Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Bertalan Imre (MSc.), Németh Péter (MSc.)

FPGA ALAPÚ TÖBBSZINTŰ ÚTVONALVÁLASZTÓ ESZKÖZ PROTOKOLL FÜGGETLEN VEZÉRLÉSÉNEK MEGVALÓSÍTÁSA NYÍLT FORRÁSÚ KÖRNYEZETBEN

KONZULENS

Moldován István

BUDAPEST, 2011

Tartalomjegyzék

1 Ábrajegyzék

2 Bevezetés

Az internet korában egyre fontosabb elvárás a hálózati eszközökkel szemben, hogy minél hatékonyabban és gyorsabban működjenek. A gyártók célja az átviteli sebesség növelése, és a kommunikációs vonalon lévő késleltetés csökkentése. Ezzel együtt egy jó hálózati eszköznek egyszerre kell gyorsnak lennie, ugyanakkor a rugalmasság is fontos szempont, hiszen a lehető legtöbb hálózati átviteli, illetve útvonal választási protokollt támogatni szeretnék.

Az eszközök megvalósítását többféle módon is megközelíthetjük ezen szempontok szerint. Általános célú processzorok és a rajtuk futó szoftver segítségével rugalmas megoldást hozhatunk létre, melyek az újonnan kidolgozott protokollokat is támogatják, csupán egy szoftverfrissítésre van ehhez szükség. Napjainkban egyre szélesebb körben elterjedt megoldást nyújtanak a szoftver alapú útvonalválasztók, amelyek a boltokban beszerezhető hardverekre telepítve is működhetnek. Mivel ebben az esetben az útvonal-választással kapcsolatos döntéseket egy processzor hozza meg, nem pedig a hálózati kártyákon lévő helyi processzorok, így ez a megközelítés egy kevésbé hatékony megoldást eredményez, bár olyan szempontból előnyös, hogy a felhasznált interfészkártyák könnyen és olcsón beszerezhetőek. Gyors, párhuzamosított műveletvégzést a hardveresen megvalósított eszközök képesek végezni, de ehhez speciális műveletvégző egységek szükségesek, és ezek a processzorok nem támogathatnak több százféle protokollt és lehetnek olcsóak is.

Előnyös megoldás tehát az egyes részfeladatokat megfelelően szétválasztani szoftveres és hardveres részfeladatokra. Dolgozatunk egy ilyen rendszert mutat be, amiben a hardver és szoftver rész optimálisan szét van választva, azok szabadon konfigurálhatóak és lehetőséget nyújtanak új funkciók és modulok létrehozására, rendszerbe illesztésére.

3 Elméleti háttér, eddigi megoldások

Az útvonalválasztás hardveres és szoftveres feladatainak optimális szétválasztásával már eddig is sokan foglalkoztak. Ebben a fejezetben említést teszünk a legfontosabbakról, illetve megmutatjuk miben tér el a mi megoldásunk ezektől.

3.1 Igények, jövőkép

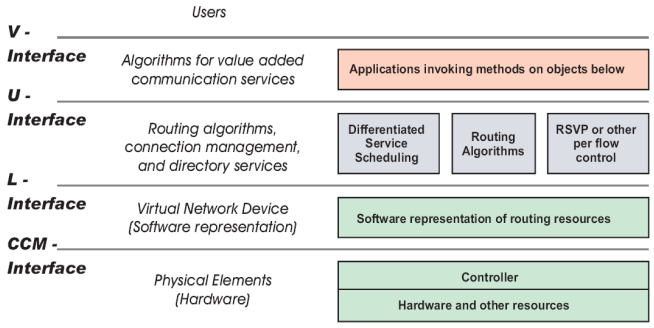
A jövő hálózati eszközeinek, több igénynek meg kell felelniük. Ezeket az igényeket három pontban összefoglalhatjuk. 1) Elég rugalmasnak kell lennie ahhoz, hogy az egyre több új szolgáltatást telepíthessük rá. 2) Nyílt és moduláris rendszer kell, hogy legyen. Ez elősegíti az eszközök árának csökkenését, illetve nem kerülhetnek gyártók monopol helyzetbe. 3) A rendszernek támogatnia kell a QoS szolgáltatásokat, így megfelelő átvitelt biztosíthat a valósidejű illetve multimédiás alkalmazásoknak. Egy nyíltforrású környezetben a vezérlési és továbbítási elemeket módszeresen szét kell választani. A továbbítási síknak vonalsebességen kell megbirkóznia a csomagok feldolgozásával. A vezérlő síknak pedig elő kell állítania a megfelelő paramétereket a működéshez. A leges legfontosabb szempont pedig, hogy a vezérlési sík és a továbbítási sík között szabványos interfésznek kell lennie, sőt mi több, a továbbításnak is szabványosnak kell lennie, így a vezérlés is szabványosan férhet hozzá a továbbítást végző erőforrásokhoz. így akár az is lehetséges, hogy a szétválasztott vezérlő és továbbító elemek, még ha más gyártótól is származnak, a szabványos kapcsolódások miatt együtt tudnak működni.

Ennek megfelelően a nyílt programozható hálózatokat kutatók előtérbe helyezték ezen szabványos interfész modelljének a kidolgozását.

3.2 Szabványosítás

Nyílt és szabványos interfész megvalósításának problémájára a P.1520 nyújtott először teljes értékű megoldást, amit az OPENSIG hozott létre 1998-ban **[6]**. A szabvány egy olyan hálózat vezérlési rendszert ír le, ami meghatároz egy interfészt a vezérlés és a menedzsment funkciók között. Nem határoznak meg konkrét protokollt, helyette egy olyan programozható felületet biztosítnak ezek az interfészek, mint egy operációs rendszer alkalmazásprogramozási felülete. A modell meghatározza a hálózat

tervezési rész legfontosabb rétegeit és bemutatja a virtualizáció folyamatát. A P.1520 egyes régetek közötti interfészei az **1. ábraán** tekinthetőek meg.



1. ábra Az IEEE P.1520 rétegek közti interfészei

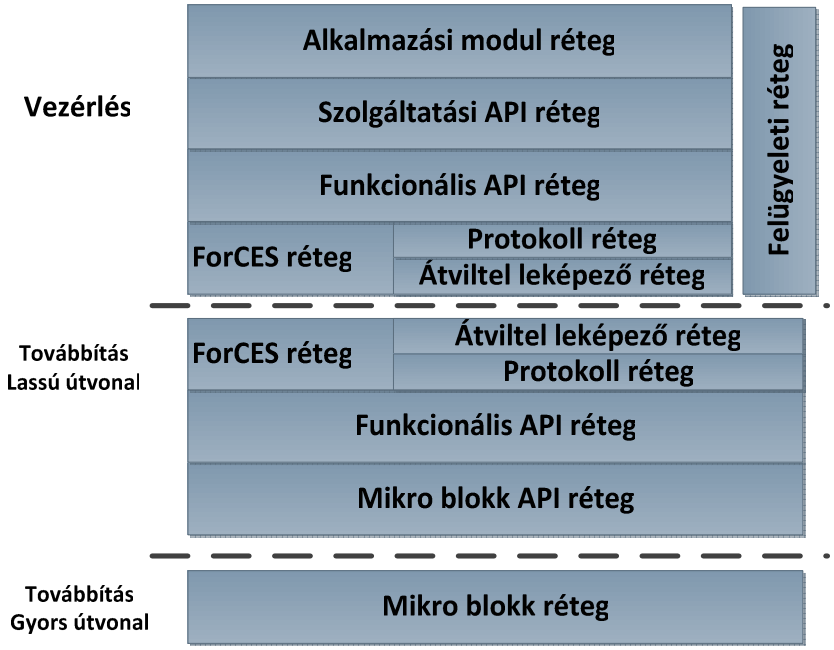
Az interfészek és erőforrások szabványosítási folyamata során jött létre például a Multi Protocol Over ATM (MPOA), General Switch Management Protocol (GSMP), és a Forwarding Element Control Element Separation (ForCES). Ezek közül a ForCES – ami az IETF útvonalválasztással foglalkozó csoportjainak egyike – érte el a legkiemelkedőbb eredményeket.

A *ForCES keretrendszer* **[1][2]** útvonalválasztó architektúrájában különválasztották az adattovábbítási részt (Forwarding element, ami a csomagtovábbításért felel) és a vezérlési részt (Control element ami feldolgozza a vezérlőprotokollok üzeneteit). Az adattovábbítási részt ASIC-kel, hálózati-processzorral vagy egy általános célú processzorral felszerelt eszközzel valósítják meg a nagy sebességű csomagtovábbítás érdekében, míg a vezérlési feladatok elvégzését tipikusan egy általános célú processzorra bízzák a vezérlő protokollok komplexitása miatt. A vezérlési és a továbbítási sík közötti kommunikációt pedig a szabványos ForCES interfészen keresztül oldják meg. Mára a ForCES csoport befejezte a ForCES-re vonatkozó követelmények (RFC3654) és a ForCES keretrendszer kidolgozását (RFC3746).

Létre jött egy a ForCES alapján implementált útvonalválasztó a ForTER **[4]**, ahol a továbbítási és vezérlési sík szétválasztása a megvalósítás kulcseleme. A köztük folyó kommunikáció a ForCES protokoll alapján történik, ennek a protokollnak a segítségével irányítja a vezérlési réteg a továbbítást. A továbbítási síkot tovább szabványosítja a ForCES. A sík elemeit logikai funkcionális blokkoknak nevezik el. Az adatutakkal összekötött blokkok alkotják a továbbítási síkot. A blokkok egyes jellemzőinek beállítását, illetve lekérdezését elnevezték konfigurációnak. A vezérlési

réteg feladata, hogy a funkcionális blokkok egyes attribútumait, illetve azok topológiáját megváltoztassa a ForCES protokoll segítségével. Így tud a vezérlési sík különböző IP alapú szolgáltatásokat nyújtani a továbbítás irányításának segítségével.

A ForTER egy többszintű architektúrát használ, ami a **2. ábraán** látható.



2. ábra ForTER architektúra rétegszerkezete

Amint látható, az architektúra továbbítási és vezérlési szintekre osztható. A továbbítási rész további két szintre, egy gyorsra és egy lassúra tagolódik. A gyors útvonal végzi a csomagfeldolgozást és továbbítást, ennek vonalsebességgel kell működnie. Ennek az adatútnak a dinamikus irányítását végzi a vezérlési szint a lassú adatútvonalon keresztül.

Az alkalmazási modul rétegben futnak az útvonalválasztó protokollok, és például a QoS modulok. A moduláris megvalósítás lehetővé teszi, hogy más gyártók által fejlesztett modulokat használjunk a ForCES rendszerben, például Zebra útvonalválasztó modult, vagy AgentX++, és NetSNMP modult, az SNMP protokoll működtetéséhez. A szolgáltatási API réteg hidalja át a felsőbb szoftveres, illetve a ForCES specifikus réteget, vagyis kiszolgálórétegként üzemel a felsőbb szoftveres modulok és a hardveres funkcióblokkok között. Tehát ez a réteg elrejti a ForCES funkciókat a felsőbb rétegek, a más gyártók által fejlesztett szoftverek elől. Mivel ez a réteg az alsóbb szinten lévő funkcionális rétegnek megfelelően készítették el, egy sor függvényhívási lehetőséget és adatstruktúrát biztosít a felsőbb szintű folyamatok, például az útvonalválasztó vagy SNMP szoftverek számára. A funkcionális réteg

tartalmazza a funkcionális blokkokhoz tartozó függvényhívásokat és adatstruktúrákat, amiken keresztül attribútumaik állíthatóak. Ezen kívül a ForCES réteg ezeket a függvényeket hívja meg, ha protokoll üzenetek érkeznek a továbbítási vagy vezérlési réteg felől.

A ForCES réteg két alrétegre bontható. A protokoll réteg felelős a funkcióblokkokban történő folyamatokhoz tartozó ForCES üzenetek létrehozásáért és továbbításáért.

A továbbítási leképező réteg fordítja le a ForCES üzeneteket más közegen például ATM-en vagy Etherneten való átvitelhez.

A ForCES modellben a továbbítási sík funkcionális blokkjai a továbbítási sík fizikai erőforrásaiból szintetizálódnak, mely erőforrások a **2. ábraán** látható mikroblokk elemei. A mikroblokk API teszi lehetővé a mikroblokk vezérlését. A továbbítási réteg lényege, hogy ennek a blokknak az elemei programozhatóak.

A **2. ábra** rétegzett, moduláris modellje segítségével lehet a ForCES rendszer nyílt, szabványos, programozható és moduláris.

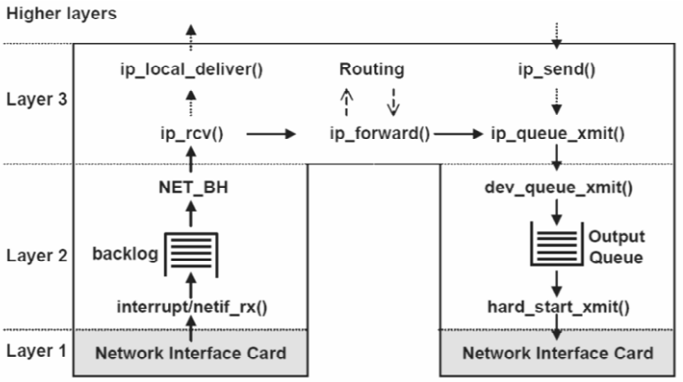
Ezeknek a követelményeknek a Linux *Netlink protokollja* **[3]** is megfelel, amelynek feladata az információtovábbítás a kernel és a felhasználói szintű folyamatok között, amihez egy socket alapú kapcsolatot használ fel. Az RtNetlink ezen belül is a routing tábla olvasására, módosítására ad lehetőséget. Egy ilyen NETLINK_ROUTE socket-en keresztül lehetőségünk van az útvonalak, IP címek, link paraméterek és egyéb paraméterek vezérlésére. Nagyon sok ma is létező és fejlesztés alatt álló protokollt meg lehet valósítani felhasználói szinten, amelyeknek szüksége van egy csatornára, amin az útvonalválasztást módosíthatják és értesülhetnek a mások által okozott változásokról. Ezek közé sorolható a dinamikus útvonalválasztó protokollok, mint a RIP (Routing Information Protocol), az OSPF (Open Shortest Path First), az EGP (Exterior Gateway Protocol), ad-hoc hálózati protokollok, amik peer-to-peer kommunikációt használnak a különböző célállomások között.

Az RtNetlink multicast socket interfészt is biztosít, amire feliratkozva értesítéseket kaphatunk a routing megváltozásáról, amellyel megoldható, hogy nem csak egy folyamat képes a tábla olvasása, módosítása. Ez a hasznos funkció kihasználható a routing változásának megfigyelésére. További előnye a protokollnak, hogy bármilyen útvonalválasztó protokollal képes együtt működni, ami RtNetlinket használ a táblamódosításhoz. Mivel a manapság leggyakrabban használt protokollok támogatják

ezt a funkciót, így könnyen megoldható az integrálódásuk a Linux útvonalválasztó architektúrába.

3.3 Csomagtovábbítás Linux környezetben

A Linuxban a csomagtovábbítást felfoghatjuk úgy, mint egy dobozt, ami útvonal választóként viselkedik **[10]**. Ennek a képzeletbeli eszköznek a sémája látható a **3. ábrán**. Amint az látható, a beérkező csomagot a hálózati kártya egy megszakítás-kéréssel jelzi. A kérés hatására egy rutin a teljes csomagot egy socket buffer struktúrába (sk_buff) másolja, innen pedig a backlog sorba, ami a belső központi sor, ahová minden bejövő csomag kerül.



3. ábra A Linux útvonalválasztó rétegei

Pontosabban a backlog egy sk_buff struktúrákból álló lista, amihez szabványos makrókon keresztül lehet hozzáférni (elemeket hozzárakni/elvenni). Például az elem hozzáadáshoz a netif_rx() függvény hívódik meg.

E fölött a réteg fölött működnek a 3-mas rétegbeli protokollok, amelyek az útvonalválasztást végzik. A rendszer még magasabb szintekre adja föl a csomagokat, amikor ő maga küld, illetve fogad információt.

A Linux ezen kívül számos haladó szintű szolgáltatást nyújt hálózati adminisztrációhoz és forgalom szabályozáshoz. Ezek a Linux Advanced Routing and Traffic Control (LARTC) szolgáltatások mind a Linuxba vannak integrálva, vagyis az operációs rendszer önmaga, ha rendelkezünk egy vagy több hálózati interfésszel, szoftver útvonalválasztóként is képes működni. Sőt mi több, még bizonyos biztonsági feladatokat is rábízhatunk.

A Linuxban a NetLink használatával a programok információk cserélhetnek a kernel különböző elemeivel, így vezérleve azok működését. A hálózati protokoll kezelő programok az RTNetLinket használva figyelhetik a kernel hálózatkezelő része elemeit,

mint például a hálózati interfészeket vagy a routing táblát. Nagyon sok létező és fejlesztés alatt álló protokollt lehet fehasználói szinten implementálni, amivel csökkentjük a kernel kódjának bonyolultságát és kihasználhatjuk a felhaználói szint fejlesztői eszközeinek előnyeit. Ezek az implementációknak szükséges, hogy módosíthassák a routingot és értesüljenek arról, ha változás történik.

Az RTNetLink egy socket alapú kommunikációt biztosít a felhasználói szint és a kernel között **[9]**. Egy ilyen komminukáció felépülése és rajta az adatküldés az alábbi sorrendben történik:

1. Socket nyitás
2. Socket hozzácsatolása a hívó folyamathoz
3. Üzenetküldés a célállomásra
4. Üzenetfogadás a célállomástól
5. Socket bezárása

Az egyes lépések elvégzéséhez az előre definiált függvényeket használhatunk.

Üzenetküldéskor, az útválasztással kapcsolatos üzeneteket a *sendmsg()* függvénnyel tudjuk elküldeni a kernelnek, az üzenetek fogadására pedig a *recv()* függvény használható, ami egy karaktertömbbe írja be a socketről érkező adatokat

Ha a folyamat nem kíván több adatot fogadni/küldeni, akkor a socketet a *close(int fd)* függvénnyel zárjuk le.

Az RTNetlink minden egyes művelete három lehetséges beavatkozást tesz lehetővé: hozzáadás *(NEW)*, törlés *(DEL)* és lekérdezés *(GET)*.

Általános hálózati környezet befolyásoló szolgáltatások:

- Adatkapcsolati szintű interfész beállítás: *RTM_NEWLINK, RTM_DELLINK* és *RTM_GETLINK*.
- IP szintű interfész beállítás: *RTM_NEWADDR, RTM_DELADDR* és *RTM_GETADDR*.
- Hálózati réteg routing tálba beállítás: *RTM_NEWROUTE, RTM_DELROUTE* és *RTM_GETROUTE.*
- Adatkapcsolati és hálózati réteg címzés összerendelését segítő gyorsítótár: *RTM_NEWNEIGH, RTM_DELNEIGH* és *RTM_GETNEIGH*.

Az RTNetLinken kérés-válasz formában cserélünk információt, melyek üzenetstruktúrák sorozatából állnak.

Kérés esetén struktúrát a hívó folyamat tölti fel, míg a válasznál a kernel. Az egyes kérések elején a *nlmsghdr* struktúra található, ez határozza meg, hogy milyen típusú RTNetLink műveletek vannak a kérés további részében. Ezek típusától függően a hívó folyamat a műveletfejlécnek nevezett struktúrákból egyet vagy akár többet kell az üzenetben elhelyeznie. Ezekből a fejlécekből a dolgozatunkhoz az *rtmsg* nevűt kellett használnunk, ami a routing tábla sorainak lekérdezésére és megváltoztatására használható.

A fogadott adat is üzenetstruktúrák sorozatából, amit a folyamatnak végig kell olvasnia, és azonosítania kell az egyes üzeneteket.

3.4 Hardver-szoftver együttműködési példák

Az Open vSwitch egy többszintű virtuális kapcsoló, ami a programozható kiterjesztéseivel igen erős hálózat automatizálást tesz lehetővé, emellett támogatva a hagyományos vezérlési interfészek és protokollokat is. Eredetileg kompatibilis lenne a modern kapcsoló chipekkel, hogy képes legyen a csomag feldolgozást rájuk bízni, de ezt a funkciót a valóságban kevés helyen használják.

A GNU Zebra **[7]** egy olyan szabad felhasználású szoftver, ami a TCP/IP alapú útvonalválasztási protokollokat menedzseli, szemben a hagyományos, monolitikus architektúrákkal, melyek leveszik a routing funkciók feldolgozását jelentette terhet a CPU-ról és speciális ASIC chipeket használnak e célra. A Zebra szoftver teljes modularitást nyújt, és az RtNetlinket használja a routing tábla dimanikus frissítéséhez.

Az XORP egy bővíthető router platform (eXtensible Open Router Platform) **[8]**. Weblapjuk szerint céljuk egy olyan szabad felhasználású router platform fejlesztése, ami stabil és elég sok funkció tartalmaz, és emellett flexibilitásával és bővíthetőségével támogatja a hálózati kutatásokat. Jelenleg a legfontosabb IPv4 és IPv6-os útvonalválasztási protokollok vannak implementálva, és egy egységes interfészt, amin konfigurálhatóak. A Vyatta OFR, ami egy flexibilis router implementáció ezen az XORP platformon alapszik.

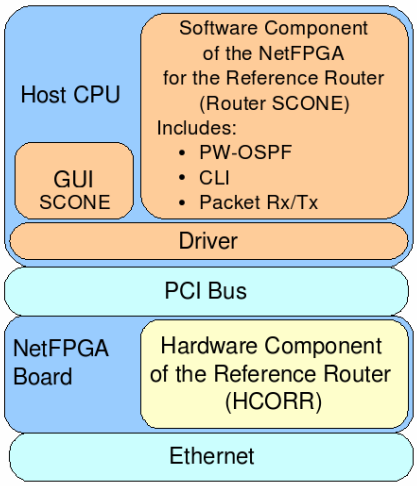
A szoftveres megoldások mellett születtek hardveres úton megvalósított útvonalválasztók is. A NetFPGA routerek módosított OSPF protokoll szerinti útvonalválasztással dolgoznak, de a hardver csak ezzel az egy protokollal tudnak együttműködni. Léteznek hardveresen megvalósított kapcsolók, mint például az OpenWRT routerek, viszont ezekben nincs létrehozva kapcsolat a vezérlési résszel (konkrétan a Spanning Tree Protocol támogatása), habár a VLAN funkciók

implementása már megoldott. Egy másik egyre jobban terjedő megoldás az OpenFlow protokoll használata. Megvalósításának célja viszont eltér az eddigiektől: működésében a vezérlés teljesen külön van választva az adattovábbítástól, aminek működése teljesen el van rejtve a rendszer elől.

A bevezetőben leírt elgondolás ugyan újdonságnak számít, de találkozhatunk hasonló megoldásokkal, melyek tehermentesítik a CPU-t, és hardveres feldolgozást biztosítanak, közben pedig egy operációs rendszerrel kommunikálva a rendszer végzi el a hardver beállításait.

Ilyen megoldást nyújt például a netFPGA. A netFPGA rendszert alapvetően oktatási és kutatási célokra, a hálózati hardverek tervezési módszereinek vizsgálatára fejlesztették ki.

A netFPGA referencia architektúrájának sémáját a **4. ábraán** láthatjuk.



4. ábra netFPGA referencia architektúra

A legalsó szinten történik az Ethernet kommunikáció fizikai része. Ennek megvalósítását végzik az FPGA-ban lévő interfészvezérlők, illetve a jelek előállításáért felelős a Broadcom átviteli áramkör.

A HCORR szinten találhatóak az FPGA-ban megvalósított logikai elemek. Ez ebben az esetben egy pipeline.

A **4. ábraán** látható architektúra SCONE (Software Component Of NetFPGA) szelete látja el az útvonalválasztó szerepét. Ez a szoftver kezeli az IPv4 továbbítást az ARP és ICMP üzeneteket. 23-mas és 8080-as porton, telneten és webes felületen keresztül lehet a routert konfigurálni. Ez valósítja meg a routing protokollt, ami az OSPF egy szűkített változata a PW-OSPF. A SCONE tartja karban a hardveren a MAC és IP címekből felépülő útválasztó táblák másolatát, és az ARP táblát.

A továbbításhoz a routerben a csomag MAC címét IP címét és ezek összerendelését kell megtalálni az ARP MAC illetve IP táblában, hogy a next-hopot tudjuk. A SCONE szoftveresen építi fel az útvonalválasztó táblázatokat, a statikus és dinamikus útvonalakból, illetve az ARP táblát, ezután átmásolja ezeket a hardveres táblákba. Ha az általunk az interfészekre beállított IP címekre küldött csomagokat is meg akarjuk kapni, akkor ezeket az IP címeket is el kell helyezni a hardveres táblában. Ezeket a szoftver szintén IOCTL hívásokkal állítja be a hardver regisztereiben.

A csomagok továbbítása tehát a beállított táblák után már független a számítógépen futó programtól, viszont a táblázatokat nem maga a Linux állítja elő, hanem egy saját program. Így módosítások nélkül, csak meghajtó telepítéssel és a kártya behelyezésével nem használható a hardveres gyorsítás, csak az implementált netFPGA SCONE alkalmazás segítségével, ami behatárolja a használható útvonalválasztó protokollok használatát. A hardvert ezen kívül interfész kártya üzemmódban is lehet használni. Ekkor pedig azért nem nyertünk semmit, mert mind a táblák kezelését, mind a továbbítást a Linux végzi, így a hardver tehermentesítése sem jelenik meg.

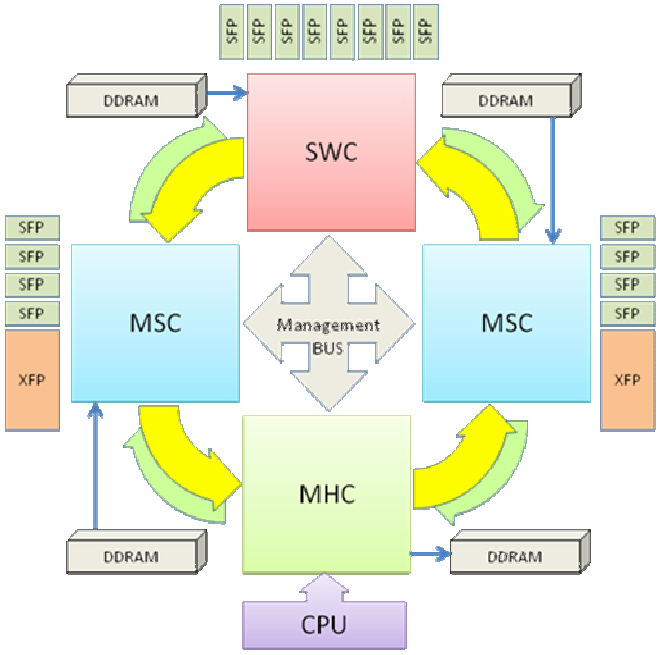
Ezzel szemben az általunk tervezett rendszerben, mind a vezérlési rész, mind az adattovábbítás szabadon megváltoztatható, amivel lehetőséget biztosítunk új útvonal-választási architektúrák tervezésére, vizsgálatára. Az általunk nyújtott interfész lehetőséget biztosít egy olyan vezérlés megvalósításához, ami felhasználja és együttműködik a Linux alatt futó eddig elkészült és jövőbeli protokollokkal. Mindemellett az adattovábbító rész fejlesztésére is mód van az FPGA nyújtotta rugalmas környezetnek köszönhetően. Nagy előnye még, hogy a feldolgozási sebesség és a memória kapacitása független a vezérléstől.

4 Alkalmazott eszköz: a C-Board bemutatása

A C-board egy fejlesztési fázisban levő FPGA alapú hálózati eszköz, ami a nagysebességű adatfeldolgozáshoz, továbbításhoz nyújt kiváló alapot. Lehetőség van adatok küldésére, fogadására, a bejövő forgalom monitorozására, Ethernet szintű kapcsolásra és IP szintű útvonalválasztásra és egyéb folyamatok vizsgálatára, mindezt igen nagy, akár 10 Gigabit/s adatátviteli sebesség mellett. További előnye, hogy az FPGA mellett található benne egy Linuxot futtató PC is, ami képes a komplex számítás igényes feladatok megoldására.

A feladatunk megvalósításához a tanszéken elérhető C-board-ot (felülnézeti képe a **[10][C]** függelékben látható) használtuk.

4.1 A C-board felépítése



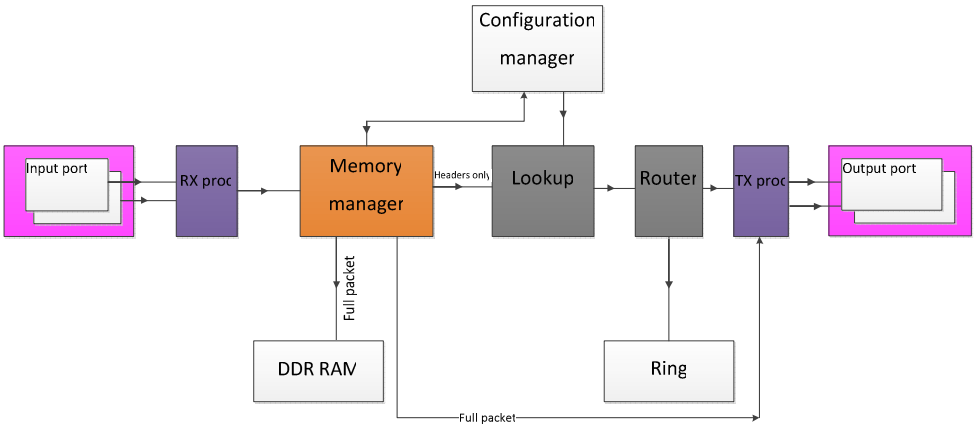
5. ábra A FPGA-k elhelyezkedése a C-boardon

A kártyán négy darab Xilinx Virtex 5 típusú FPGA van gyűrűbe kötve, ahogy ez az **5. ábraán** látható. Az egyes FPGA-k külön feladatot látnak el, a Management Host Core (MHC) a kártyán található PC-vel és így a Linuxszal való kommunikációért felel, a Switch Core (SWC) felel a csomagok útvonalválasztásáért, a másik kettő pedig általános feladatokat (be- és kimenetek kezelése, információtovábbítás) valósít meg. A kártyán tizenkét darab 1 Gigabites, két darab 10 Gigabites Ethernet port található, ezen

felül 2 ATM és 2 POS port és a PC-vel PCI Express buszon keresztül kommunikál. Emellett mindegyik FPGA-hoz tartozik egy-egy DDR2 RAM. A feladatunk ebben a rendszerben az MHC és SWC FPGA módosítása oly módon, hogy azok a Linuxszal együttműködve egy optimálisan elosztott rendszerként működjenek.

4.2 Csomagtovábbítás a C-board-on

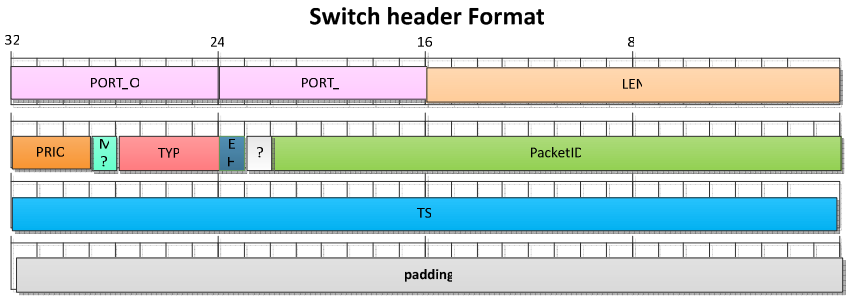
A Cboarddal együtt adott volt a kártyán belüli adatáramlást leíró általános csomagtovábbítási keretrendszer, aminek egyszerűsített felépítését a **6. ábra** mutatja. (a részletes felépítés az **[10][A]** függelékben tekinthető meg)



6. ábra A csomag útja az FPGA-ban

Ebből a rendszerből több egység is adott volt, mint például az *RX processzor*, amelynek a fő a feladata a bemeneti portok kezelése és a Board-on belül a csomagot azonosító Switch Header (felépítését **7. ábra** mutatja) elhelyezése az adat elejére.

A processzor kimenő sorából az adatok a *Memória menedzserbe* kerülnek, ahol a Header *Packet ID* mezejének kitöltése történik, ami az egész kártyán belül a letárolt csomagot azonosítja. Az eszköz sajátossága, hogy a teljes csomag a DDR RAM-ban tárolódik, a további modulokhoz csak a csomag fejlécének egy része (ez állítható) jut el. A *Lookup modul*, aminek a továbbfejlesztése is a feladatunk részét képezte, eredetileg egy adatkapcsolat szintű kapcsolót valósított meg. Ez a modul tölti ki a Switch Header kimeneti port mezejét, miután a csomag fejléce a *Router modulba* kerül, ahol eldöntődik: egy másik FPGA-ra kell küldeni a Ring menedzserrel vagy egy lokális kimeneti porton kell kiküldeni.



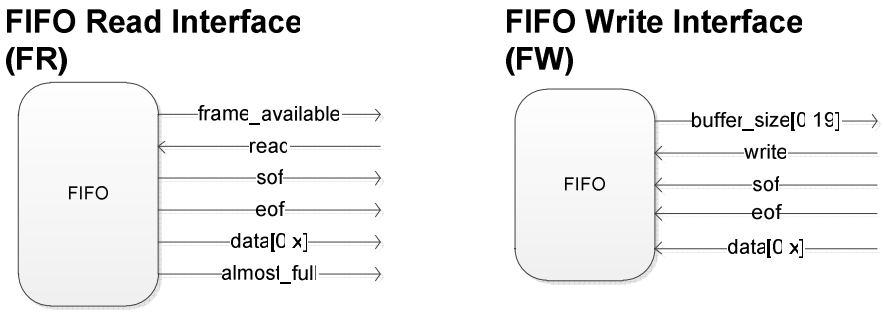
7. ábra Switch Header felépítése

Ha egy helyi port a kimenet, akkor a fejléc *TX processzorba* kerül, aminek feladata a fejléchez tartozó csomag lekérdezése a Memória menedzsertől, a kimeneti port funkciók megvalósítása és a Switch Header eltávolítása. A csomag ezután a kimeneti FIFO-ban várakozik a kiküldésig.

A fent leírt modulok közül egyes dinamikusan változó paraméterekkel dolgoznak. Ezeknek a paramétereknek vagy a paraméterek lekérésének az eljuttatása a megfelelő modulhoz a *Configuration menedzsment* feladata. Ezen kívül ez az egység felel a CI interfész vezérléséért, amit minden hozzá kapcsolódó modulnak támogatnia kell. Ezt a modult még implementálni kellett.

4.3 Generic FIFO bemutatása

Ahogy az előző fejezetben látható volt az eszközben fontos szerepe van a FIFO-nak, így fontosnak tartottuk írni róla pár szót. Egy-egy nagyobb egység, vagy egy komplexebb modul be- és kimenete is egy ilyen egységhez kapcsolódik. A bemeneti interfészét FW-nek (FIFO Write), kimenetét FR-nek (FIFO Read) hívják, melyeket a **8. ábra** szemléltet. Működésére jellemző, hogy az adatok beírása és kiolvasása (órajelre ütemezve) a *write* és *read* jelekre történik. Egy-egy adat elejét és végén az *sof* (Start Of Frame) és *eof* (End Of Frame) jelek jelzik úgy, hogy egy órajelig az 1 értéket veszik fel. Az adatbusz (*data*) állítható szélességű, a kártyán a be- és kimenet szélessége azonos érték esetén 64 vagy 128 bit eltérő szélesség esetén 8-64, 64-128 bit. Emellett van még egyéb, a működést segítő jele is. A kimeneti interfész *frame_available* jele arról tájékoztatja a hozzá csatlakozó modult, hogy a FIFO-ban adat van, a bemenet *buffer_size* busza pedig a FIFO telítettségét jelzi. A *frame_available* jel két órajel alatt áll be, ezt a hozzá kapcsolt modul megvalósításánál figyelembe kell venni.



8. ábra A Generic FIFO interfészei

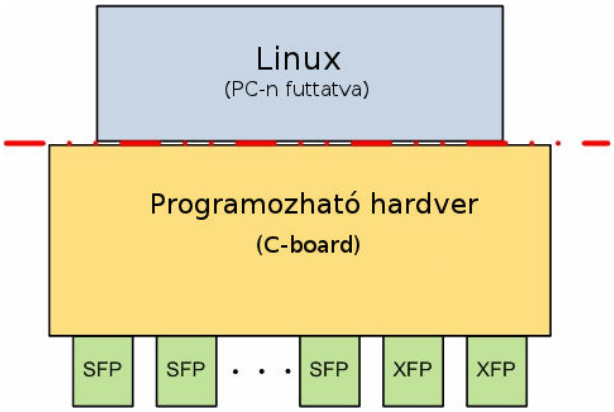
5 Tervezés és Megvalósítás

5.1 Követelmények

A C-board Ethernet és IP szintű hardveres csomagtovábbítást biztosít, emellett található rajta egy Linuxot futtató PC is. TDK dolgozatunk célja egy olyan rendszer megvalósítása, amiben a C-board általános csomagtovábbítási rendszere és a Linuxon futó hálózati protokollok hatékonyan együttműködnek.

A megoldásnak az alábbi követelményeknek kell megfelelnie:

- A beérkező csomagok Ethernet és IP szintű kapcsolása, útvonalválasztása.
- A vezérlési csomagok továbbítása a Linuxnak, és a Linuxtól kapott csomagok kiküldése a helyes interfészen.
- A hálózati interfészek láthatóvá tétele a Linux számára.
- A vezérlési rész változásainak érvényesítése az adattovábbítási részen gyorsan és megbízhatóan.
- A fent említett célok megvalósítása úgy, hogy a lehető legtöbb Linuxon futó hálózati protokollal együttműködjön azok megváltoztatása nélkül is.



9. ábra Az optimális interfész keresése a programozható hardver és a Linuxon futó protokollok között

Adattovábbítás szempontjából ügyelnünk kellett arra, hogy a kapott eszköz moduláris felépítésű (ha egy elemet egy újabbra cserélek, akkor azzal is tökéletesen működnie kell a rendszernek) így a keretrendszerben a Lookup-ot is úgy kellett módosítani, hogy egy új protokoll szerinti szűrés, csomagtovábbítás gond nélkül illeszthető legyen a rendszerbe, ezért praktikusnak tűnt egy olyan felépítést, *top layert* kitalálni, ami szintén moduláris felépítésű. Létre kellett hozni egy ilyen moduláris Layer 3 szintű útvonalválasztó egységet és illeszteni kellett a már meglevő kapcsoló

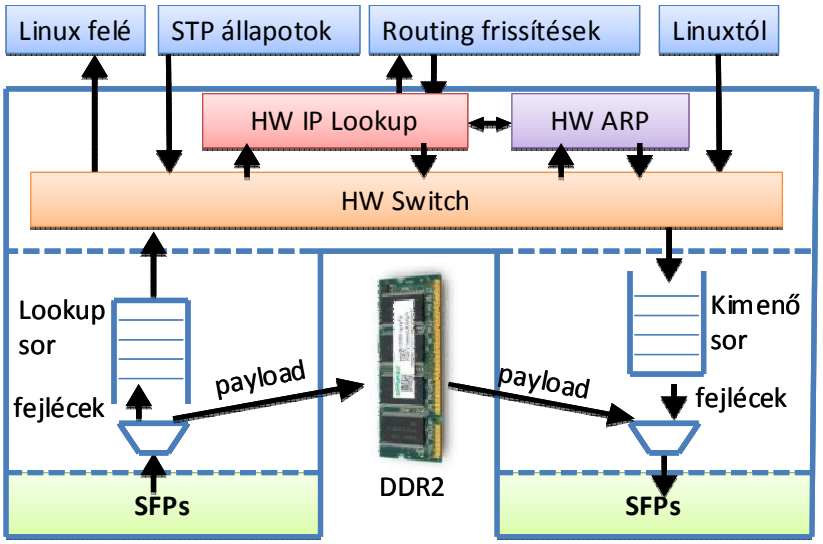
modulhoz. Meg kellett oldani a vezérlési rész, vagyis a Linux-tól jövő információ fogadását, tárolását, vagyis fel kellett készíteni a Lookup modul egységeit az útvonal-választási információk kezelésére. Mivel az adattovábbító és a vezérlési rész nem egy FPGA-n kapcsolódik össze, ezért létre kellett hozni a gyűrűn való kommunikációhoz szükséges egységeket, amik továbbítják a vezérlés szempontjából szükséges információt a Linuxnak és eljuttatják az adattovábbítási résznek a működéséhez szükséges paramétereket.

Nem utolsó sorban követelmény volt még, hogy az egyes modulok a lehető leggyorsabban, minél kisebb késleltetéssel működjenek, és a lehető legkevesebb erőforrást igényeljék (különösképpen a FIFO-k számát tekintve), és ezeket a legjobban kihasználják.

Mint már azt az összefoglalóban említettem egy olyan rendszer működését terveztük meg, amely képes a csomagtovábbítás egyes feladatait szétválasztani hardveres és szoftveres részfeladatokra. A megvalósítás hasonló, mint a netFPGA elgondolás esetében, viszont a mi esetünkben a cél úgy megoldani a működést, hogy a Linux saját interfészeiként ismerje föl az alatta lévő C-boardot, és úgy is kezelje, mintha az egy egyszerű több portos interfész lenne. Ehhez azonban szükség van egyrészt olyan driver és olyan hardver megvalósítás létrehozására, ami ezt lehetővé teszi.

5.2 A szoftveres-hardveres feladatok optimális szétválasztása

A mi megoldásunk célja, hogy a 3.3 bekezdésben taglalt Linuxon megvalósított elgondolást megváltoztatva, egyes harmadik rétegbeli műveleteket hardveres szinten tudjunk elvégezni. Ehhez szükséges a csomagtovábbító keretrendszer, ami megvalósítja a bufferelést, illetve az útvonalválasztást, a hálózati címek adatkapcsolati (MAC) címmé történő feloldását, illetve a kapcsolást a MAC címek alapján. Így az előbbi modell módosul a **10. ábraán** látható módon. Az ip_forward() folyamat átkerül a hardverbe, és kiegészül egy kapcsolóval is. Vagyis olyan feladatok, amiket tipikusan hardver segítségével szokás elvégezni a gyors és párhuzamosított műveletvégzés érdekében, a megfelelő helyen történnek elvégzésre.



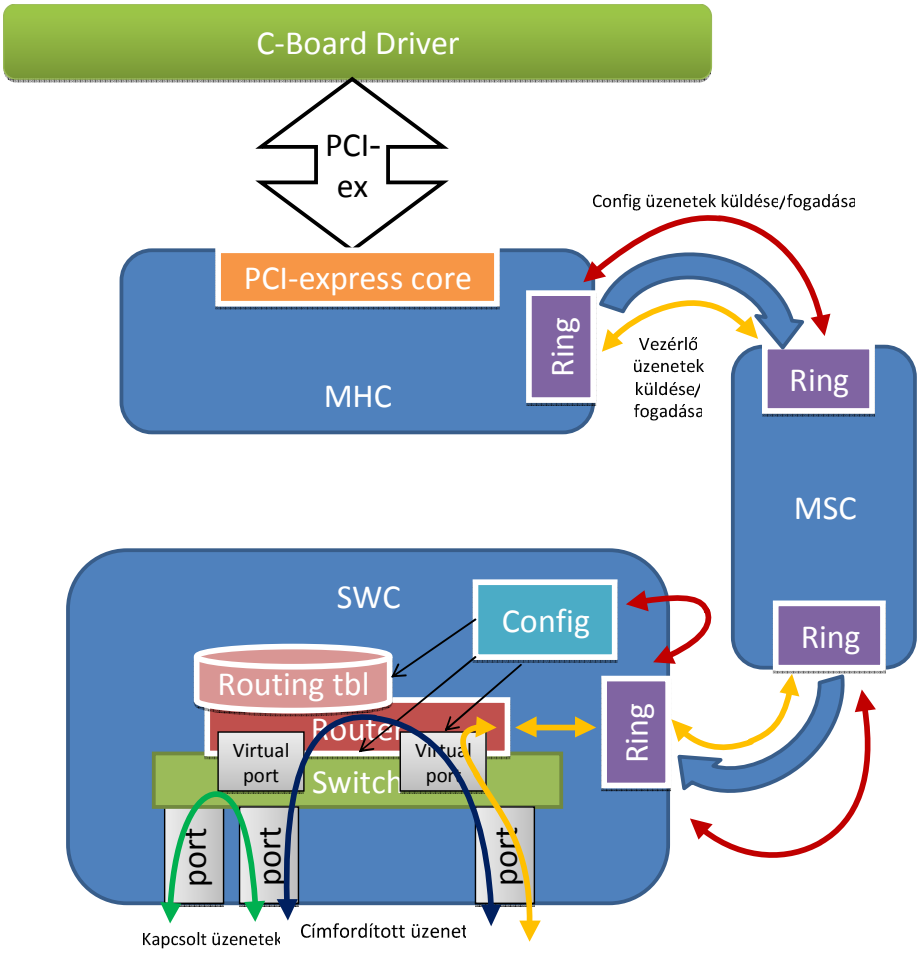
10. ábra Linux router hardveres támogatással

A hardver meghajtójának azonban meg kell oldani a táblák konzisztensen tartását a Linux táblázataival, illetve a lehetőséget továbbra is meg kell adni a Linuxnak, hogy csomagot küldhessen/fogadhasson. Tehát a Linux számára láthatóvá kell tenni a hardveres interfészeket, és lehetővé kell tenni a Linux kommunikációját ezeken keresztül, hogy a meglévő útvonalválasztó protokollok módosítás nélkül fussanak és menedzselhessék a routing táblát. Ugyanakkor az összes vezérlő üzenet ezeken az interfészeken el kell jusson a routing protokollokhoz, míg az általuk küldött csomagok ki kell tudjanak menni a megfelelő interfészeken.

Mivel a csomagtovábbítást a hardverre bízzuk, az útvonalválasztó táblák módosításai haladéktalanul el kell jussanak a hardverre is. Tehát a Linux oldalt ki kell egészíteni egy olyan funkcióval, amely az interfészek állapotainak változását és az útvonalválasztó tábla módosításait azonnal leküldi a hardver részére. Mindezt a legkönnyebben egy saját eszközkezelő szoftver segítségével valósíthatjuk meg, amely létrehozza az interfészeket, megvalósítja a csomagküldést/fogadást, és ugyanakkor nyomon követi a változásokat az RtNetlink segítségével és tükrözi őket a hardverre. Így a Linux egy szabványos meghajtóként látja a hardvert, és a felsőbb szintek ennek megfelelően kezelik.

Ilyen módon a szoftveres feladatok elvégzését a processzorra tudjuk bízni, és kellően rugalmas marad a rendszer, hiszen a protokollok rugalmas, szoftveresen megvalósított környezetben futnak.

5.3 A firmware környezet



11. ábra A csomagok útja a C-boardon belül

Ebben a fejezetben egy átfogó képet szeretnénk adni a csomagok útvonaláról a C-boardon belül. Attól függően, hogy a csomag kinek szól, eltérő utat tesz meg a rendszerben, ahogy ez a **11. ábraán** is megfigyelhető. A lehetséges útvonalak az architektúrában:

- Kapcsolás a portok között
- Címfordítás a portok között
- Gateway port és a Linux PC között
- Config interfészen keresztül

Azonos alhálózaton levő gépek között a küldött csomagtovábbítást a kapcsoló modulig végzi, ahogy ezt a fenti ábrán látható *zöld nyíl* mutatja. Ha eltérő hálózatokra kapcsolódó gépek között küldünk adatot, akkor már routeren keresztül mennek a csomagok (az ábrán *kék nyíllal* jelölt úton), melyek továbbítása az útvonalválasztási tábla szerint történik, és ez a modul végzi hálózatok közötti címfordítást is. Abban az

esetben, ha egy vezérlési csomagot észlel a router (a csomag neki lett címezve), akkor fel kell küldeni azt a Linuxnak. A csomag a gyűrű menedzser segítségével (a **11. ábrán** a sárga nyíl jelölte útvonalon) az SWC-ről átkerül az MSC FPGA-t érintve az MHC-ra, ahol a PCI-Express illesztőn keresztül megkapja a Linuxon futó C-board driver. Ugyanezt az útvonalat járják be a Linux felől küldött csomagok is. A konfigurációs csomagok küldése/fogadása egy speciális esetnek tekinthető, mivel itt nem egy külső fizikai interfészen kapjuk vagy kell kiküldenünk az üzenetet. Az adatátvitel tipikusan a Linux és egy a rendszerbeli modul között zajlik, az előző oldani ábrán a piros útvonalat bejárva. Például a Routing tábla feltöltésekor a Linux, a driver segítségével leküldi az adatot az MHC egységre, ahonnan a gyűrűn keresztül eljut a csomag az SWC Konfigurációs menedzser moduljába, ami a CI interfészen keresztül továbbítja az útvonalválasztási információt az FPGA-n található helyi Routing táblába.

A megvalósításhoz meg kellett tervezni és meg kellett valósítani az ábrán látható modulokat valamint a Linux meghajtót

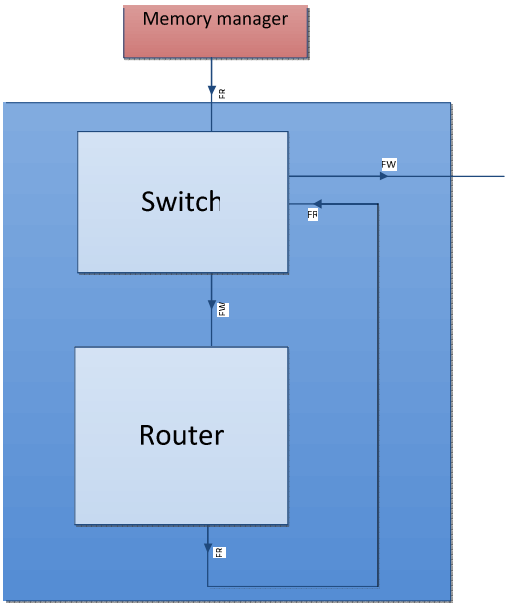
5.4 A firmware környezet elemeinek megvalósítása

5.4.1 Lookup modul bemutatása

Egy kész rendszer módosításakor, továbbfejlesztésekor, először is az adott rendszert kell megismerni, felmérni az állapotát, megvizsgálni mi használható fel belőle később a saját feladatunkhoz. Ezzel nem csak időt takaríthatunk meg magunknak, hanem az egyes megvalósítási lehetőségek számát is csökkenthetjük.

Egy többszintű útvonalválasztó modulnak az alábbiak szerint kell működnie: a beérkező csomagfejlécet először a Switch modul vizsgálja meg. Az Ethernet fejléc tartalma alapján, ha a MAC-cím alapján egy helyi gépnek címezték, akkor a megfelelő kimeneti porton kiküldi, ha külső hálózatba szól, akkor továbbítja a Routernek. A Router modul végre hajtja a megfelelő módosításokat a fejlécen (MAC-cím csere stb.) Routing táblája és az ARP modul segítségével, majd a dedikált portján továbbítja a csomagot, vagy egy másik, hozzá kapcsolódó alhálózat gépének címezve, vagy egy másik, köztes útvonalválasztó célcímével.

A Lookup modul tervezésekor figyelembe vettem a fent említett működést, noha kisebb, nagyobb változtatásokat el kellett végezni, mivel a C-board működése, felépítése eltér a szokványos hálózati eszközöktől. A Lookup vázlatos felépítését az **12. ábra** mutatja be, két nagyobb blokkra osztható, amiket FIFO-k kötnek össze.



12. ábra A Lookup egyszerűsített blokkvázlata

5.4.1.1 Az Input arbiter

Mivel az összes fejléc a Switchen keresztül hagyja el a modult, ez a Lookup kimenete, ezért nem csak a Memória menedzser lesz a fejlécek forrása, hanem a Router modul is ide továbbítja a feldolgozott fejléceket. Az *Input arbiter* feladata, hogy e bemenetek közül mindig egyet kapcsoljon a Switch bemenő sorába. A kapcsolás Round Robin elven történik, vagyis egy számláló értéke reprezentál egy bemenetet (ha a számláló 1, akkor az egyes bemenetet kapcsolja).

5.4.1.2 Switch modul

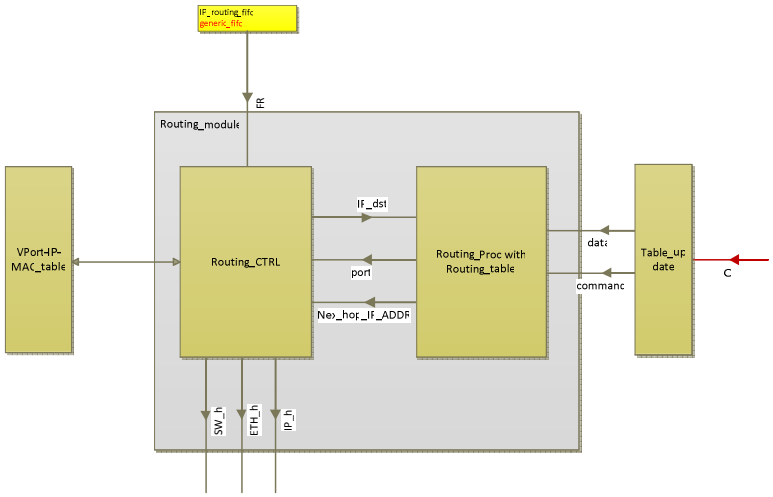
Ezt az egységet a C-boarddal együtt kaptuk, kapcsolási mechanizmusa helyesen működött, viszont fel kellett készíteni, a Router modul miatt szükséges extra információk továbbítására, kezelésére. Az kapcsoló eredetileg két modulból állt (*Controller* és *Processor*), melyeket egy harmadik modullal egészítettem ki. A *Controller* felelős a be- és kimeneti adatok kezelésért, a bejövő fejlécből kinyeri a számára fontos adatokat, a MAC címeket és a bejövő portot, melyeket ezután átad a *Processornak*, ami ezen információk alapján elvégzi a kimenő port keresést. A táblát dinamikusan tölti fel a bejövő fejlécek alapján; minden bejegyzéshez tartozik egy időbélyeg, amely ha a megadott időintervallumból kiesik, akkor a bejegyzés törlődik. Ha a keresés nem jár sikerrel (nincs a keresett MAC-hez port bejegyezve), akkor a csomag minden portra ki lesz küldve, találatkor pedig a megfelelő kimenetre.

Ahhoz, hogy ez az egység fejléceket tudjon küldeni és fogadni a router és PC felől ki kellett egészíteni egy olyan egységgel, a *Switch_Vport_Table*-lel, amely meghatározza, hogy az adott fejléc az útvonalválasztónak szól, ekkor a belső buszon keresztül kell a teljes, beolvasott fejlécet kiküldeni, vagy a kimenet egy fizikai vagy a PC port, ebben az esetben a Lookup kimeneti FIFO-ba kerül a feldolgozott adat. Ezen adatok tárolására egy táblázatot használ, amiben fel vannak sorolva a létrehozott virtuális portok és az a router által lefoglalt fizikai port, amin az alhálózaton kívülre szóló csomagokat küldi. Minden bejegyzéshez, tartozik egy státusz mező, amely megmondja, hogy az adott port kihez tartozik **13. ábra**. A tábla feltöltését, frissítését a táblamenedzser végzi, amely a CI interfészen keresztül kapja az adatokat a PC-től. A módosított kapcsoló előnye, hogy üres virtuálisport tábla esetén az eredeti kapcsoló működése szerint történik a fejlécek feldolgozása.

13. ábra A Switch Vport tábla bejegyzés felépítése

5.4.1.3 Router modul

Ha a csomagot Layer 3 szinten is kell vizsgálni, akkor annak fejlécét a Switch modulból a Routerbe kell továbbítani. A modulban jelenleg egy IP és egy ARP protokoll szerint működő egység található. Az *IP router* feladata a Layer 3 útvonalválasztás megvalósítása. Két nagyobb egységből épül fel, az egyik az útvonalválasztásért felel, a másik az IP-MAC címfeloldást menedzseli. Az útvonalválasztó modul további két nagyobb egységre osztható, ahogy ezt a **14. ábra** is mutatja. A *Controller* feladata a be- és kimenet felügyelete, emellett fejléc ellenőrzést is végez. Először megvizsgálja az IP verzió és TTL értékét, majd a kiadja a cél port, MAC és IP címet a *VPort-IP-MAC_table*-nek keresésre. Abban az esetben, ha nincs találat (nem neki címezték), akkor a fejléc eldobódik.



14. ábra Az IP Router modul felépítése

MAC-cím egyezéskor szétbontja a bejövő fejlécet protokollok szerint (Switch, Ethernet és IP fejlécre), és kiszámítja az új TTL-t. Ezután a cél IP-címet átadja a *Processornak*, ami megkezdi a Routing táblában a keresést, aminek befejeztével visszaad a *Controllernek* egy portot (ez egy virtuális port), amin a csomagot ki kell küldeni, illetve a cél IP-t (helyi hálózatnál a kapott cél IP-t, külső hálózatnál az átjáró címét). A modul végül három fejlécet továbbít: a Switch Headert, az Ethernet keretet, az IP fejlécből pedig csak a cél IP-címet a hálózati maszkkal, az új TTL-t és a port azonosítót.

Az *IP router* másik modulja, felel a címfordítás menedzseléséért, vagyis elküldi a cél IP-t az ARP modulnak és a *VPort-IP-MAC_table*-nek, utóbbinak a hálózati maszkkal együtt. Ha mindkét keresés sikerrel járt, akkor a modul visszakap az *ARP modultól* egy MAC-címet, ami a következő állomást azonosítja, a *VPort-IP-MAC_table*-től pedig az új forrás MAC címet, majd ezután egy fejlécet rak össze a kapott értékekből; a Switch Headert bejövő port mezeje megkapja a Routing táblából kiolvasott értéket, ezután kerülnek a számított cél és forrás MAC-címek, a legvégén pedig az új TTL. Ha bármelyik a keresés nem jelez találatot, a fejléc eldobódik. A Routing tábla feltöltéséért és karbantartásáért, itt is a táblamenedzselő modul felel.

Routing Table			
INET addr (32 bit)	Gateway addr (32 bit)	Netmask (32 bit)	Interface (8 bit)

15. ábra A routing tábla bejegyzés felépítése

Abban az esetben, ha a bejövő fejléc cél IP címe a routeré, akkor vezérlő üzenet jött, amit továbbítani kell a PC-nek, és ilyenkor az útvonalválasztást ki kell hagyni és

továbbítani a címfordító modulnak, ami Switch Header cél port címét a PC port címére módosítja.

Az *ARP modul* a tervek szerint egy teljes egészében hardveresen megoldott, dinamikusan frissülő egység lesz, de jelenleg csak egy csökkentett funkciójú, statikus táblával operáló változatát implementáltam a rendszerben.

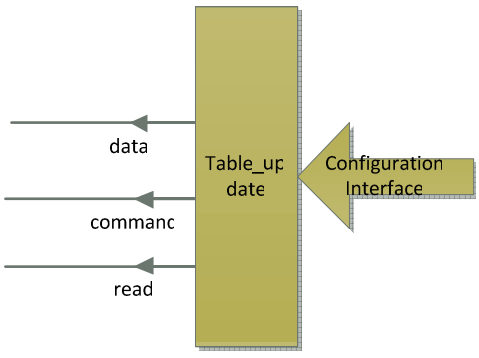
A *VPort-IP-MAC_table* a Router modul egységeit szolgálja ki olyan módon, hogy táblájában nyilvántartja az egyes hálózatokhoz csatlakozó portjait, illetve azok IP- és MAC-címét (**16. ábra)**. A bejegyzések között többféleképpen kereshetünk, az egyes modulok igényei szerint. A tábla frissítéséért a táblamenedzselő a felelős.

Router Vport Table			
IP addr (32 bit)	MAC addr addr (48 bit)	Virtual Port (8 bit)	Status (8 bit)

16. ábra A Router Vport Table bejegyzés felépítése

5.4.1.4 Táblafrissítő modul

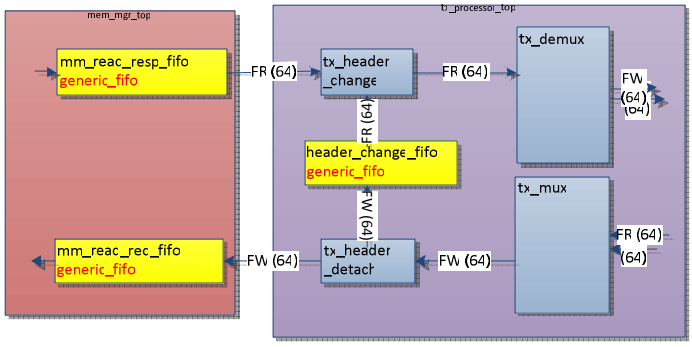
Az egység feladata a konfigurációs interfész illesztése a hozzá kapcsolódó táblakezelőhöz oly módon, hogy az adatok beírása az adott tábla szerint bejegyzésenként történhessen (**17. ábra)**. Ha a konfigurációs menedzser adatot küld, akkor minden egyes példányosított modul megnézi a címbuszt és ellenőrzi a saját címével. Egyezés esetén megkezdődik az adatbusz olvasása, illetve átadja a parancs busz tartalmát és jelzi a táblakezelőnek (ezzel áttételesen a táblából dolgozó egységnek is), hogy frissítés következik. Ha beolvasott egy bejegyzésnyi adatot, akkor azt átadja a táblakezelőnek, ami végrehatja beírást. A modul előnye, hogy könnyen illeszthető tetszőleges szélességű bejegyzéssel operáló táblakezelőhöz.



17. ábra Táblafrissítő modul blokkvázlata

5.4.2 Header módosító egység

A megismert Routing modul beillesztése a már meglevő rendszerben, nemcsak a Lookup modul újragondolását tette szükségessé, hanem meg kellett oldani, hogy az általa végzett fejlécmódosítások a memóriából kiolvasott és kiküldendő teljes csomagra átkerüljenek. A probléma megoldása azért sem volt egyértelmű, mivel meg kellett találni azt a helyet, ahova ez a modul úgy illeszthető be, hogy a környezetét a legkevésbé kelljen megváltoztatni. Nagy kihívás volt még, hogy az egységnek valós időben kell működnie, nem akaszthatja meg a kiküldés menetét. A lehetőségeket mérlegelve a **18. ábra** szerinti elrendezéshez jutottam. A csere a TX modul és a Memória menedzser határán történik két külön modul, a *TX_header_detach*, a *TX_header_change* és egy *Generic FIFO* által. Az adatbusz, mint az a képen is látható mindenhol 64 bites.

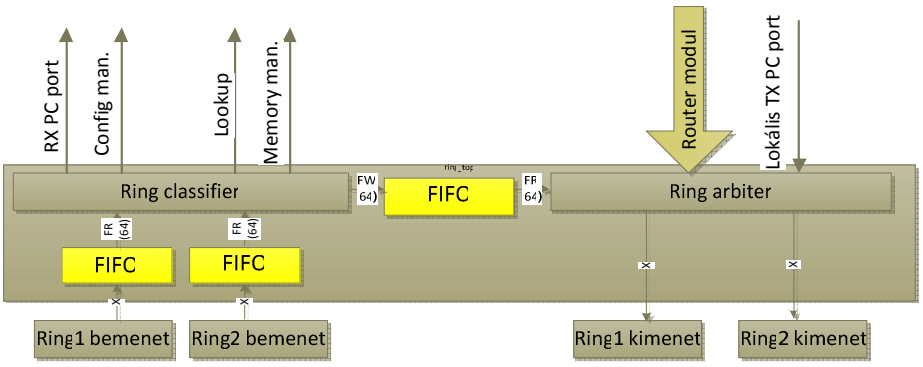


18. ábra A Header módosító egység felépítése és elhelyezkedése

Azt már tudjuk, hogy a Router a fejlécmódosításokat (új MAC-címek, TTL) a Switchen keresztül a Switch Header kiegészítéseként küldi el. Ezeket az információkat, hogy ne vesszenek el még a fejléc Memória menedzser bemeneti FIFO-jába történő írása előtt le kell róla választani, viszont az extra információ nélküli csomagokat a korábbi módon kell továbbítani a bemeneti sorba. A *Detach modul* feladata tehát, a routing csomagok extra adatainak leválasztása és FIFO-ba írása. Minden egyes beírt adat elejére a Package ID-t is beírja, amivel biztosítható, hogy a módosítások a megfelelő csomagoknál történjenek. A *Change modul* feladata ennek az ellenkezője, vagyis a Memória menedzsertől jövő csomag és a FIFO-ban levő adat Package ID-jának egyezésekor írja felül az eredeti paramétereket (a MAC-címeket és a TTL-t), egyéb esetben továbbítja a csomagot változtatás nélkül.

5.4.3 Gyűrű kommunikáció

A CBoard-on belül az FPGA-k közötti adatcsere a gyűrűn történik. Minden FPGA-n található egy *Ring menedzser*, ami gyűrűről érkező és a gyűrűre küldendő adatokat kezeli. A modul megléte alapvető fontosságú ahhoz, hogy az adattovábbító egység kommunikáljon a vezérlő egységgel, jelen esetben a Linuxszal. A csomagtovábbítási keretrendszerben a **19. ábra** szerinti felépítés van meghatározva, aminek egy egyszerűsített változata készült el. Két nagyobb modulból (és néhány FIFO-ból) épül fel, melyek működését az alábbi fejezetekben mutatok be.



19. ábra A Ring menedzser elvi felépítése

5.4.3.1 Ring arbiter

A modul feladata, hogy ütemezze a PC-nek vagy egy másik FPGA-nak szóló üzenetek gyűrűre kerülését. Ahogy az egység nevében is benne van, ezt egy arbiter valósítja meg, ami a számlálója állása szerint olvassa a Router modul vagy a TX PC-nek szóló sorát. Jelenleg csak Konfigurációs és a PC-nek szóló üzenetek sorát olvassa, illetve a Classifier-től érkező, egy másik FPGA-nak szóló üzeneteket képes kiküldeni (a többi bemeneten jelenleg nem érkezik adat). A gyűrűt úgy látja, mint egy FIFO bemenetét, így minden egyes írás előtt ellenőrzi, hogy van-e hely az adott csomagnak (ha nincs, az üzenetet eldobja).

5.4.3.2 Ring classifier

Másik fontos egysége a Ring Classifier, aminek feladata a gyűrűn érkezett csomagok feldolgozása és eldöntése, hogy erre az FPGA-ra szól az üzenet (ha igen, akkor kiküldeni megfelelő modulnak), vagy egy másiknak kell továbbítani az Ring arbiter segítségével. Feladata még a Ring Header eltávolítása az adott FPGA-ra szóló csomagok elejéről. Jelenleg három egységnek tud csomagot továbbítani: ha a fejléc

vizsgálat megállapítja, hogy a csomag nem ide szól, akkor módosítás nélkül beírja az arbiter bemenő sorába. Ha a csomag fejlécében ezt az FPGA-t jelölték meg célállomásnak, akkor eltávolítja az elejéről a Ring Headert, majd megvizsgálja a típus mezőt és ez alapján továbbítja vagy a Konfigurációs modulnak (ha a típus mező 1 értékű), vagy az RX modul PC bemenő sorába (ha a típus mező 1 értékű).

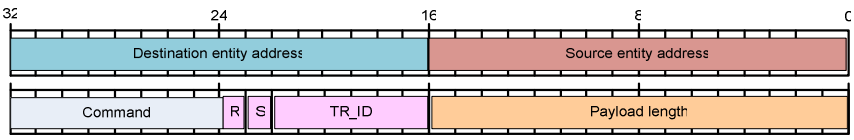
5.4.4 Interfész megvalósítása a vezérlő FPGA és a Linux között

A kártyán működő rendszert szoftveresen, egy számítógép segítségével szeretnénk vezérelni. Ehhez adott a PCI-express interfész, ami kommunikálni tud az FPGA-kal. A kommunikációhoz szükséges feladatvégző elemek sora azonban még hiányos. A PC-kártya interakció meglétéhez szükség van a PCI-express interfész kezelését megvalósító hardver elemekre, amelyek az FPGA-n várják az érkező információt. A leküldött csomagok a gyűrűn eljutnak a cél FPGA-ra. Itt várakozhatnak olyan modulok is melyeket be kell állítani, hogy működni kezdjenek. Ezt a feladatot látja el a konfigurációt vezérlő alrendszer. A PCI-express kommunikáció megvalósításához is szükséges néhány hardveres és szoftveres elem megvalósítása. Ebben a fejezetben ezek megvalósításáról lesz szó.

5.4.4.1 A konfigurációs interfész

A konfiguráció menedzsment feladata a konfiguráció eljuttatása a konfigurálható modulokhoz. A moduloknak támogatnia kell a konfigurációs interfészt. A konfiguráció egy parancsból, továbbá paraméterekből áll. A konfigurációs ciklus mindig egy parancs elküldéséből a konfigurálandó modul felé, majd onnan a válasz visszaolvasásából áll.

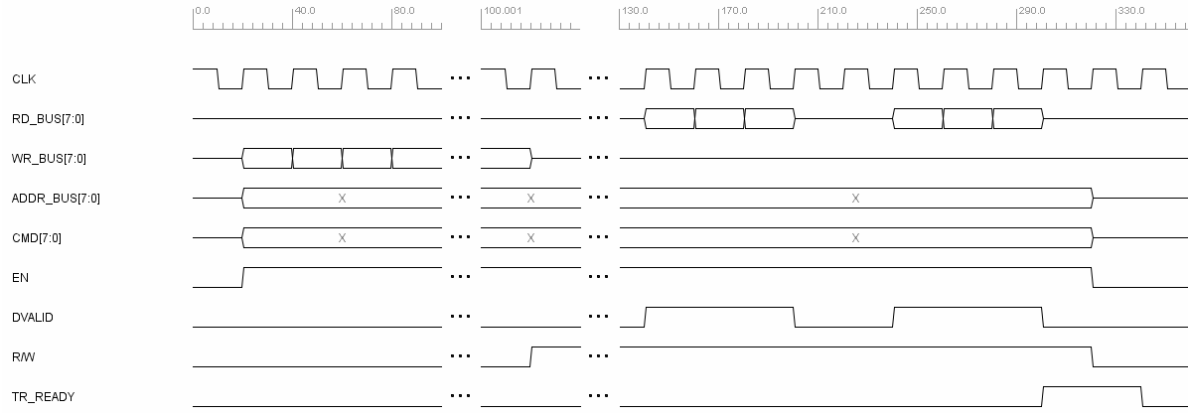
Az alrendszer a gyűrűről érkező konfigurációs csomagok alapján végzi a beállításokat. A gyűrűről érkező adatról egy osztályozó modul a megfelelő helyre továbbítja a csomagokat, és a gyűrű fejléc sorát eltávolítja. Így a konfigurációs interfész a következő fejléccel kapja a csomagot:



20. ábra konfigurációs fejléc

A **20. ábraán** látható, hogy a fejlécben meg kell címezni az adott modult, parancsot kell neki kiadni. Az R/S (Request/Sucess) bitek mondják meg, hogy kérésről van-e szó, illetve az adott kérést lehetett-e teljesíteni.

A megvalósításhoz létre kellett hozni olyan modulokat, melyeket lekérdezni / beállítani lehet. A modulok egy buszra vannak felfűzve, és mindig csak a megszólított elemek hajtják meg a buszt, a többi kimenetének eközben és a megszólítás végén nagyimpedanciás állapotban kell lennie.



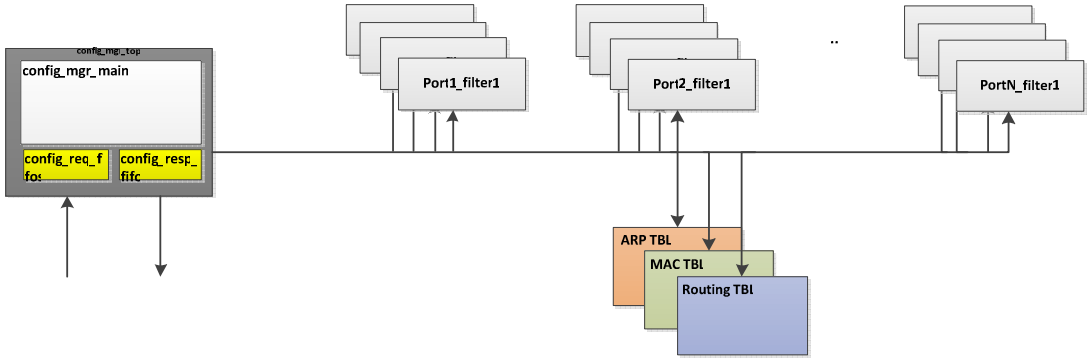
21. ábra konfigurációs interfész busz

Az interfészen a konfigurációs modul, ahogy a **21. ábraán** látható, először engedélyezi a buszt, és közben kiadja a megszólított egység címét, kiadja a parancsot, illetve ír az egység regisztereibe, ha a parancs egyéb paramétereket is megkövetel. Ezután olvasó módba állítja a buszt az R/W jel bebillentésével. Ekkor a megszólított modul válaszol. Minden érvényes válasz bájt kiadásakor bebillen a DVALID jel, majd a válasz végén a TR_READY jel.

A konfigurációs modul egy generic_fifo-ba kapja a parancs csomagokat. A modul egy állapotgép segítségével kezeli ezután a parancsokat. Alap (IDLE) állapotból, ha van csomag (frame_available), CMD_READ állapotba megy, ahol a fejlécből regiszterekbe tölti a parancsokat. Ezután CMD_SEND állapotba kerül, ahol a kimenetre adja a címet és a parancsot, engedélyezi a buszt, és elkezdi bájtonként kiadni az adatot, közben egy számláló számolja a kiadott bájtokat. Az adatok kiküldésének a végén, ha a számláló megegyezik a Payload length (**20. ábra**) értékével, befejeződik az adás. Ekkor RESP_READ állapotba kerül a modul, és egy másik generic fifo-ba bájtonként, minden DVALID-ra beolvassa a választ. A TR_READY jelre újra IDLE állapotba kerül az állapotgép. A slave modulok kimenetei nagyimpedanciás állapotban vannak, ha nincsenek engedélyezve. Ahhoz, hogy a busz jelei ne lebegjenek, készítettem egy default perifériát is, ami “tisztán” tartja a buszjeleket, amikor nincs meghajtás.

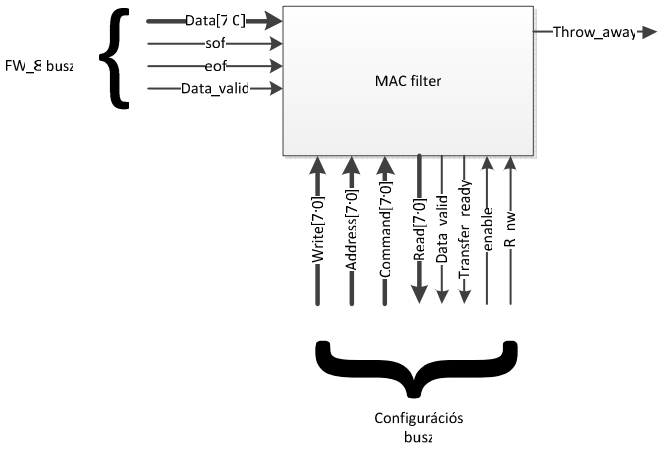
Fontos, hogy megoldjuk az FPGA-kban a konfigurálandó slave modulok felkészítését, hogy azok képesek legyenek a konfigurációs üzenetek fogadására, és a beállítások alkalmazására.

Az alrendszer felépítése a **22. ábraán** látható. Az alrendszer a gyűrűről kapott parancsokat továbbítja a konfigurációs interfészen a többi modulnak.



22. ábra A konfigurációs alrendszer felépítése

A fejlesztéskor egy MAC szűrő modult készítettem el, aminek írható és olvasható paraméterei vannak. A megoldás felhasználható lesz a többi modul megvalósításánál. A megoldás lényege, hogy a MAC szűrő üzemmódjait a konfigurációs interfész parancs busza alapján állítja be.



23. ábra MAC szűrő modul

Az előbb feltüntetett ábrán láthatjuk a konfigurálható MAC címszűrő modult. Az egység a konfigurációs interfészen keresztül felprogramozható, illetve egyes paraméterei lekérdezhetőek. A szűrő négyféle parancs értelmezésére lett felkészítve:

- Modul azonosító lekérdezése (0x00h): a parancs kiadásának hatására a szűrő egy azonosítóval válaszol, Az azonosító a modul funkcióját adja meg.
- MAC szűrési szabály beállítása (0x05h): a parancs hatására a write buszon oktettenként bejövő adatot a modul egy belső MAC szűrési táblában tárolja, illetve a sikeres beállítás esetén a read vonalon jelzi a sikeres beállítást, illetve probléma esetén a hibát

- MAC szűrési szabály törlése (0x06h): a parancs hatására a modul a megadott MAC címet kitörli a belső táblából.
- MAC szűrési szabályok lekérdezése (0x07h): a parancs hatására a modul a tárolt MAC címekkel válaszol, ha nincs tárolt szabály, akkor 0x000000000000h a válasz

Az egység a beállított MAC címeket egy erre a célra létrehozott blokk RAM-ban tárolja. A keretek fogadásakor ebben a táblában keresi a címeket. Találat esetén a throw away kimenetének bebillentésével jelzi, hogy a szóban forgó keretet el kell dobni, mivel ez a cím tiltva van.

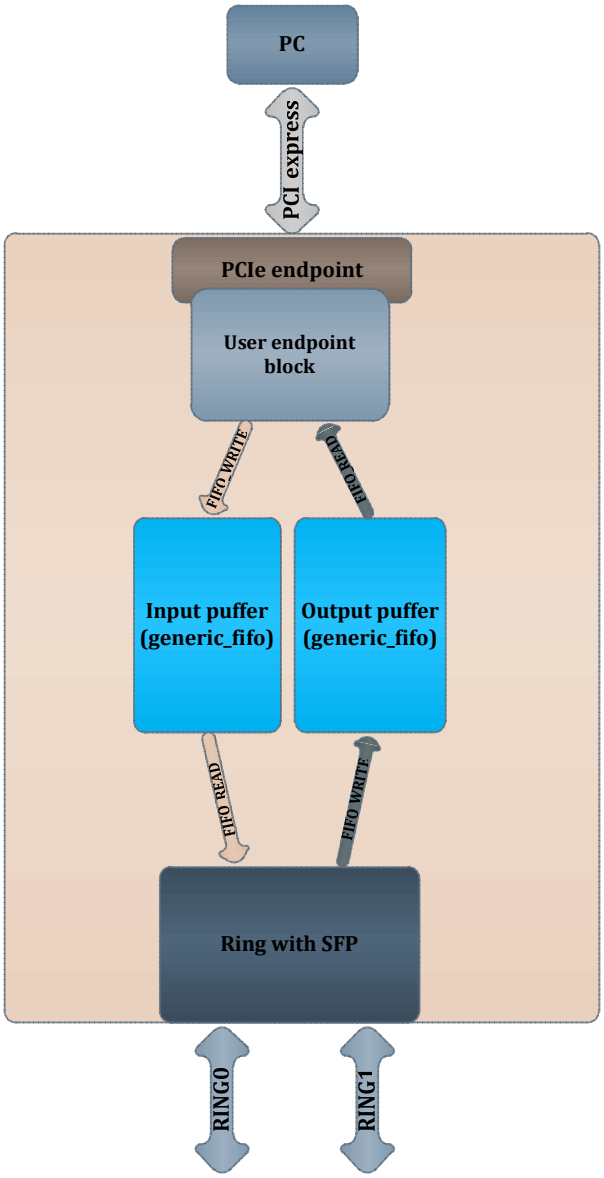
Az eldobást jelző vonal egy speciális FIFO bemenete, amely alap esetben tovább engedi a tartalmat, és az kiolvasható a túloldalról, de ha megkapja az eldobó jelet, akkor nem engedi tovább a csomagot, hanem eldobja azt.

A megoldás lényeges pontja, hogy hasonló feladatok elvégzésére hasonló implementáció szükséges, így a különböző táblák frissítésére illetve lekérdezésére is hasonló megoldást lehet implementálni.

5.4.4.2 PCI-express interfész

Az interfész megvalósítása hardveres és szoftveres részekre oszlik. A bemutatást a hardveres végponttal kezdem.

A megvalósításhoz rendelkezésemre állt egy PCI-express endpoint block, amit a Xilinx Coregenerátorával is el lehet készíteni, ettől megkapjuk a buszon közlekedő TLP csomagokat, illetve azokat tudjuk elküldeni a buszra. Ezen kívül kaptunk ehhez egy User endpoint blockot, ami feldolgozza a PCI-express által küldött TLP csomagokat, és címek alapján a 32 bites bemenő adatot tud kiadni a kimenetén, illetve kiolvasni egy bemenetről. Van egy-egy 16 bit széles írást és olvasást jelző kimenet (WrReq és RdReq). Ezek egy órajelnyi impulzust adnak a címzéstől függően.

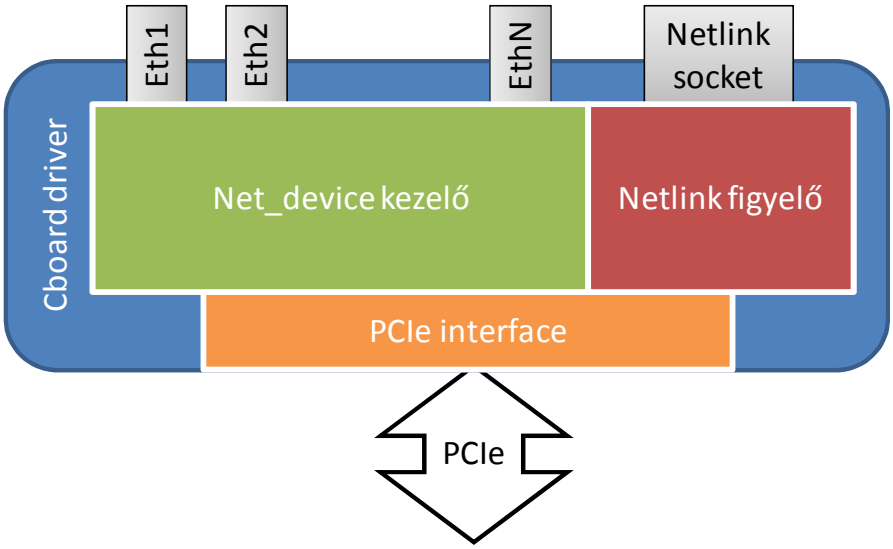


24. ábra MHC FPGA felépítése

Ezt az író olvasó interfészt fel tudtam használni a rendszer interfészének létrehozásához. Létrehoztam az olvasó / író jeleknek megfelelő számú regisztert. Ezeket egy case szerkezetben a write illetve read jelnek megfelelően órajelre be tudom írni, illetve ki tudom olvasni az értékeket a regiszterekből a felhasználói végpont modul segítségével. Így négy cím segítségével a Generic FIFO bemutatása fejezetben szereplő FIFO interfészhez a vezérlőjeleket, illetve egy-egy 32 bites író illetve olvasó interfészt létre tudtam hozni. Tehát ahogy az a Hiba! A hivatkozási forrás nem található.**án** is látható, az író/olvasó processzel kiegészített végpont modul csatlakozni tud ki- és bemeneti FIFO-khoz, amelyekbe le illetve-föl tudja küldeni az adatot. A bemeneti FIFO-ból a gyűrűt kezelő modul kiolvassa a csomagokat, amit tovább is küld a gyűrűre.

A kiolvasás teljesen hasonló módon történik. A gyűrűről megérkezik a csomag, ezt elveszi a gyűrűt kezelő modul, és beírja a FIFO író interfészen a pufferbe. Ezt pedig már mi tudjuk kiolvasni a végpont interfész segítségével.

5.5 Az eszközmeghajtó működése



25. ábra A Linux driver felépítése

A PCI-expressz végpont kezeléséhez rendelkezésünkre állt egy alap (**25. ábra**) és egy úgynevezett satelit driver.

Az alap driver feladata az alapvető PCI-express hívások kezelése, és a csomagok előállítása a busz számára. Erre kapcsolódik rá a satelit driver. Ezt azért nevezik satelit drivernek, mert az FPGA-n beállított magazonosító (core ID) alapján töltődik be. Ezt az alap driver után kell betölteni. A satelit driver arra jó, hogy ha többféle rendszert is fejlesztünk, amelyek azonos csomagokat fogadnak az buszról, de például más regisztereik vannak, vagy más funkciókat (pl. DMA, interrupt) is támogatnak, akkor az azonosítónak megfelelő, hozzá tartozó meghajtó program töltődik be. Ennek a meghajtó programnak a struktúráiba vannak leképezve az általunk használt regiszterek címei. A program moduljának elindítása után lehetséges a C-board eszközt végpontként bejegyezni a rendszerbe.

A meghajtó program segítségével ezután IOCTL hívásokon keresztül tudunk kommunikálni az FPGA-val. A kommunikációhoz írtam függvényeket, melyek képesek az IOCTL híváshoz előállítani a megfelelő struktúrákat, amelyekben átadjuk a mutatót az adatra, amit szeretnénk leküldeni, illetve amit várunk, és a megfelelő regiszter számát, illetve a megfelelő parancsszót, ami egy négybájtos hexadecimális szám. Az

IOCTL hívás után olvasáskor a megfelelő adat megjelenik az átadott pointerben, illetve leküldéskor az adat megjelenik az FPGA regiszterében.

A korábban említett beíró függvények egy címet és egy unsigned long adatmezőt várnak. Ezen alapmetódusok segítségével már létre lehetett hozni olyan függvényeket is, amelyek egész adatsorokat képesek kiolvasni, illetve beírni. Az adatsor beírásához már csak egy karaktermutatót, illetve egy hosszt várok. A mutató tartalmát először átkonvertálom unsigned long mutatóvá, hogy a regiszteríró függvénynek át tudjam adni. Ezután a pointer elemein végiglépkedve beírogatom őket a regiszterbe. Természetesen az első és az utolsó adatrész leküldésekor kezelem a FIFO keretkezdet és -vég jeleit.

Létrehoztam a megfelelő struktúrákat a rendszerben használt fejléceknek megfelelően. A létrehozott struktúrák a **1. Táblázatban** láthatóak. Észrevehetjük, hogy a struktúrák mezői rendre megegyeznek a normál-switch és konfigurációs fejléccel. A konfigurációs gyűrű fejléc értelemszerűen a normál gyűrűs esettől abban tér el, hogy az egy konfigurációs fejlécet tartalmaz.

Normál fejléc	**Konfigurációs fejléc**

`typedef struct _fpga_hdr`
`{`
`  int port_i;`
`  int port_o;`
`  int length;`
`  ULONG rest;`
`  ULONG timestamp;`
 	
`}GPFF_FPGA_HDR;`
	`typedef struct _conf_hdr`
	`{`
	`  int dst_entity;`
	`  int src_entity;`
	`  int command;`
	`  int rs_tr_id;`
	`  int pload_length;`
	`}GPFF_CONFIG_HDR;`

Normál gyűrű fejléc	**Konfigurációs gyűrű fejléc**

`typedef struct _ring_hdr_norm`
`{`
`  int fpga_dst;`
`  int ring_type;`
`  int fpga_src;`
`  int rsvd;`
`  int length;`
`  GPFF_FPGA_HDR fpga_hdr;`
`}GPFF_RING_HDR_NORMAL;`
	`typedef struct _ring_hdr_conf`
	`{`
	`  int fpga_dst;`
	`  int ring_type;`
	`  int fpga_src;`
	`  int rsvd;`
	`  int length;`
	`  GPFF_CONFIG_HDR conf_hdr;`
	`}GPFF_RING_HDR_CONFIG;`

1. Táblázat A rendszer által használt fejléc formátumoknak megfelelő struktúrák

A struktúrák létrehozása után megírtam azokat a függvényeket, amelyek képesek az egyes fejlécekkel együtt az adatot leküldeni. Ehhez segédfüggvényeket kellett létrehoznom, amelyek átkonvertálják a fejléceket a megfelelő hexadecimális karakterekké, hiszen a leküldő függvény ilyen mutatót vár. Vagyis készítettem egy konfigurációs gyűrű fejléc struktúrát és egy normál gyűrű struktúrát karakterlánccá

konvertáló függvényt. A függvények prototípusát a 49. oldalon a [10][B] függelékben követhetjük.

Ezeket a függvényeket hívják meg a konfigurációs üzenetet leküldő és a normál üzenetet leküldő függvények. Ezek már csak egy kitöltött struktúrát várnak, illetve egy adatot tartalmazó karaktermutatót, ami az üzenet hexadecimális leképzését reprezentáló sztring elejére mutat. Az eljárások megnyitják a cboard bejegyzett eszközfájlját a /dev könyvtárban, és létrehozzák ennek a HANDLE számát – az eszközt ezzel tudja azonosítani a program. Meghívják a nekik megfelelő konvertáló eljárásokat, majd a kapott karakterláncot az adat elejére másolják, ezután a hossz mezejében lévő számot – ami a teljes adatmező hosszával egyenlő – illetve a HANDLE számot át tudják adni az adatleküldő függvénynek, ami csak megfelelő (unsigned long tömb) formába hozza az adatot, majd a tömbön végiglépkedve leküldi azt a PCI-express interfész végén várakozó FIFO-ba. Továbbá létrehoztam egy olvasó függvényt is, ami normál csomagokat olvas ki, és a kiolvasott adatot karakterekké, illetve normál fejléccé konvertálja. Ez egy packet struktúra mutatót vár, ami tartalmazza az adatot és a hozzá tartozó normál fejlécet. A függvény lefut, és a futás után a tömbben benne lesz a várt eredmény.

5.6 Virtuális interfészek létrehozása a Linuxban

A hardveren a táblák frissítését kapcsolatba kell hoznunk a Linuxszal, vagyis a board interfészeinek meg kell jelennie a Linux interfészei között, különben nem tudjuk milyen változások alapján generáljunk parancsokat és táblafrissítést a hardver számára. Így ha vannak beregisztrált interfészek az operációs rendszerben, akkor akár manuálisan történik egy táblaváltoztatás, akár valamilyen útvonalválasztó protokoll működése során változik meg a Linux útvonalválasztó táblája, azt a NETLINK interfész segítségével el tudjuk kapni, illetve a bejegyzéseket le tudjuk kérdezni. A lekérdezett bejegyzésekből ezután legeneráljuk a saját táblázatunkat, ezt ellátjuk a táblafrissítéshez szükséges konfigurációs FPGA fejléccel, és az előbbi fejezetben leírt módon elküldjük a hardvernek, aki értelmezi, és alkalmazza a változtatásokat.

5.6.1 Interfészek beregisztrálása a Linuxban

Az interfészek létrehozásához és beregisztrálásához egy kernel modult kell létrehozni.

A kernel modul a felcsatoláskor a cboard_probe(void) függvényt futtatja le. Ez a függvény hozza létre a net_device struktúratömböt, amit aztán kitölt netdevice elemekkel. Minden egyes elem egy-egy interfész struktúrája, a függvény a felhozáskor inicializálja őket, a privát struktúrájuknak helyet foglal a memóriában az alloc_etherdev(sizeof (struct cboard_eth_priv)) függvény segítségével. a felhozáshoz meghívódik az ether_setup(netdev) függvény. Kitölti továbbá az interfész struktúra eszközcím (netdev → dev_addr) mezőjét, ami a port fizikai MAC címét tartalmazza. Hiba esetén a free(netdev) segítségével a lefoglalt területet fel tudja szabadítani.

A modul létrehozza az interfészeknek megfelelő számú netdevice struktúra tömböt. Később ezeken keresztül tudjuk elérni az egyes interfészek paramétereit. A cboard_open(struct net_device *ndev) függvényt használjuk az eszköz inicializálására, az interfész felkapcsolásakor (ifup) ez fut le. Itt lehet végrehajtani olyan beállításokat, amelyek az indításkor meg kell, hogy történjenek. Ilyenek például a különböző működési módok engedélyezése, de amikor az rc.inet beállítja az eszköz IP címét, akkor is ez a függvény fut le. Illetve a netif_start_queue(netdev) függvény létrehozza az interfészekhez tartozó puffereket.

A cboard_close(struct net_device *ndev) függvény az előző inverze. Ez fut le, ha lekapcsoljuk az interfészt (ifdown). Itt állnak le az interfészek pufferei.

A cboard_start_xmit(struct sk_buff *skb, struct net_device *ndev) függvény a felelős a küldésért. Ez egy pointerben megkapja a pufferelt adatokat, amiket el szeretnénk küldeni, illetve az interfész pointerét. Az sk_buff struktúra data mezője tartalmazza a kimenő adat pointerét, illetve a len mező az adat hosszát. Így egy egyszerű pointer átadással itt megtörténhet a küldés. Ez általában DMA-val történik, ilyenkor nem veszítünk processzoridőt. Így az interfészek megszólítását is leállítjuk a netif_stop_queue(netdev) függvénnyel, majd a start párjával újraindítjuk a puffereket a küldés befejeztével. A folyamat végén a dev_kfree_skb(skb) függvénnyel tudjuk a puffert felszabadítani.

A kernel modulok írásához létre kell hozni néhány speciális függvényt. Ilyen az __init cboard_init(void) függvény, ami a cboard_probe() függvényt hívja. Ez hívódik meg, amikor elindítjuk a kernelmodult. A modul kivételekor pedig az __exit cboard_exit(void) függvény hívódik meg, ami sorban meghívja minden létrehozott interfészre az unregister_netdev(netdev[i]) függvényt és a korábban említett free_netdev(netdev[i]).

Legvégül egy net_device_ops struktúrában meg kell adni a függvényeket, amik lefutnak, amikor az interfészt felkapcsoljuk (.ndo_open), lekapcsoljuk (ndo_close), vagy küldünk az interfészen (ndo_xmit).

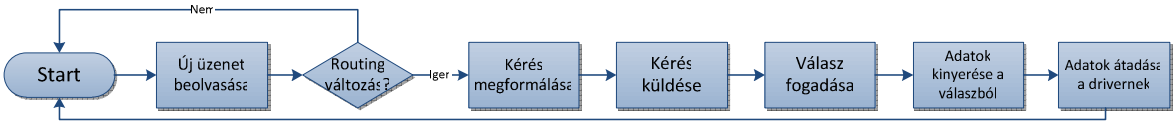
A modul fordítása után az insmod cboard.ko nr_interfaces=8 paranccsal behelyezhetjük a modult, és létrejön 8 darab virtuális interfész.

5.6.2 Interfész modulok és a kártya interfészei közötti kapcsolat

Az interfészmodulok és a kártya interfészei között a kapcsolatot a már sokszor említett fejlécek adják. Vagyis ha tudjuk, hogy melyik interfészmodul küld, akkor a neki megfelelő módon a már ismertetett függvényeket meghívva lehet leküldeni a csomagokat. Vagyis az interfészmodulok és a kártyainterfészek összerendelése teljesen dinamikus. Létrehozatjuk az összerendelést például az interfészeket beregisztráló kernelmodulban is például a privát struktúrában elhelyezve az interfészmodulnak megfelelő kimeneti port számot, amit a leküldéskor az xmit függvényben a már korábban említett normál gyűrű fejlécbe beírva, és a fejléc további részét kitöltve a csomag a megfelelő fizikai porton távozik.

5.7 Az útvonalválasztási tábla frissítése a Linux segítségével

Az egység feladata, hogy a Linuxon futó routing tábla változásának esetén kérje le a teljes tartalmat, majd a Cboard driver segítségével küldje le az adattovábbító részben működő router táblájába. A változások nyomon követéséhez és a tábla lekéréshez a NetLink protokollt használtam, ami egy socket alapú interfészen keresztül biztosítja a kommunikációt a kernel és a felhasználói szinten futó programom között, kérés és válasz üzenet formájában. Továbbá egy multicast socket interfészt is biztosít, amire feliratkozva értesítéseket kaphatunk a routing megváltozásáról. A programom ötvözi ezt a két funkciót, mivel az FPGA-ra mindig a teljes routing táblát kell továbbítani. Éppen ezért a program szerkezete olyan, hogy a multicast kérésbe vannak beágyazva a teljes táblalekérdezés függvényei. Induláskor a main függvény a socket létrehozása után, amiben feliratkozik a routing üzenetek multicast-jára is, meghívja a trigger függvényt (lásd **26. ábra**), ami folyamatosan beolvassa a bejövő üzenetek egy karaktertömbbe, majd megvizsgálja a multicast csoportot.



26. ábra A trigger függvény folyamatábrája

Ha routing táblával kapcsolatos változás jött (ez a Netlink fejléc group mezejében van), akkor a beolvasott adat eldobódik és mesghívódnak a lehívó függvények. Az első függvény kitölti a kérés paramétereit (Netlink header és a routing üzenet paraméterek), majd a küldő függvény egy csomagba rendezi őket és elküldi a socket-en a kernelnek. Ezután hívódik meg az válaszüzenet fogadó függvény, ami ugyanúgy a bejövő csomagokat írja egy globálisan elérhető karaktertömbbe, mint a trigger függvény. Ha végzett, akkor jön a program fő része, az adatok kinyerése az üzenetekből. A függvény elején megtörténik a leküldéshez szükséges fejlécek kitöltése, majd sorban megvizsgál minden csomagot a karaktertömbben, amit a fogadó függvény beleírt. Ha routing táblával kapcsolatos üzenetet talál, akkor abból egy másik ciklus segítségével kiolvassa a hálózati címet vagy a gateway címet (mindig csak az egyik van benne), az interfészazonosítót és a netmask 1-es bitjeinek számát. Majd attól függően, hogy hálózati vagy gateway címet olvastunk be, feltöltjük 0-val a másik címét. Ezután a kiolvasott netmask 1-es bitek száma alapján kiolvassa a helyes hálózati maszkot egy általam létrehozott konstans táblából. Ha mindez megvan, akkor beírjuk a kigyűjtött adatokat egy karaktertömbbe és megnöveljük a Ring és Configuration fejlécekben a hosszváltozót egy bejegyzésnyivel. Ha minden csomagot kiolvastunk a tömbből, akkor visszatérünk a trigger függvénybe és átadjuk a tömböt a driver leküldő függvényének.

6 Tesztelés és értékelés

A teszteléskor próbáltunk arra törekedni, hogy szemléletes módon mutassuk be az eszköz általunk elkészített funkcionalitásait. A fejezetben bemutatjuk, hogyan működik a kiegészített keretrendszer, illetve a Linux és a keretrendszer közötti kommunikáció.

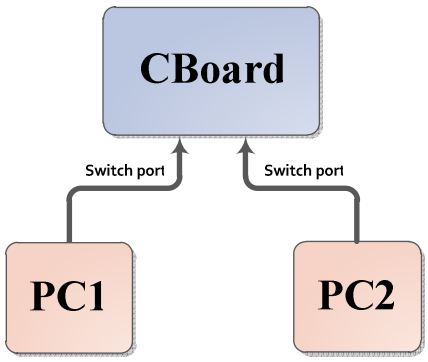
6.1 Funkcionális tesztek

Ebben a fejezetben tárgyaljuk a keretrendszeren végzett méréseket. Itt két működési módot tudunk megkülönböztetni. A kapcsoló módot, amikor a rendszer egyszerű Ethernet kapcsolóként működik, illetve az útvonalválasztó módot, amikor a rendszer útvonalválasztóként működik egy átjáró (gateway) és a többi kapcsoló port között.

6.1.1 Kapcsoló mód

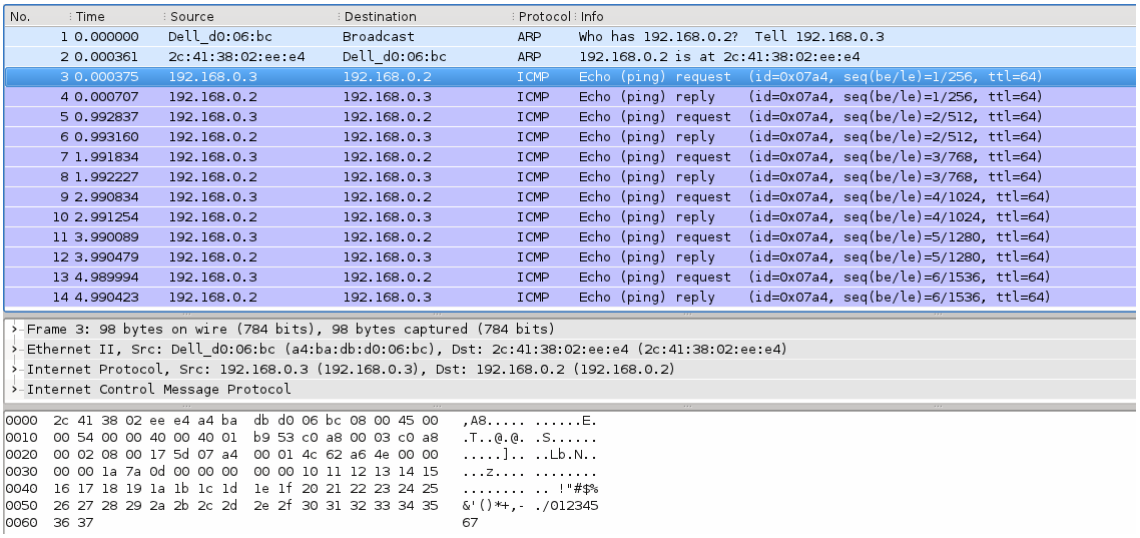
A rendszer feltöltött táblák nélkül egyszerű kapcsolóként működik, egyszerű MAC címek szerinti továbbítás történik. Tehát a rendszer jól illeszkedik az eredeti felépítéshez, vagyis a megfelelő portokon a megvalósított modulokkal kiegészítve az eredeti rendszerfunkciók nem tűnnek el, ezt mutatjuk meg ebben a tesztben.

Ahogy a mérési elrendezésben is látható a **27. ábraán**, a Cboard két tetszőlegesen választott portjára egy-egy számítógépet kötöttünk (jelen esetben minden portot a switch kezel), majd az IP címek beállítása után - ahol figyelni kellett, hogy a két gép egy tartományon legyen - ping üzenetekkel ellenőriztük, hogy az adattovábbítás mindkét irányba működik-e. Mérés közben mindkét számítógépen figyeltük a csomagforgalmat a WireShark forgalomfigyelővel.



27. ábra Switch modul ellenőrzéséhez használt mérési elrendezés

Ahogy az alábbi képen is látható, a küldő gép ARP kérése után, amivel mellesleg a broadcast funkciót is ellenőriztük, minden egyes ping kérésre a célállomás egy válaszcsomagot küld, vagyis a módosított kapcsoló feltöltetlen tábla esetén az eredeti működést valósítja meg.

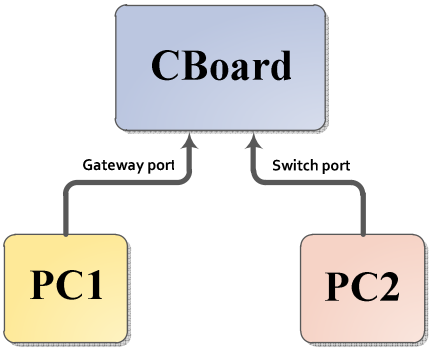


28. ábra Azonos alhálózaton levő gépek közötti adatátvitel

6.1.2 Útvonalválasztó mód

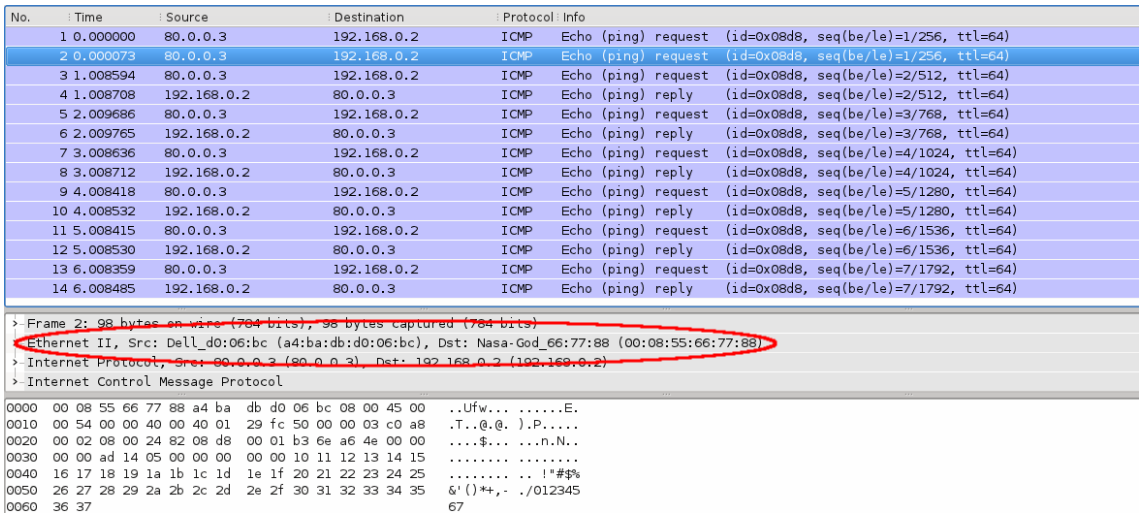
Ebben a tesztben megmutatjuk, hogy a táblák feltöltése után már az útvonalválasztás is belép a működési láncba. Vagyis megszűnik a kapcsoló portok és az átjáróport közötti közvetlen, MAC szintű kapcsolat, eltűnnek a hálózati réteg szintű szórások és életbe lép a címfordítás az átjáró felé.

A méréselrendezés hasonló az előbbihez, de itt az egyik számítógép által használt portot a Cboard-on már gateway portnak állítottuk be a konfigurációs interfészen (a képen a PC1). A PC-k beállításakor arra kellett ügyelni, hogy a két gép különböző IP tartományba kerüljön. A tesztben ezért a PC1-en a 80.0.0.3-as címet, míg a PC2-n a 192.168.0.2-es címet állítottuk be. Emellett meg kellett még adni mindegyik számítógépen az alapértelmezett átjáró IP címét (a Cboard routerének címeit) a routing táblájukban, ez mindkét gép esetén a saját tartományuk .1-es címe.



29. ábra A router ellenőrzéséhez használt mérési elrendezés

Ennél a mérésnél az útvonalválasztó címfordítását akartuk ellenőrizni, hogy a routing táblában való keresés és a cél IP címfeloldás helyesen hajtódik-e végre. A rendszer sajátos felépítése miatt (a csomag először a Switchbe érkezik, aztán kapja meg a Router) várható volt, hogy az elején előfordulnak „hibásan” kiküldött csomagok. Ennek az az okozója, hogy egyrészt a csomag útvonalválasztásához, címfordításához idő kell (táblákban kell keresni), másrészt a kapcsoló táblája olyan, hogy dinamikusan töltődik fel a bejövő csomagok forráscímeiből. Így amíg a router nem küld ki csomagot, addig a címe nem kerül be ebbe a táblába, így minden neki szóló csomag ki lesz küldve az összes porton. Ezt felfoghatjuk egy kezdeti tranziensként is. A csomagvesztést ilyen esetben azzal védtük ki, hogy router és a kapcsoló közé egy jól méretezett FIFO-t építettünk be. Az alábbi két képen megfigyelhető ez a kezdeti tranziens, majd ezután a címfordítás működése. Mivel kevés bejegyzéssel dolgoztunk, ezért a tranziens is kicsi, ahogy látható, már a második csomag után megtörténik a MAC-cím fordítás, ahogyan ezt a **30.** és **31. ábrán** is láthatjuk, ha összehasonlítjuk a bekarikázott részeket. A mérési eredmények alapján megállapíthatjuk, hogy a router képes két különböző tartományhoz kapcsolódó gép közötti adatátvitelre.

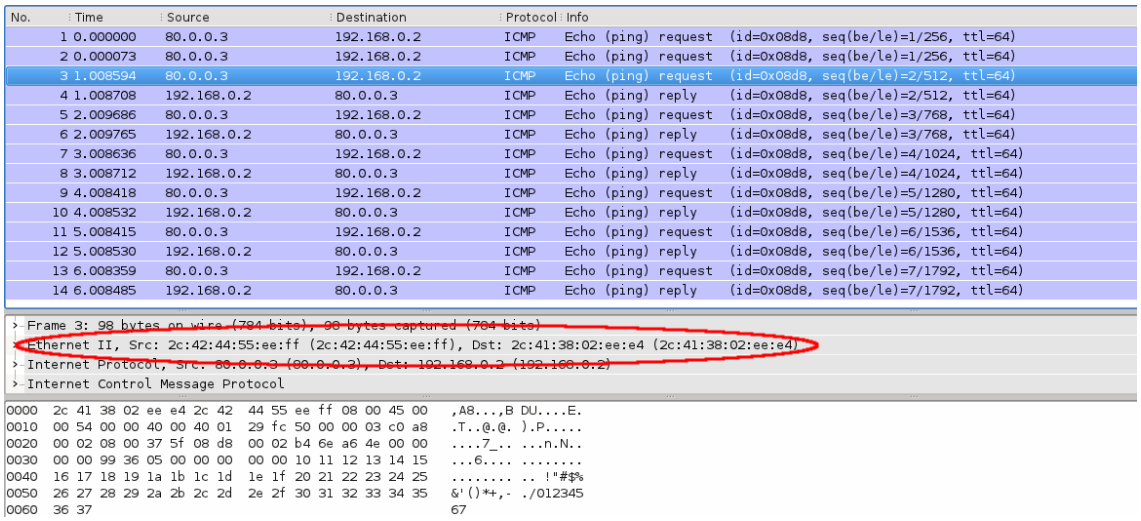


No.	Time	Source	Destination	Protocol	Info
1	0.000000	80.0.0.3	192.168.0.2	ICMP	Echo (ping) request (id=0x08d8, seq(be/le)=1/256, ttl=64)
2	0.000073	80.0.0.3	192.168.0.2	ICMP	Echo (ping) request (id=0x08d8, seq(be/le)=1/256, ttl=64)
3	1.008594	80.0.0.3	192.168.0.2	ICMP	Echo (ping) request (id=0x08d8, seq(be/le)=2/512, ttl=64)
4	1.008708	192.168.0.2	80.0.0.3	ICMP	Echo (ping) reply (id=0x08d8, seq(be/le)=2/512, ttl=64)
5	2.009686	80.0.0.3	192.168.0.2	ICMP	Echo (ping) request (id=0x08d8, seq(be/le)=3/768, ttl=64)
6	2.009765	192.168.0.2	80.0.0.3	ICMP	Echo (ping) reply (id=0x08d8, seq(be/le)=3/768, ttl=64)
7	3.008636	80.0.0.3	192.168.0.2	ICMP	Echo (ping) request (id=0x08d8, seq(be/le)=4/1024, ttl=64)
8	3.008712	192.168.0.2	80.0.0.3	ICMP	Echo (ping) reply (id=0x08d8, seq(be/le)=4/1024, ttl=64)
9	4.008418	80.0.0.3	192.168.0.2	ICMP	Echo (ping) request (id=0x08d8, seq(be/le)=5/1280, ttl=64)
10	4.008532	192.168.0.2	80.0.0.3	ICMP	Echo (ping) reply (id=0x08d8, seq(be/le)=5/1280, ttl=64)
11	5.008415	80.0.0.3	192.168.0.2	ICMP	Echo (ping) request (id=0x08d8, seq(be/le)=6/1536, ttl=64)
12	5.008530	192.168.0.2	80.0.0.3	ICMP	Echo (ping) reply (id=0x08d8, seq(be/le)=6/1536, ttl=64)
13	6.008359	80.0.0.3	192.168.0.2	ICMP	Echo (ping) request (id=0x08d8, seq(be/le)=7/1792, ttl=64)
14	6.008485	192.168.0.2	80.0.0.3	ICMP	Echo (ping) reply (id=0x08d8, seq(be/le)=7/1792, ttl=64)

```
> Frame 2: 98 bytes on wire (784 bits), 98 bytes captured (784 bits)
> Ethernet II, Src: Dell_d0:06:bc (a4:ba:db:d0:06:bc), Dst: Nasa-God_66:77:88 (00:08:55:66:77:88)
> Internet Protocol, Src: 80.0.0.3 (80.0.0.3), Dst: 192.168.0.2 (192.168.0.2)
> Internet Control Message Protocol
```

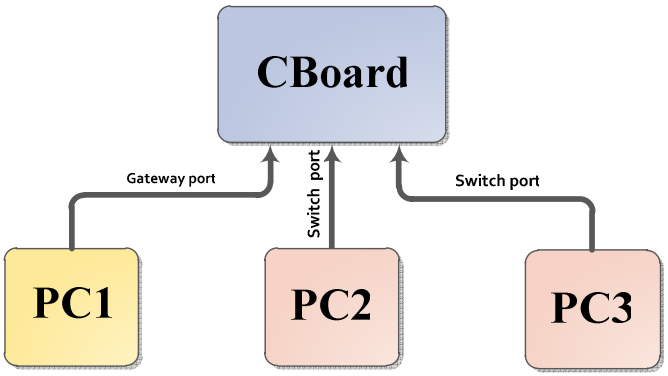
```
0000  00 08 55 66 77 88 a4 ba  db d0 06 bc 08 00 45 00   ..Ufw... ......E.
0010  00 54 00 00 40 00 40 01  29 fc 50 00 00 03 c0 a8   .T..@.@. ).P.....
0020  00 02 08 00 24 82 08 d8  00 01 b3 6e a6 4e 00 00   ....$... ...n.N..
0030  00 00 ad 14 05 00 00 00  00 00 10 11 12 13 14 15   ........ ........
0040  16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25   ........ .. !"#$%
0050  26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35   &'()*+,- ./012345
0060  36 37                                             67
```

30. ábra Csomag tartalma a címfordítás előtt



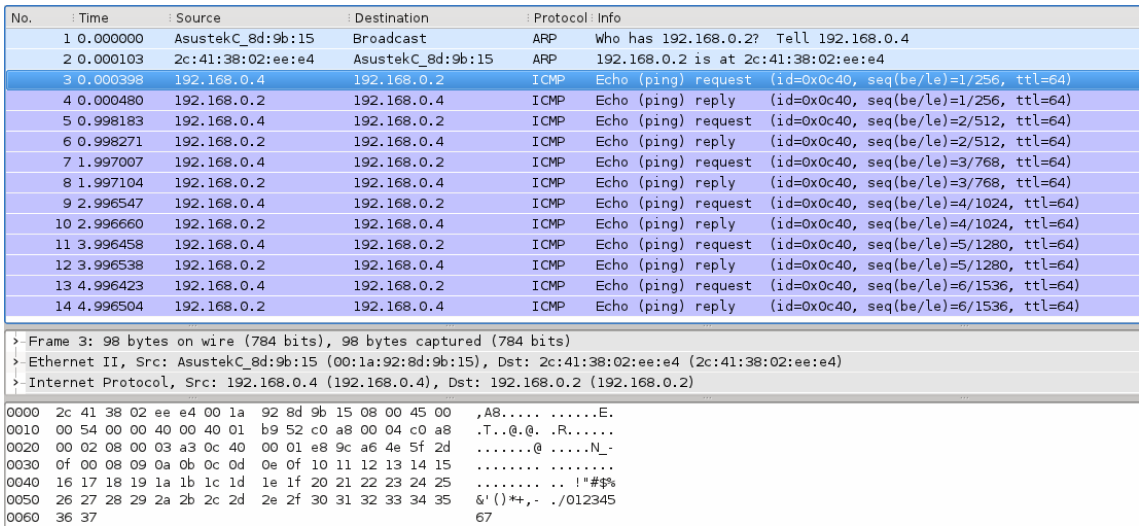
31. ábra Csomag tartalma címfordítással

A router működése mellett érdekelt még az is, hogy a módosított kapcsoló ellátja-e feladatát, vagyis két azonos tartományon levő gép között képes-e a kapcsolásra, és az üzenetszórásból kihagyja-e a routernek lefoglal fizikai portot. A mérési elrendezés annyiban változott az előzőhöz képest (lásd **32. ábra**), hogy bekötöttünk még egy számítógépet a Cboard Switch Portjára, majd teszteltük ezt az összeállítást.



32. ábra Switch modul ellenőrzése Router modul használatakor

Mérés előtt mindhárom gépen indítottunk egy forgalomfigyelőt, majd a PC2-ről ping üzenetet küldtünk a PC3-ra. Ahogy az a **33. ábraán** is látható, az ARP üzenet megválaszolása után a kérésekre egyből érkezik válasz, nem történik címfordítás. Mivel a PC1 másik IP tartományon van, ezért ebből az adatcseréből nem érzékel semmit (még az ARP üzenetet sem kapja meg)



33. ábra A kapcsoló oldal ellenőrzése

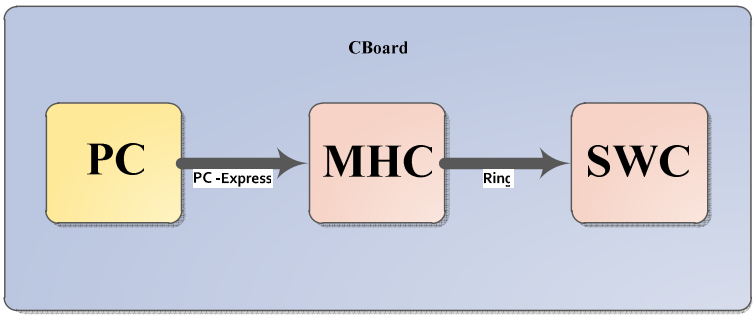
6.2 Együttműködési tesztek

A fejezetben láthatjuk, hogy a PC képes kommunikálni a keretrendszerrel, sőt akár csomagot is tudunk a C-boardon keresztül a hálózatra küldeni.

6.2.1 Konfigurációs üzenet küldése az FPGA-nak

Ebben a tesztben megmutatjuk, hogy a Linux képes a driveren keresztül kommunikálni a keretrendszerrel. Tudunk a számítógépről táblaadatokat küldeni, ami megjelenik a konfigurációs interfészen. Ugyanakkor a Linux képes maga felismerni a beregisztrált Ethernet portokat, és ha táblaváltoztatást végzünk valamelyik interfészen, akkor annak adatai megjelennek a NETLINK interfészen. Az elkapott Netlink üzenetekből a program legenerálja a szükséges fejléceket a tábláknak, illetve hozzáfűzi a táblaadatokat, és elküldi a hardvernek. A változási adatok megjelennek a hardver konfigurációs interfészén, és a táblák ez alapján frissülnek.

A mérési elrendezés ebben az esetben adott volt, vagyis az adat a Linux-ot futtató PC-ről a PCI-Express csatolón keresztül eljut az MHC-ra, onnan pedig a gyűrű keresztül az SWC FPGA-ra, azon belül pedig a konfigurációs menedzser segítségével a megfelelő táblába.



34. ábra Az adat továbbítás útvonala táblafrissítéskor

35. ábraán már az interfészek beregisztrálása után vagyunk, és módosítjuk a routing táblát oly módon, hogy hozzáadunk, majd elveszünk egy bejegyzést belőle, mivel a programnak mindkettőre reagálnia kell.

```
root@cboard-desktop:/home/cboard/Imre/Linx_cBoard_drv/cboard_api# route add 10.0.0.0 eth1
root@cboard-desktop:/home/cboard/Imre/Linx_cBoard_drv/cboard_api# netstat -r
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
10.0.0.0        *               255.255.255.255 UH        0 0          0 eth1
152.66.247.0    *               255.255.255.0   U         0 0          0 eth0
link-local      *               255.255.0.0     U         0 0          0 eth0
default         router.tmit.bme 0.0.0.0         UG        0 0          0 eth0
root@cboard-desktop:/home/cboard/Imre/Linx_cBoard_drv/cboard_api# route del 10.0.0.0 eth1
root@cboard-desktop:/home/cboard/Imre/Linx_cBoard_drv/cboard_api#
```

35. ábra Táblamódosítás a Linuxon

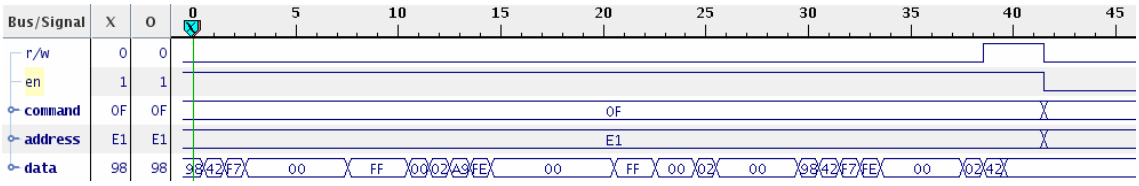
Működés ellenőrzése miatt figyeltük az adatáramlást a Linux-on futó programban, hogy elkapja-e a routing tábla változásokat és ennek hatására működik-e a tábla letöltés és helyesen állítja-e össze a küldendő csomagot. 36. **ábra** a routing tábla változás hatására bekövetkező lekérdezést (a képen az első három sor) és csomagba helyezését mutatja (4 sor), aminek elején a Ring Header van, utána pedig a kitöltött Config Header a legvégén pedig a hasznos adat, ami jelen esetben a letöltött routing tábla.

```
<terminated> update_tables [C/C++ Application] /home/cboard/workspace/update_tables/Debug/update_tables (2011.10.25. 11:25)
dst 9842f700 mask ffffff00 gw 0 if 02
dst a9fe0000 mask ffff0000 gw 0 if 02
dst 0 mask 00000000 gw 9842f7fe if 02
113033 e11f027 9842f70000000000ffffff0002a9fe000000000000ffff000002000000009842f7fe0000000002
```

36. ábra A program kimenete a változásokról

Mivel az adat sok modulon és architektúrán megy keresztül, mire beíródna megfelelő táblába, ezért figyeltük a konfigurációs interfészt és a rajta áramló adatokat is, amihez az egyes táblafrissítők kapcsolódnak. A **37. ábraán** látható, ahogy a PC-ről kiküldve és egy köztes FPGA-n továbbítódva csomagunk megérkezik erre az interfészre, ahonnan a táblafrissítő beírja az útvonalválasztó által használt routing táblába. Érdemes megfigyelni, hogy a **36. ábra** utolsó sora (ezt állította össze a Linux-

on futó program) megegyezik a **37. ábra** utolsó sorával (a táblába beírásra kerülő adattal a fejléc nélkül).

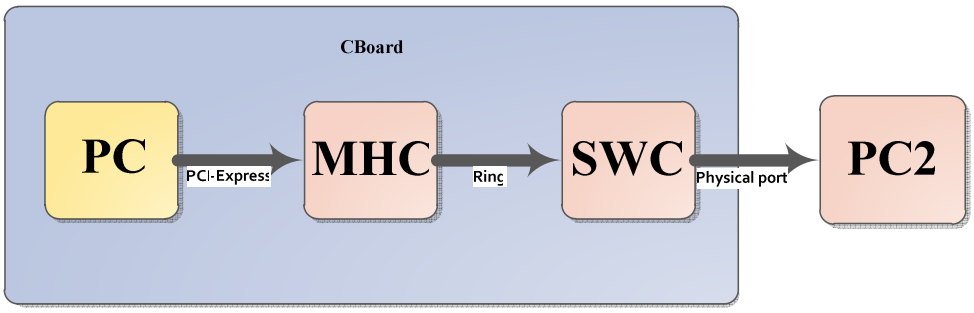


37. ábra A konfigurációs vezérlő modul által küldött adatok

Ezzel a teszttel közvetve azt is bizonyítottuk, hogy a rendszerünk a szabványos NETLINK interfészen keresztül képes együttműködni bármilyen útvonalválasztó protokollal. Hiszen azok is olyan táblaváltozásokat idéznek elő, mint ahogy azt mi tettük manuálisan a *route del* paranccsal. Tehát az ebből adódó változásokat is képesek vagyunk elfogni, és ezután a hardveres táblákat frissíteni

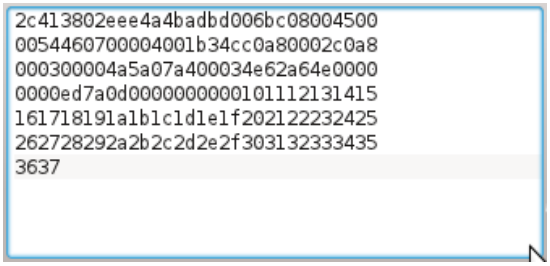
6.2.2 Csomagküldés Linux-ból egy külső célállomásra

A rendszer működésében nagyon fontos szempont volt még, hogy a vezérlési rész (vagyis a Linux), ha információra van szüksége, vagy egy információt akar megosztani, akkor tudjon csomagot küldeni a hálózat többi tagjának. Ebben a fejezetben megmutatjuk, hogy rendszerünk erre is képes. A mérési elrendezést, az alábbi ábrának megfelelően, egy a C-board egyik portjára kapcsolódó számítógéppel is kiegészítettük, amire a Linux-ból a csomagot küldeni fogjuk.



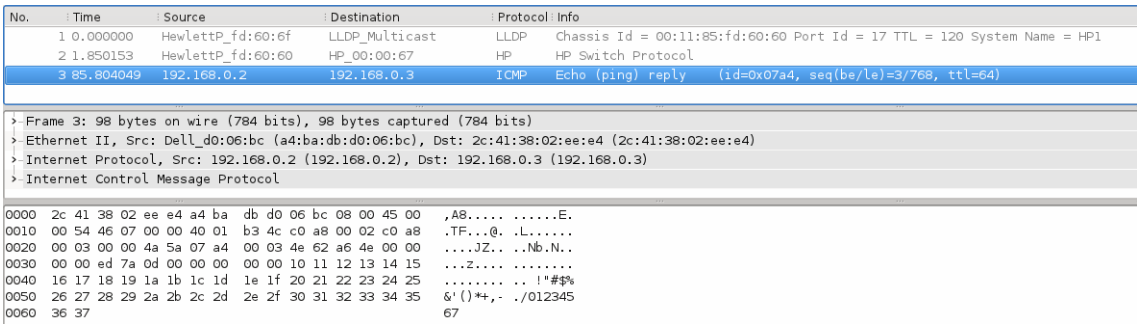
38. ábra A csomag útvonala a Linux-ot futtató PC-től a célállomásig

A tesztelésnél, egy ping üzenetet küldünk el a Linuxot futtató PC-ről, ami továbbítódott az eszközmeghajtó programon keresztül FPGA-n üzemelő útvonalválasztónak, ami a megfelelő porton kiküldte a csomagot.



39. ábra A leküldendő csomag hexadecimálisan

Szintén érdemes megfigyelni, hogy a célállomás által fogadott csomag, ami a **40. ábraán** látható, megegyezik a Linuxon futó csomagküldő program által beolvasott adattal (lásd **39. ábra**).



40. ábra Megérkezett a célállomásra a kiküldött csomag

6.3 Értékelés

Kijelenthető, hogy a funkcionális vizsgálat a teljes rendszerre nézve és a módosított vagy új modulokra lebontva is sikeres volt; minden vizsgált modul teljesíti a tervben leírt követelményeket.

Emellett fontos megemlíteni, hogy sikerült egy olyan Lookup felépítést megtervezni, amihez csak minimálisan kellett a hozzá kapcsolódó egységek működését módosítani, így nem változtattuk meg a kártya működési struktúráját. Elmondható még, hogy az SWC jelenlegi állapotában az FPGA regisztereinek csak a harmadát igényli, ahogy az a **[10][D]** függelék *Number of Slice Registers* bejegyzésében látható, viszont BRAM-ból és FIFO-ból az rendelkezésre állók 97%-át felhasználja, ezért a további fejlesztéseknél ezt figyelembe kell venni.

7 Összefoglalás

Tudományos Diákköri Dolgozatunkban tehát megterveztünk, megvalósítottunk és teszteltünk egy flexibilis csomagtovábbító rendszert.

A komplex rendszer sajátossága, hogy a vezérlés és a továbbítás optimálisan szétválasztott, így megtartottuk a szoftveres útvonalválasztók rugalmasságát, és a hardveres csomagtovábbítás gyorsaságát. További előnye, hogy önmagában, azaz szoftver nélkül is működőképes, vagyis sikerült megtartani a kiindulási csomagtovábbító rendszer funkcionalitását. Ezen kívül nemcsak szoftveresen, hanem hardveresen is flexibilis az FPGA megvalósításnak köszönhetően. Így a chipen megvalósított funkciók tovább bővíthetőek, optimalizálhatóak. (Sőt lehetőség van a működés közbeni dinamikus újrakonfigurálásra is, ami azt jelenti, hogy például forgalmasabb időszakokban ki lehet cserélni a kereső algoritmusokat gyorsabbakra, illetve kisebb forgalom esetén a lassabb, de energiatakarékosabb módok is futhatnak.)

A szoftveres rész bekapcsolása után a rendszert a vezérlő táblák feltöltése szerint teljesen rugalmasan lehet kezelni. A szoftver általános interfészt biztosít Linux tábláihoz, amelyek az útvonalválasztást vezérlik. Minden változásról értesülünk az interfészen keresztül, és a változásokat a hardveren is érvényesíteni tudjuk. A tesztekben leírtakhoz hasonlóan lehetséges a vezérlőprotokollok által létrehozott változásokat is közölni a rendszerrel. Mivel a NETLINK interfész csak a táblaváltozásokkal foglalkozik, ez nem függ attól, hogy milyen útvonalválasztási metódust hoztunk működésbe a számítógépen, így a rendszer a jelenlegi és a jövőbeni fejlesztésű protokollokkal is kompatibilis. Ráadásul a megoldásunk illeszkedik a ForCES keretrendszerbe, tehát megoldásunk nemcsak nyílt forrású megoldáson alapuló, de szabványos is. Mindazonáltal, hogy mások is felismerték a NetLink protokoll használhatóságát IP szolgálati protokollként [3], tudomásunk szerint más hasonló implementáció nem született.

A rendszer nemcsak az útvonalválasztó protokollokkal tehető kompatibilissé, hanem módosításokkal Ethernet kapcsoló implementációkkal is együtt futhat a rendszer, ez azonban azt feltételezi, hogy a Linux kapcsoló implementációk támogassák a NETLINK protokollt.

8 Irodalomjegyzék

[1] L. Yang, R. Dantu, T. Anderson, R. Gopal, “Forwarding and Control Element Separation (ForCES) Framework”, IETF RFC 3746, April 2004.

[2] H.Khosravi, T. Anderson, - “Requirements for Separation of IP Control and Forwarding”, IETF RFC 3654, November 2003.

[3] J. Salim, H. Khosravi, A. Kleen, A. Kuznetsov, “Linux Netlink as an IP Services Protocol”, IETF RFC 3549, July 2003.

[4] Wei-Ming Wang, Li-Gang DongBin Zhuge, „Analysis and Implementation of an Open Programmable Router Based on Forwarding and Control Element Separation”, July 8, 2008.

[5] György, Horváth. cBoard User's Guide. Budapest : AITIA, 2010. old.: 2

[6] J. Biswas, et al., “Application Programming Interfaces for Networks”, IEEE P.1520 Whitepaper, January 1999

[7] GNU Zebra, online: http://www.zebra.org/

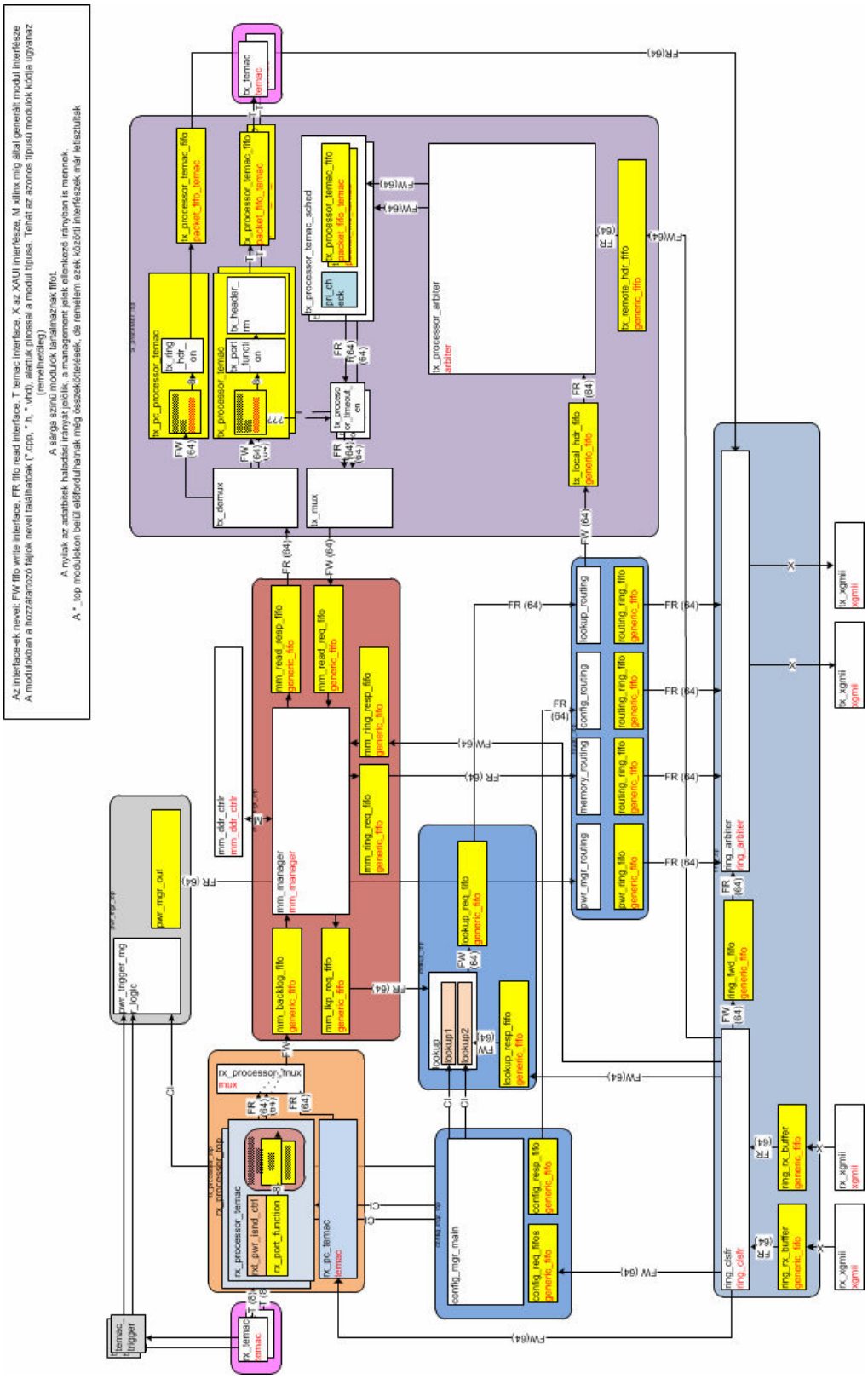
[8] XORP: Xtensible Open Router Platform, online: http://www.xorp.org/

[9] A. Udugama, “Manipulating the Networking Environment Using RTNETLINK”, Linux Journal, Mar 30, 2006, Online: http://www.linuxjournal.com/article/8498

[10] Roland Bless, Klaus Wehrle, “Evaluation of Di_erentiated Services using an Implementation under Linux”, IWQOS’99, London, Jun 1 - June 4, 1999

9 Függelék

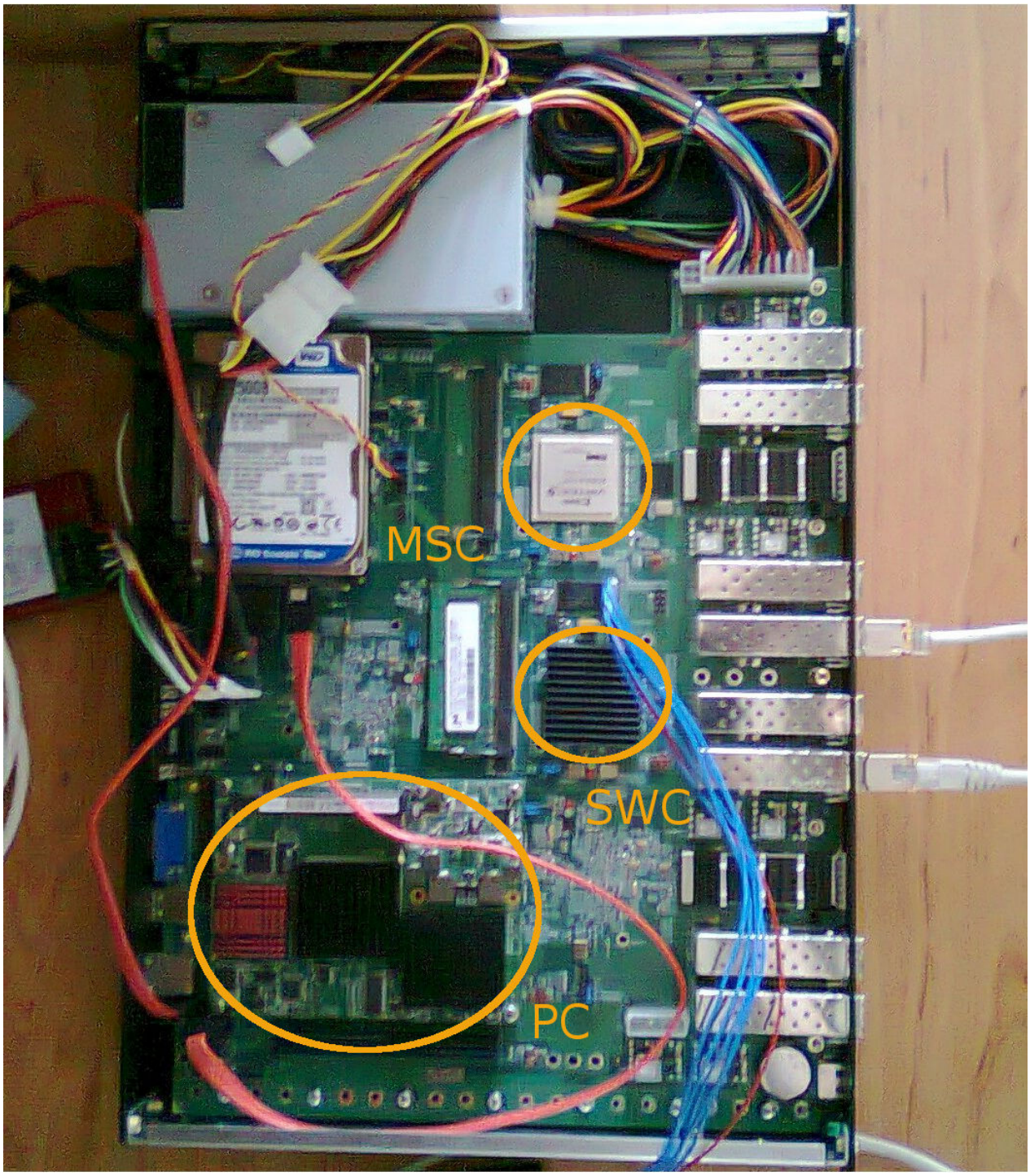
[A] A módosított általános csomagtovábbítási rendszer:



[B] A leküldő és felolvasó függvények prototípusai:

```
int SetRegister  (HANDLE fd, unsigned long dwAddress, unsigned long dwValue);
int GetRegister  (HANDLE fd, unsigned long dwAddress, unsigned long *dwValue);
int Dump_wr (HANDLE fd, char *data, int length);
int Norm_hdr_to_char(GPFF_RING_HDR_NORMAL header, char *data);
int Dump_wr_norm(HANDLE fd, char *data,  GPFF_RING_HDR_NORMAL header);
int Conf_hdr_to_char(GPFF_RING_HDR_CONFIG header, char *data);
int Dump_rd (HANDLE fd, GPFF_PACKET *data, int *length);
int Dump_wr_conf(HANDLE fd, char *data,  GPFF_RING_HDR_CONFIG header);
```

[C] A Cboard felülnézeti képe:



[D] Az SWC erőforrás felhasználása:

```
Design Summary
--------------
Slice Logic Utilization:
  Number of Slice Registers:              21,303 out of  69,120   30%
    Number used as Flip Flops:            21,302
    Number used as Latches:                    1
  Number of Slice LUTs:                   25,104 out of  69,120   36%
    Number used as logic:                 23,528 out of  69,120   34%
    Number used as Memory:                 1,376 out of  17,920    7%
    Number used as exclusive route-thru:     200
  Number of route-thrus:                   1,730

Slice Logic Distribution:
  Number of occupied Slices:              11,359 out of  17,280   65%
  Number of LUT Flip Flop pairs used:     32,769
    Number with an unused Flip Flop:      11,466 out of  32,769   34%
    Number with an unused LUT:             7,665 out of  32,769   23%
    Number of fully used LUT-FF pairs:    13,638 out of  32,769   41%
    Number of unique control sets:         1,312
    Number of slice register sites lost
      to control set restrictions:         2,324 out of  69,120    3%

IO Utilization:
  Number of bonded IOBs:                     118 out of     640   18%
    Number of LOCed IOBs:                    118 out of     118  100%
    IOB Flip Flops:                          267
    Number of bonded IPADs:                   22
    Number of bonded OPADs:                   18

Specific Feature Utilization:
  Number of BlockRAM/FIFO:                   145 out of     148   97%
    Number using BlockRAM only:               20
    Number using FIFO only:                  125
    Total Memory used (KB):                2,772 out of   5,328   52%
  Number of BUFG/BUFGCTRLs:                   10 out of      32   31%
    Number used as BUFGs:                     10
  Number of IDELAYCTRLs:                       4 out of      22   18%
  Number of BSCANs:                            1 out of       4   25%
  Number of BUFDSs:                            2 out of       8   25%
```

```
Number of GTP_DUALs:                          5 out of       8   62%
  Number of LOCed GTP_DUALs:                  5 out of       5  100%
Number of PLL_ADVs:                           1 out of       6   16%
```