato ga je potrebno najprej inicializirati. To storimo tako, da hkrati pritisnemo rotacijsko tipko (*START*) in tipko WEST (*INIT*) na razvojni ploščici.

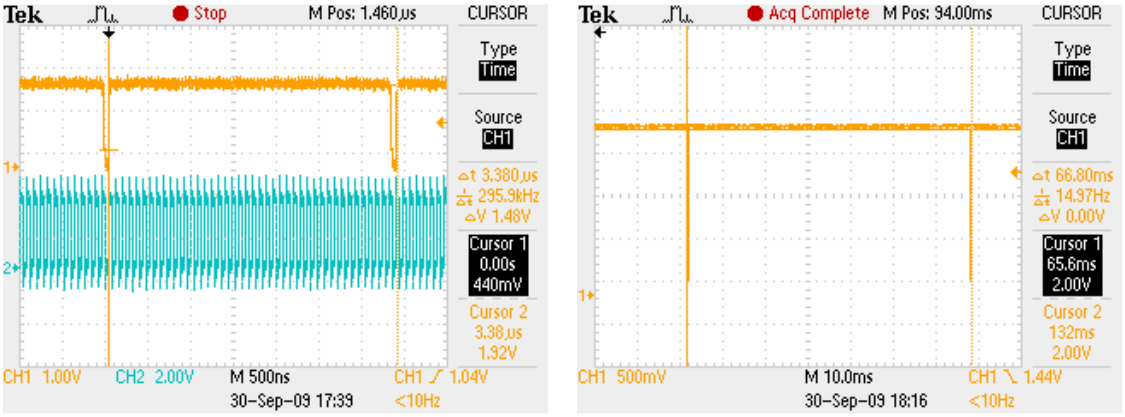
Meritve učenja na čipu so prikazane na sliki 5.13. Vsi štirje kanali osciloskopa so priklopljeni na izhod nevronske mreže. Kako so kanali priklopljeni na izhod, je na sliki označeno, prav tako je na sliki z rdečimi črtami označeno kako interpretirati podatke. Kakšno binarno vrednost podatki predstavljajo, je označeno na vrhu meritve. Kot vidimo na sliki 5.12a je na začetku izvajanja učenja nevronska mreža na izhode dajala neveljavne vektorje (F_{Hex}), čez nekaj časa pa je začela dajati na izhod nekatere vektorje, ki so bili veljavni. Na sliki 5.12b vidimo, da se je nevronska mreža že naučila in da gre očitno za cikel učenja, ki je najmanj petdeseti po vrsti. Motnje, ki jih opazimo med štetjem, lahko pripišemo algoritmu za računanje izhodov, ki se izvaja pred vsakim učenjem. Med računanjem izhodov se namreč lahko zgodi, da se zaradi spreminjanja izhodnega vektorja pojavi na izhodu prepovedana kombinacija. Tem motnjam bi se lahko izognili s sinhronimi pomnilnimi celicami, ki bi jih priklopili na izhode nevronske mreže in jih sinhronizirali na signal *BUSY* izvajalnega avtomata.



a) b)

Slika 5.13: Meritve izhodov iz nevronske mreže a) na začetku učenja in b) po končanem učenju.

Na sliki 5.13a vidimo čas izvajanja izvajalnega avtomata za računanje izhodov iz nevronske mreže. Na prvi kanal je priklopljen signal *BUSY* izvajalnega avtomata, na drugi kanal pa je priklopljen urin signal (*CLK*). Vidimo, da traja ena izvedba izvajalnega avtomata približno 3,4 µs. Na sliki 5.13b je podobno predstavljen signal *BUSY* avtomata za učenje. Vidimo, da se 255 iteracij učenja izvede v 66,8 ms, kar pomeni, da vezje za en korak učenja potrebuje približno 26,2 µs.



a) b)

Slika 5.14: Meritve časov izvajanja a) izvajalnega avtomata in b) avtomata za učenje.

Iz meritev je razvidno, da je paralelizem, ki smo ga uvedli z enoto ALE pomagal pri izvajanju in učenju mreže, saj so te naloge izvedene hitro, navkljub relativno nizki frekvenci delovanja. Če bi v vezje vstavili zunanji oscilator s frekvenco ure bližje maksimalni dovoljeni frekvenci, bi se eno računanje izhodov izvedlo v približno 1,8 µs, en korak učenja pa v 14,2 µs.

6. Zaključek

V diplomskem delu smo poskušali razviti nevronsko mrežo v čipu FPGA, ki jo lahko naučimo. Da bi to lahko storili čim bolj učinkovito, smo si najprej ogledali zgradbo čipa FPGA ter spoznali vse njegove pomembnejše elemente in razvojno ploščo, na kateri smo našo izvedbo nevronske mreže kasneje tudi testirali.

Omejili smo se na večplastni perceptron, eno najbolj standardnih in splošno uporabljanih nevronskih mrež. Osredotočili smo se na večplastni perceptron z enim skritim nevronom, ki smo ga učili z znano metodo vzvratnega učenja. Model večplastnega perceptrona smo postavili v okolju Matlab in ga prilagodili za računanje v celoštevilski logiki, pri čemer smo poskušali natančno posnemati omejitve, ki jih postavlja čip FPGA.

Izkazalo se je, da je za izvedbo računanj izhodov in učenja na enemu čipu smiselno narediti posebno aritmetično logično enoto, ki se nato uporabi za večino računskih operacij. Pri snovanju aritmetično logične enote smo poskrbeli, da se množenja in seštevanja v največji možni meri izvajajo paralelno. Za podporo aritmetično logični enoti smo dodali več izbiralnikov, ki smo jih nadzirali s hierarhično zasnovanimi avtomati. Celotno shemo sistema smo zasnovali in izvedli v jeziku za opisovanje strojne opreme VHDL.

Praktično uporabnost predlagane nevronske mreže na čipu FPGA smo potrdili na aplikaciji razpoznavanja znakov. Omejili smo se na razpoznavanje števk predstavljenih v obliki črno-bele matrike velikosti 6 x 5 slikovnih točk. Sledila so uspešna testiranja nevronske mreže na čipu FPGA, ki smo jih podkrepili z meritvami na osciloskopu.

Možnosti za nadaljnje delo je veliko. Vsekakor bi bilo zanimivo videti kako se mreža obnese na vzorcih, ki so predstavljeni s sivinami. Zanimivo bi bilo tudi raziskati za koliko je potrebno povečati bitno širino uteži ali pa povečati število nevronov v skriti plasti, da bi bilo uspešno tudi razpoznavanje močno zašumljenih znakov. Prav tako zanimivo bi bilo pridobiti informacijo o tem, kako bi se ta nevronska mreža obnesla, če bi bila izvedena v po meri narejenem čipu (*ang.* Application-Specific Integrated Circuit, ASIC).

7.

Literatura

[1] R. Rojas: Neural Networks, Springer, Berlin, 1996.

[2] A. R. Omondi, J. C. Rajapakse: FPGA Implementations of Neural Networks, Springer, Dordrecht, 2006.

[3] K. Parnell, N. Metha: Programmable Logic Design Quick Start Handbook, Xilinx, 2004

[4] Xilinx: Spartan-3 Generation FPGA User Guide, Xilinx, 2009.

[5] P. Bulić: Načrtovanje in sinteza digitalnih in mikroprocesorskih sistemov v FPGA, skripta, Univerza v Ljubljani, FRI, 2007.

[6] Xilinx: Spartan-3A/3AN FPGA Starter Kit Board User Guide, Xilinx, 2009.

[7] M. Bratina, Modeliranje nelineranih dinamičnih sistemov z metodami teorije informacij, doktorska disertacija, Univerza v Ljubljani, 2009.

[8] Spletna enciklopedija Wikipedia, gesla: backpropagation, brain, dostopno na: http://www.wikipedia.org, 2009.

[9] V. A. Predroni: Circuit Design With VHDL, MIT, Cambridge, 2004.

Priloga A.

Izvorna koda simulacije v okolju Matlab

```
%*************************************************************************
%* Simulacija navronske mreze
%* Datoteka : nnetnumHW5x6nobias.m
%*************************************************************************
clear all;          % pocisti spremenljivke
clc;
rand('seed', 0);    % inicializacija generatorja nakljucnih stevil
precWeight = 18;    % bitna preciznost utezi
precIO = 16;        % bitna preciznost vhodov in izhodov
precLUT = 4;        % bitna preciznost LUT tabele. Izhod je dolocen z precIO
rangeWeight = 4;    % interval utezi = [-4..+4]
rangeLUT = 2;       % interval LUT tabele = [-2..+2]

maxWeight = 2^(precWeight-1)-1; % razpon utezi
maxIO = 2^(precIO-1)-1;         % razpon vhoda in izhoda
maxLUT = 2^(precLUT-1)-1;       % stevilo LUT vzorcev
% ucni parametri
iteracij = 1000;    % stevilo iteracij
disturb =1/8;       % verjetnost pokvarjenega bita
eta = 1/64;         % ucni parameter
N0 = 30;            % stevilo vhodnih nevronov
```

```
N1 = 8;             % stevilo skritih nevronov
N2 = 10;            % stevilo izhodnih nevronov
S  = 10;            % stevilo vhodnih vzorcev
% nalozi osnovne matrike stevil (ucna mnozica)
load numData5x6;
% Inicializacija
W1 = (rand(N1, N0+1)-0.5)/rangeWeight;    % N1 x (N0+1) x precWeight bitov
W1 = floor(W1 * maxWeight);
W2 = (rand(N2, N1+1)-0.5)/rangeWeight;    % N2 x (N1+1) x precWeight bitov
W2 = floor(W2 * maxWeight);
% vhodni in izhodni vzorci
y0 = zeros(1, N0+1);
t  = zeros(1, N2);
% izvajanje mreze
v1 = zeros(1, N1);                         % N1 x precLUT bitov
y1 = zeros(1, N1);                         % N1 x precIO bitov
v2 = zeros(1, N2);                         % N2 x precLUT bitov
y2 = zeros(1, N2);                         % N2 x precIO bitov
% ucenje
d2a = zeros(1, N2);                         % N2 x precIO bitov
d2b = zeros(1, N2);                         % N2 x precIO bitov
d2 = zeros(1, N2);                          % N2 x precIO bitov
d1a = zeros(1, N1);                         % N1 x precIO bitov
d1b = zeros(1, N1);                         % N2 x precWeight bitov
d1 = zeros(1, N1);                          % N2 x precWeight bitov
dW2 = zeros(1, N2+1);                       % (N2+1) x precWeight bitov
dW1 = zeros(1, N0+1);                       % (N1+1) x precWeight bitov

tt = tOrig*1.6-0.8;                         % zelene vrednosti
tt = floor(tt * maxIO);

x = -rangeLUT:(2*rangeLUT)/(2*maxLUT+1):rangeLUT;
LUT = floor(tanh(1.4*x)/tanh(1.4*rangeLUT)*maxIO);  % LUT tabela
LUTd = floor((1-tanh(1.4*x).^2)*maxIO);             % LUTd tabela
ERR = [];                                     % opazovanje napake
for g = 1: iteracij
    % uvedemo napako v vzorec
    ins = abs(yOrig - (rand(size(yOrig)) < disturb))*1.6-0.8;
    ins = floor(ins*maxIO);
    err = 0;
    for s = 1: S
        % vzorci
        y0 = [ins(:,s)', maxIO];
        t = tt(:,s)';
        israngeOK(y0, precIO, 'y0 - 1');
        % procesiranje
        for n = 1: N1
            v1(n) = mult( W1(n,:), y0, precIO-1 + precWeight-1 - precLUT );
```

```
            v1(n) = rangeLimit( v1(n), precLUT);
            israngeOK(v1(n), precLUT, 'v1(n) - 2');
            y1(n) = LUT( v1(n) + maxLUT + 2 );
            israngeOK(y1(n), precIO, 'y1(n) - 3');
        end
        y1(N1+1) = maxIO;
        israngeOK(y1(N1+1), precIO, 'y1(N+1) - 4');
        for n = 1: N2
            v2(n) = mult( W2(n,:), y1, precIO-1 + precWeight-1 - precLUT );
            v2(n) = rangeLimit( v2(n), precLUT);
            israngeOK(v2(n), precLUT, 'v2(n) - 5');
            y2(n) = LUT( v2(n) + maxLUT + 2 );
            israngeOK(y2(n), precIO, 'y2(n) - 6');
        end
        % napaka
        e = t - y2;
        e = rangeLimit(e, precIO);
        israngeOK(e, precIO, 'e - 7');
        % izracuna delte
        d2a = LUTd( v2 + maxLUT + 2);
        israngeOK(d2a, precIO, 'd2a - 8');
        d2b = e;
        israngeOK(d2b, precIO, 'd2b - 9');
        [dummy1, d2] = mult(  d2a, d2b, precIO-1 );
        israngeOK(d2, precIO, 'd2 - 10 ');
        d1a = LUTd( v1 + maxLUT + 2);
        israngeOK(d1a, precIO, 'd1a - 11');
        for n = 1: N1
            d1b(n) = mult( W2(:,n)', d2, precIO-1 );
            d1b(n) = rangeLimit( d1b(n), precWeight );
            israngeOK(d1b(n), precWeight, 'd1b(n) - 12');
        end
        [dummy2, d1] = mult( d1a, d1b, precIO-1 );
        israngeOK(d1, precWeight, 'd1 - 13');
        % popravljanje utezi
        for n=1: N2
            [dummy3, dW2] = mult( d2(n)*ones(size(y1)), y1, precIO-1+log2(1/eta)...
                                - (precWeight-precIO) + log2(rangeWeight));
            W2(n,:) = rangeLimit( W2(n,:) + dW2, precWeight);
            israngeOK(W2(n,:), precWeight, 'W2(n,:) - 14');
        end
        for n=1: N1
            [dummy4, dW1] = mult( d1(n)*ones(size(y0)), y0, precIO-1+log2(1/eta) );
            W1(n,:) = rangeLimit( W1(n,:) + dW1, precWeight);
            israngeOK(W1(n,:), precWeight, 'W1(n,:) - 15');
        end
        err = err + e*e'/length(e);
    end
```

```
    ERR = [ERR; err];
end
%izrisi potek napake
plot(ERR);

%preverjanje
OK = 0;
for g=1:1000
    % uvedemo napako v vzorec
    ins = abs(yOrig - (rand(size(yOrig)) < disturb))*1.6-0.8;
    ins = floor(ins*maxIO);
    for s = 1: S
        y0 = [ins(:,s)', maxIO];
        t = tt(:,s)';
        % procesiranje
        for n = 1: N1
            v1(n) = mult( W1(n,:), y0, precIO-1 + precWeight-1 - precLUT );
            v1(n) = rangeLimit( v1(n), precLUT );
            israngeOK(v1(n), precLUT, 'v1(n) - 16');
            y1(n) = LUT( v1(n) + maxLUT + 2 );
            israngeOK(y1(n), precIO, 'y1(n) - 17');
        end
        y1(N1+1) = maxIO;
        for n = 1: N2
            v2(n) = mult( W2(n,:), y1, precIO-1 + precWeight-1 - precLUT );
            v2(n) = rangeLimit( v2(n), precLUT );
            israngeOK(v2(n), precLUT, 'v2(n) - 18');
            y2(n) = LUT( v2(n) + maxLUT + 2 );
            israngeOK(y2(n), precIO, 'y2(n) - 19');
        end
        OK = OK + (sum(tOrig(:,s)'==(y2>0))==N2);
    end
end
OK = OK / S / 1000
```

```
%************************************************************************
%* Simulacija navronske mreze
%* Datoteka : mult.m
%************************************************************************
function [weightedsum, scalar] = mult(w, x, rshiftbits)

scalar = w.*x;
%scalar
weightedsum = sum(sum(scalar));
%weightedsum
scalar = floor( scalar / 2^rshiftbits );
%scalar
```

```
weightedsum = floor( weightedsum / 2^rshiftbits );
%weightedsum
```

```
%*************************************************************************
%* Simulacija navronske mreze
%* Datoteka : rangeLimit.m
%*************************************************************************
function y = rangeLimit(x, prec)
maxVal =  2^(prec-1)-1;
y = max( min( x, maxVal ), -maxVal-1 );
```

```
%*************************************************************************
%* Simulacija navronske mreze
%* Datoteka : israngeOK.m
%*************************************************************************
function r=isRangeOK(val, prec, text)
m = 2^(prec-1)-1;
a = (val <= m);
b = (val >= -m-1);
r = sum(sum(a.*b))==(size(a,1)*size(a,2));
if r == 0
    disp(['Napaka: ' text]);
    keyboard
end
```