Írta:
FODOR ATTILA
VÖRÖSHÁZI ZSOLT

BEÁGYAZOTT RENDSZEREK ÉS PROGRAMOZHATÓ LOGIKAI ESZKÖZÖK

Egyetemi tananyag

TYPOTEX

2011

LEKTORÁLTA: Dr. Keresztes Péter, Széchenyi István Egyetem Műszaki Tudományi Kar Automatizálási Tanszék

TÁMOGATÁS:
Készült a TÁMOP-4.1.2-08/1/A-2009-0008 számú, „Tananyagfejlesztés mérnök informatikus, programtervező informatikus és gazdaságinformatikus képzésekhez” című projekt keretében.

ISBN 978-963-279-500-3

KÉSZÜLT: a Typotex Kiadó gondozásában
FELELŐS VEZETŐ: Votisky Zsuzsa
AZ ELEKTRONIKUS KIADÁST ELŐKÉSZÍTETTE: Juhász Lehel

KULCSSZAVAK:
beágyazott rendszerek, CAN, FPGA, MicroBlaze, MODBUS, RTOS, VHDL.

ÖSSZEFOGLALÁS:
Napjainkban a legtöbb elektronikus eszköz kisebb önálló működésre is alkalmas részegységből áll, amelyek egy beágyazott rendszert alkothatnak. Az első fejezet röviden összefoglalja a beágyazott rendszerek alapelveit, a második az FPGA (Field Programmable Gate Arrays) eszközökkel és azok fejlesztői környezetével foglalkozik, amelyet a beágyazott rendszerekben lehet használni.

Az FPGA-k – a felhasználó által többször „tetszőlegesen” konfigurálható kapuáramkörök – különböző feladatok megvalósításának igen széles spektrumát biztosítják: az algoritmusok végrehajtásának gyorsítását szolgáló tudományos számításoknál, a jelfeldolgozásban, a képfeldolgozásban, a titkosítás vagy az autóipari alkalmazások stb. területén. A hagyományos ASIC VLSI technológiával szembeni nagy előnyük a relatív olcsóság, a gyors prototípus-fejlesztési lehetőség, és nagyfokú konfigurálhatóság.

A jegyzetben a legnagyobb FPGA gyártó, a Xilinx chipek általános felépítését, funkcióit, egy a Xilinx Spartan-3E sorozatú FPGA-ra épülő Digilent Nexys-2-es fejlesztő kártyát, integrált tervező-szoftver támogatottságát és programozási lehetőségeit ismertetjük.

Célul tűztük ki, hogy a VHDL összes nyelvi konstrukciójának áttekintése helyett mindvégig az FPGA-s környezetekben alkalmazható és szintetizálható tervek leírására fókuszálunk, amely egyben szimulálható RTL szintű VHDL megadását is jelenti. Nem a VHDL nyelvi szintaxisának mélyreható ismertetését követjük, hanem mindig egy konkrét gyakorlati példán keresztül mutatjuk be a VHDL nyelvi elemkészletének a szintézis szempontjából legfontosabb részeit.

Tartalomjegyzék

Rövidítések

ABS	Anti-lock Braking System – Blokkolásgátló fékrendszer
ABEL	Advanced Boolean Expression Language
ACK	Acknowledge – Nyugtázás
API	Application Program Interface
ASIC	Application Specific Integrated Circuit – Alkalmazás specifikus IC v. Berendezés orientált áramkörök
ASIL	Automotive Safety Integrity Level – Integrált Autóipari Biztonsági Szint
BRAM	Blokk RAM – Dedikált, véletlen hozzáférésű memóriatömb
CAL	CAN Application Layer – CAN Alkalmazási Réteg
CAN	Controller Area Network
CBSE	Component-Based Software Engineering – Komponens Alapú Szoftver-fejlesztés
CLB	Configurable Logic Block – Konfigurálható Logikai Blokk
CMOS	Complementer MOS
CPLD	Complex Programmable Logic Device – Komplex, programozható logikai eszközök
CRC	Ciklikus Redundancia Check
DCM	Digital Clock Manager – Digitális órajel menedzser (áramkör)
DSP	Digital Signal Processor – Digitális jelfeldolgozó processzor
EBD	Electronic Brakeforce Distribution – Elektronikus Fékerő Szabályozó Rendszer
ECU	Electronic Control Unit – Elektronikus Vezérlő Egység
EDA	Electronic Design Automation – Elektronikai tervezés-automatizálás
EDIF	Electronic Digital Interchange Format – Elektronikai tervek közös digitális formátuma
ESP	Electronic Stability Program – Elektronikus Menetstazbilizátor
FCFS	First Come First Served
FMEA	Failuremode and Effect Analysis
FPGA	Field Programmable Gate Arrays – Felhasználó által programozható kapu-áramkörök

FSM	Finite State Machine – Véges állapotú automata
GPIO	General Purpose IO – Általános célú I/O
HDL	Hardware Description Language – Hardver Leíró Nyelv
HIL	Hardware-in the- Loop
IEEE	Institute of Electrical and Electronics Engineers – Villamosmérnökök és elektrotechnikusok nemzetközi szervezete
IP	Intellectual Property – Szellemi termék
ISO	International Standardisation Organisation – Nemzetközi Szabványügyi Hivatal
JTAG	Joint Test Advisory Group
LIN	Local Interconnect Network
LRC	Longitudinal Redundancy Check – Longitudinális Redundancia Ellenőrzés
LUT	Look-Up Table – Keresési tábla/Logikai függvénygenerátor FPGA-n
LVDS	Low Voltage Differential Signals – Kis-feszültségű differenciális jelek
MAC	Multiply And Accumulate – Szorzat-összeg képző áramköri elem
MBD	Model-Based Development – Model Alapú Fejlesztés
MDD	Model-Driven Development – Model Vezérelt Fejlesztés
MFQ	Multilevel Feedback Queues
MISO	Master In Slave Out
MISRA	Motor Industry Software Reliability Association
MOSI	Master Out Slave In
MSB	Most Significant Bit – Legnagyobb Helyiértékű Bit
MULT	Multiplier – Dedikált szorzó
NACK	Negative Acknowledge – Negatív Nyugta
OOP	Object Oriented Programming – Objektum Orientált Programozás
OSI	Open Systems Interconnection
PA	Parking Assistant – Parkolás Könnyítő Rendszer
PDA	Personal Digital Assistant
POF	Plastic Optical Fiber – Optikai Kábel
PROFIBUS	Process Field Bus
PROFIBUS-DP	Process Field Bus Decentralized Peripherals
PROFIBUS-FMS	Process Field Bus Fieldbus Message Specification
PROFIBUS-PA	Process Field Bus Decentralized Peripherals
RAM	Random Access Memory
RR	Round-Robin
RS-232	Revised Standard 232
RS-422	Revised Standard 422
RS-485	Revised Standard 485

RTC	Real Time Clock – Valós Idejű Óra
RTL	Register Transfer Language/Level – Regiszter átviteli szint
RTOS	Real-Time Operating System – Valós Idejű Operrációs Rendszer
RTU	Remote Terminal Unit
SCI	Serial Communication Interface – Soros Kommunikációs Interfész
SDS	Smart Distributed System
SIL	Softver In the Loop
SJF	Shortest Job First
SMQ	Static Multilevel Queue
SOC	System On-a Chip – Egychipes rendszer
SOPC	System On-a Programmable Chip – Egychipes programozható rendszer
SPI	Serial Peripherial Interface – Soros periféria interfész
SRTF	Shortest Remaining Time First
TCB	Task Controll Block – Taszk Vezérlő Blokk
TSP	Trailer Stability Program – Utánfutó Menetstabilizátor Program
USART	Universal Synchronous / Asynchronous Receiver Transmitter – Univerzális szinkron / aszinkron adó-vevő
UCF	User Constraints File – Felhasználó által megadott feltételek (kényszerek)
VHDL	VHSIC (Very High Speed Integrated Circuit) HDL – Nagyon-nagy sebességű Integrált Áramkörök Hardver Leíró nyelven támogatott tervezése
XST	Xilinx Synthesis Tool – Xilinx Szintézis Eszköz

Felhasznált FPGA eszközök és fejlesztő környezetek

Demonstrációs és oktatási célokból a jegyzetben konzisztens módon a Xilinx ISE™ ingyenesen elérhető WebPack™ fejlesztő környezetének [*WEBPACK*] aktuális változatát (12.2) és a beépített ISim integrált HDL szimulátorát, illetve az Xilinx EDK beágyazott fejlesztő rendszerét használjuk a Digilent Inc. Nexys-2 FPGA-s fejlesztő kártyáján, amely egy Xilinx Spartan™-3E FPGA-t tartalmaz. Ugyan a példák specifikusan ezen az FPGA-s kártyán lettek bemutatva és tesztelve, természetesen más Xilinx FPGA-kkal, vagy más gyártók (pl. Altera, Actel, Quicklogic stb.) FPGA-ival és azok fejlesztő környezeteivel is használhatóak, bizonyos fokú módosítások után. A példákon keresztül az FPGA-k, mint újrakonfigurálható számítási eszközök alkalmazásának széles spektrumát mutatjuk be röviden, bevezető jelleggel egyszerű kapu-szintű VHDL viselkedési és strukturális leírásoktól, fontosabb VHDL nyelvi szerkezetekről a Xilinx MicroBlaze™ 32-bites beágyazott szoft-processzorán át, egészen az I/O perifériákig bezárólag (pl. VGA), amelyek mindegyike a mai FPGA-alapú beágyazott rendszerek egy-egy fontosabb komponensét képezheti.

Veszprém, 2011. március 31.

Bevezetés

Napjainkban a legtöbb elektronikus eszköz az kisebb önálló működésre is alkalmas részegységből áll, amelyek valójában egy beágyazott rendszert alkotnak. A beágyazott rendszerek ismerete nélkülözhetetlen egy mai mérnök számára. A jegyzet ehhez igyekszik segítséget nyújtani. A jegyzet két fejezetre osztható, az első fejezet röviden összefoglalja a beágyazott rendszerek alapelveit, a második fejezet pedig az FPGA eszközökkel és azok fejlesztői környezetével foglalkozik.

Napjainkban az FPGA-knak (Field Programmable Gate Arrays) – a felhasználó által többször „tetszőlegesen” konfigurálható, újrakonfigurálható kapuáramkörök, mint nagyteljesítményű számítási eszközök a különböző feladatok megvalósításának igen széles spektrumát biztosítják, akár az algoritmusok végrehajtásának gyorsítását szolgáló tudományos számításoknál, a jelfeldolgozásban, a képfeldolgozásban, a titkosítás vagy, akár az autóipari alkalmazások stb. területén, főként ott, ahol a feladatok párhuzamos végrehajtására van lehetőség. A hagyományos ASIC VLSI technológiával szembeni előnyük a relatív olcsóság, a gyors prototípus-fejlesztési lehetőség, és nagyfokú konfigurálhatóság. Ebben a jegyzetben a legnagyobb FPGA gyártó, azaz a Xilinx chipek általános felépítését, funkcióit, illetve egy kiválasztott Xilinx Spartan-3E sorozatú FPGA-ra épülő Digilent Nexys-2-es fejlesztő kártyát, integrált tervező-szoftver támogatottságát, és programozási lehetőségeit ismertetjük.

Napjainkban a legtöbb létező könyv és jegyzet a HDL nyelvek pontos szintaktikai elemkészletének bemutatására és szimulálható, de nem mindig szintetizálható kódok megadására törekszik.

Jelen jegyzet készítése során célul tűztük ki, hogy a VHDL összes nyelvi konstrukciójának áttekintése helyett mindvégig az FPGA-s környezetekben alkalmazható és szintetizálható tervek leírására fókuszálunk, amely egyben szimulálható RTL szintű VHDL megadását is jelenti. Továbbá sosem a VHDL nyelvi szintaxisának mélyreható ismertetését követjük, hanem mindig egy konkrét gyakorlati példán keresztül mutatjuk be a VHDL nyelvi elemkészletének a szintézis szempontjából legfontosabb részeit.

A jegyzet mind mérnök-informatikusok, mind villamos-mérnökök számára korszerű, konkrét, gyakorlati ismereteket tartalmaz programozható logikai eszközökön megvalósítható beágyazott, főleg mikroprocesszoros rendszerek tervezéséhez és fejlesztéséhez.

1. fejezet

Beágyazott rendszerek

A mindennapi életben szinte folyamatosan beágyazott rendszereket használunk anélkül, hogy azok felépítésével működésével tisztába lennénk. Amikor beülünk egy autóba természetesnek vesszük annak különböző minket segítő funkcióit (ABS, ESP, EBD, PA, TSP stb.). Amikor az autóban megnyomjuk a gázpedált természetesnek vesszük azt, hogy az autó gyorsulni fog, ha megnyomjuk a fékpedált, akkor pedig azt várjuk, hogy az autó lassuljon. A modern autók esetében egy-egy ilyen egyszerű művelet közben is több tízezer sornyi program fut le. A modern autók működés közben 5-15 milliszekundumonként futtatnak le egy-egy vezérlési ciklust, ami meghatározza a gépjármű pontos menetdinamikai tulajdonságait (sebesség, megcsúszás, autó sodrodása, becsült tapadás stb.).

A különböző rendszerek bonyolultságába csak akkor gondolunk bele, amikor egy-egy ilyen beágyazott rendszer részegysége meghibásodik és a hiba pontos okát meg kell találni. Ez a fejezet az ilyen beágyazott és biztonságkritikus rendszek felépítését, tervezési és tesztelési módjait próbálja rövid tömör formában bemutatni a rendelkézésre álló hely szűkössége miatt.

Mivel a személygépjárművek fejlődése szinte összekapcsolódik a beágyazott rendszerek fejlődésével ezért a jegyzet első fejezete a legtöbb példát az autóipar területéről meríti.

1.1. Beágyazott rendszerek definíciója, követelmények, tipikus alkalmazások

A beágyazott rendszer a (számítógépes) hardver- és szoftverelemeknek kombinációja, amely kifejezetten egy adott funkciót, feladatot képes ellátni. A beágyazott rendszerek tartalmaznak olyan számítógépes eszközöket, amelyek alkalmazás-orientált célberendezésekkel vagy komplex alkalmazói rendszerekkel szervesen egybeépülve azok autonóm működését képesek biztosítani, vagy segíteni.

Az ipari gépek, gépkocsik, gyógyászati eszközök, fényképezőgépek, háztartási eszközök, repülőgépek, automaták és játékok (csakúgy, mint a látványosabb mobiltelefonok és PDA-k) közé tartoznak számtalan lehetséges host-ok a beágyazott rendszerek számára.

A programozható beágyazott rendszerek, valamilyen programozási interface-el vannak ellátva, de a beágyazott rendszerek programozása speciális szoftverfejlesztési stratégiákat és technikákat igényel.

Egyes operációs rendszereket és a nyelvi platformokat kimondottan a beágyazott rendszerek piacára fejlesztették ki, mint például a Windows XP Embedded és az EmbeddedJava.

Sok végfelhasználói termék jellemzője, hogy nagyon olcsó mikroprocesszort tartalmaz és korlátozott tárolási lehetőséggel rendelkezik, ezeknél az eszközöknél általában az egyetlen alkalmazás az operációs rendszer részét képezi. Az előre elkészített program ezeknél az eszközöknél betöltődik a RAM-ba (Random Access Memory), mint a programok a személyi számítógépeken.

Az iparban előforduló beágyazott rendszereknek folyamatosan kapcsolatban kell lenni a környezetükkel, a méréseket különböző fizikai/kémiai/biológiai elven működő szenzorokok és érzékelőkön keresztül tudják a berendezések megvalósítani. Ha ezek a rendszerek a monitorozáson kívül még valamilyen szabályozási/vezérlési funkciót is ellátnak, akkor a technológiai folyamatba be is kell avatkozniuk.

Manapság, szinte bármilyen tevékenységet végezve a mindennapi életben, valószínűleg használunk olyan terméket vagy szolgáltatást, aminek az irányadó magatartása számítógépalapú rendszer, más néven beágyazott rendszer. Ez a fejlődés érinti az autóipart is.

A több beágyazott elektronikus vezérlő biztosítja a mai járművekben az olyan jármű centrikus funkciót, mint például a motorvezérlés, fékrásegítő, blokkolásgátló stb, valamint az olyan utas központú rendszerek, mint a szórakoztatás, ülés/tükör ellenőrzés és állítás stb.

A jegyzet igyekszik azt bemutatni, hogy ez a fejlődés elkerülhetetlen volt, és igyekszik vázolni a fejlődés fő vonalait.

Először az állami szabályozás, mint például a kipufogógáz káros anyag összetételének a szabályozása vagy a kötelező aktív biztonsági berendezések (pl. légzsákok), amelyek megszabják a beágyazó rendszer összetett vezérlési szabályait, törvényeit. Második lépésként a felhasználók igényeltek kényelmesebb, könnyebben és biztonságosabban vezethető autót, továbbá az autógyártók egy új innovatív terméket akartak létrehozni, amely sokkal jobban eladható a gépjárművezetők számára. A vezetők igényei és az autógyártók igényei is növelik a gépjármű műszaki tartalmát és ezáltal a fejlesztési költségeket is növelték.

Úgy tűnik, a mai előrehaladott szoftver-technológia jó kompromisszumot képes teremteni az eszközök és a termék fejlesztési költségei között, így képes megkönnyíteni az új szolgáltatások bevezetését az autóban.

Ahhoz, hogy meghatározzuk az alkalmazott beágyazott elektronikus rendszerek követelményeit, osztályozni kell a rendszer funkcionális területeit. A technológiai megoldások, a hardver-összetevők, valamint a szoftverfejlesztés megoldásai és költségei a követelményeket is befolyásolják. A gazdasági megfontolások és megszorítások is megváltoztatják a rendszer követelményeit, az ilyen jellegű befolyásolás általában a követelmények enyhítése irányába hat. A fejlesztési költségek csökkentését a régi project-ek részegységeinek újrafelhasználásával lehet elérni. Ezért a nagy gyártók igyekeznek a fejlesztéseiket a leginkább hardver / szoftver független módon elkészíteni, így hatékony közös fejlesztéseket lehet létrehozni a beágyazott elektronikus architektúrák valamint a hardver és szoftver egységek újrafelhasználásával. A szoftverfejlesztés területén az objektum orientált programozás (Object Oriented Programming – OOP) nyújt nagy segítséget. Az autóipar területén jelen pillanatban, a CAN kommunikáció túlsúlyban van az ECU-k (Electronic Control Unit) összekapcsolásában, de

már másféle megoldások is felmerültek (például a FlexRay[1]) az ECU-k összekapcsolására és integrálására más mechatronikai rendszerrel.

Az autóba beépített szoftverek növekvő összetettsége jól mutatja a megfelelően irányított fejlesztési folyamatot. Autonóm és automatikus közúti járművek létrehozásához elengedhetetlen a kommunikáció autók és a környezetük között.

Az integrált közlekedési megoldások lesznek a fő kulcsszavai a jövő járműinek fejlesztése közben. Ezek a trendek képesek a motorizált forgalmat megfigyelni és képesek célzottan beavatkozni a forgalomba így csökkentve a zsúfoltságot, a környezetszennyezést, valamint biztonságosabbá képes tenni a közlekedést. (Gondoljunk például a Google azon megoldására, hogy a GPS-es Android-os operációs rendszerrel rendelkező telefonok kommunikálnak a Google szervereivel és folyamatosan küldik a pozíciójukat, amely segítségével meg lehet mondani, hogy melyik úton milyen sebességgel mozognak a járművek. Ezzel a megoldással el lehet kerülni azokat az utakat, amelyeken torlódás van.) Ebben az esetben a jármű fejlesztését már nem lehet külön-külön fejlesztési projectként kezelni, hanem egy bonyolult rendszer részének kell tekinteni. Az ISO 26262 szabvány foglalkozik közúti forgalom szervezésével és irányításával, a szabvány szilárd, strukturált tervezési módszereket ajánl az alkalmazóknak.

A nemzetközi kezdeményezéseknek köszönhetően új koncepciók alakultak ki autóipari rendszertervezők körében: a modell-alapú fejlesztés (Model-Based Development – MBD), a modell-irányított fejlesztés (Model-Driven Development – MDD) és a komponens alapú szoftverfejlesztés (Component-Based Software Engineering – CBSE).

A beágyazott rendszerek tervezési fázisában kell a rendszertervezőknek kidolgozniuk a rendszer pontos specifikációját, természetesen a megrendelő bevonásával. Ekkor kell lefektetni a pontos követelményeket rendszer működéséről. Természetesen az nem megengedhető, hogy a rendszer csak egy bizonyos kívánalmat teljesítsen és más követelményeket pedig ne tartson be, például a teljes rendszer számára fontos biztonsági előírások megszegése nem lehet elfogadható.

Ha a fontosabb kívánalmakat figyelembe vesszük, akkor kettő fő követelmény lehet a beágyazott rendszerknél:

- **Idő**: Egy bekövetkező esemény lereagálását a rendszer egy meghatározott időn belül kezdje el
- **Biztonság**: A rendszer feladata egy olyan rendszer vezérlése, amely hibás működés esetén egezségkárosodás vagy komoly anyagi kár következne be.

E filozófia mentén tudjuk definiálni a beágyazott rendszerek kettő fő alcsoportját:

Valós idejű rendszer, melynél az időkövetelmények betartása a legfontosabb szempont. A valós idejű rendszerekkel részletesebben foglalkozik a jegyzet *1.3. fejezete*.

- **Biztonságkritikus rendszer**, melynél a biztonsági funkciók sokkal fontosabbak, mint az időkövetelmények betartása. A biztonságkritikus rendszerekkel részletesebben foglalkozik a jegyzet *1.4. fejezete*.

[1] Jelenleg már a szériában gyártott autókban is alkalmazzák, például BMW X5. Biztonságkritikus funkciók működéséhez nem szükséges a FlaxRay működése.

A valóságban nem lehet ilyen könnyedén a rendszereket csoportosítani, mert lehetnek olyan valós idejű rendszerek is, melyek rendelkeznek a biztonságkritikus rendszerek bizonyos tulajdonságaival. A szabványok és a törvények szabályozzák azt, hogy milyen alkalmazásoknál kell kötelezően biztonságkritikus rendszert alkalmazni.

1.2. Időkezelés és adattovábbítás problémái

Az időkezelés rendkívül fontos a valós idejű rendszerek esetében, mivel egy-egy bekövetkező eseményt egy meghatározott időn belül el kell kezdeni az esemény feldolgozását. A követelmények szigorúsága alapján két féle valós idejű rendszert különböztethetünk meg: a Hard real-time és a Soft real-time rendszert. A Hard real-time rendszerek esetében szigorú követelmények vannak előírva, és a kritikus folyamatok meghatározott időn belül feldolgozásra kerülnek. Soft real-time rendszer esetében a követelmények kevésbé szigorúak és a kritikus folyamatokat a rendszer mindössze nagyobb prioritással dolgozza fel.

1.2.1. Sorrendezési és megegyezési problémák

Egy esemény a rendszerállapot detektálható, pillanatszerű változása. Előfordulhat, hogy ha két csomópont egymást követő e1 és e2 eseményről tájékoztatást ad két másik csomópontnak, akkor az üzenetek megérkezési sorrendje el fog térni az események időbeni sorrendjétől. Q csomópont a korábban bekövetkezett eseményről később szerez tudomást. Ezen relativisztikus hatás kiküszöbölése miatt fontos a csomópontok közti megegyezés.

1.2.2. Lehetetlenségi tétel

Ha A csomópont üzenetet küld B-nek, az üzenet megérkezik, de B nem lehet biztos benne, hogy A tud az üzenet sikeres megérkezéséről, ezért küld egy nyugtázó üzenetet. Ezt A megkapja, de ő sem lehet benne biztos, hogy B tud az üzenet sikeres megérkezéséről. Lehetetlenségi tétel: Véges meghibásodású csatornáról nem lehet hibátlan kommunikációt feltételezni.

1.2.3. Bizánci tábornokok problémája (Byzantine generals problem)

Bizonyos biztonságkritikus rendszereknél több érzékelőt, vezérlőt esetleg több számítógépet használnak ugyanannak a jelnek, folyamatnak a megmérésére, vezérlésére feldolgozására. Erre azért van szükség, hogy valamely részegység meghibásodása esetén is működőképes maradjon a rendszer (redundancia). A probléma az az, hogy mi a teendő akkor, ha egy rendszernél egy érzékelő teljesen más értéket mér, mint a többi érzékelő.

A probléma szemléltetésére létezik egy rövid tanmese, amely a következő képpen hangzik: négy szomszédos hegyen sorban táborozik 1000, 2000, 3000 és 4000 katona, míg köztük a völgyben táborozik 5000 ellenséges katona. Az adatok alapján látszik, hogy a hegyiek csak összefogással győzhetik le a völgybelieket. A hegylakók szeretnék szövetségben megtámadni a völgybelieket, de nem mindegyikük mond igazat, mert azt hazudja, hogy több katonája van a ténylegesnél. Jelen esetben tehát az adat érvényességével van a baj, nem a kommunikációval.

Mindegyik csomópont (vezér) elküldi a többieknek, hogy mennyi katonája van, feltételezzük, hogy a 3-as hazudik, minden üzenetben mást mond, jelöljük ezeket x, y, z-vel. Az egyes csomópontok a következő üzeneteket kapták:

1. (1, 2, x, 4), 2. (1, 2, y, 4), 3. (1, 2, 3, 4), 4. (1, 2, z, 4)

Ezután elküldik egymásnak a kapott üzeneteket, 3-as ismét mindenkinek mást hazudik:

1. a következő üzeneteket kapja: (1, 2, y, 4), (a, b, c, d), (1, 2, z, 4)
2. a következő üzeneteket kapja: (1, 2, x, 4), (e, f, g, h), (1, 2, z, 4)
4. a következő üzeneteket kapja: (1, 2, x, 4), (1, 2, y, 4), (i, j, k, l)

Így már kiszűrhető, hogy 1000 + 2000 + 4000 = 7000 katona biztosan van, tehát megkezdhetik a támadást. A helyes döntés meghozatalához m megbízhatatlan egység esetén 3m+1 iterációra van szükség. Ezt a szabály a redundáns rendszerek esetében is használható.

1.2.4. A Bizánci-probléma

A biztonságos üzenetküldés és kommunikáció alapvető követelménye, hogy a küldő meggyőződjön arról, hogy az elküldött üzenetet a vevő fél megkapta, vagyis mindkét fél biztos legyen abban, hogy egy adott művelet végrehajtódott. Itt a veszélyt az jelenti, ha manipuláció vagy hibás átvitel eredményeként az egyik fél úgy gondolja, hogy a művelet sikeresen végbement. A probléma az, hogy hogyan tud a művelet sikeréről meggyőződni a küldő fél. Látszólagos egyszerűsége ellenére ezt a követelményt nehéz biztosítani.

A nyugtázott üzenetküldés problémáját a szakirodalomban a *Bizánci-problémaként* szoktak említeni. A történelmi példa alapján két szövetséges hadsereg két szemközti domb tetején foglal állást, az ellenség pedig a völgyben helyezkedik el. Az ellenség létszám fölényben van, így mindkét dombon lévő hadsereggel képes lenne elbánni külön-külön, viszont ha a szövetséges hadseregek egyszerre tudnának támadni a völgyben állomásozó ellenséges hadsereget, akkor győzelmet tudnak aratni felette.

A két szövetséges hadvezér csak futárok segítésével az ellenséges vonalakon keresztül tud kommunikálni egymással. A futárokat azonban elfoghatja az ellenség, így nem biztos, hogy az üzenet megérkezik. A kérdés az, hogy: megoldható-e valamilyen kommunikációs módszerrel az, hogy a két hadvezér meg tud e egyezni a támadás időpontjában.

Indirekt bizonyítással és teljes indukcióval egyszerűen belátható, hogy ilyen megoldás nem létezik. Tételezzük fel, hogy véges n lépésben (n futár küldése után) meg tudnak egyezni a hadvezérek. Ekkor viszont az n-ik lépésben küldött futár elfogásának esetén arra a következtetésre kellene jutnunk, hogy már az (n-1)-ik lépésben is tudniuk kellett volna a hadvezéreknek a támadás időpontjában. Véges lépésben ellentmondásra jutunk, ha az (n-1)-ik lépésre ugyanezt a gondolatmenetet alkalmazzuk, ez azt jelenti, hogy az első futár küldésekor tudni kellett volna a támadás időpontját. A kiindulási helyzet viszont az volt, hogy nem tudják a hadvezérek a támadás időpontját.

A történelmi példa alapján láthatjuk, hogy a legrosszabb esetet feltételező üzenetvesztés esetén nem létezik olyan biztonságos nyugtázott üzenet küldés, amely során mindkét fél meggyőződhet arról, hogy egy egyeztetés sikeres volt. Ha valamilyen természetes, nem rosszindulatú üzenetvesztést tételezünk fel (például az átviteli csatornán lévő zaj miatt), akkor az üzenet továbbításának sikerességét valamekkora valószínűséggel jellemezhetjük.

Abban az esetben ha egy üzenetküldés biztonságos nyugtázását valamilyen valószínűséggel meg tudjuk oldani, akkor az gyakorlatban *biztonságosan megoldható probléma.*

Ha rosszindulatú, intelligens támadást feltételezünk, akkor a támadó az üzenet egyeztetés módszerét is ismeri. Ebben az esetben a Bizánci problémánál látott bizonyítás alapján egy támadó bármely kifinomult protokoll esetén elnyelhet bizonyos üzeneteket. Így valamelyik felet kétségek között tudja tartani. Ebben az esetben nem létezik elméleti és nem létezik gyakorlati megoldás sem. Ez a probléma csak mindkettő fél által hitelesített harmadik fél bevonásával oldható csak meg.

1.2.5. Ütemezés

A feladatok közül a valós idejű operációs rendszerek számára kritikus az ütemezés és az erőforrásokkal való gazdálkodás megvalósítása. Mivel minden rendszer, valamilyen periféria segítségével kommunikál a környezetével, ezért fontos e perifériák valós idejű rendszer követelményeinek megfelelő módon történő kezelése. Az ütemezés és az erőforrásokkal való gazdálkodás azért kiemelt fontosságú, mert egy-egy esemény kezelésekor a válaszidő betartásához az eseményt lekezelő utasítás sorozatot végre kell hajtani. Az utasítássorozat lefutása erőforrásokat igényel, melyeket az operációs rendszernek biztosítani kell, ez úgy valósítható meg a leggyorsabban, ha az operációs rendszer folyamatosan rendelkezik szabad erőforrásokkal, melyeket oda tud adni az időkritikus folyamatoknak.

A CPU ütemezésnek különböző szintjeit tudjuk megkülönböztetni:

- Hosszútávú (long term) ütemezés vagy munka ütemezés
- Középtávú (medium term) ütemezés
- Rövidtávú (short term) ütemezés

Nem minden általános célú operációs rendszerben van mindegyik ütemezés megvalósítva.

A hosszú távú ütemezés feladata, hogy a háttértáron várakozó, még el nem kezdett munkák közül meghatározza, hogy melyek kezdjenek futni, a munka befejeződésekor ki kell választania egy új elindítandó munkát. A hosszútávú ütemezést végző algoritmusnak ezért ritkán kell futnia.

A középtávú ütemezés az időszakos terhelésingadozásokat hívatott megszüntetni, hogy a nagyobb terhelések esetében ne legyenek időtúllépések. A középtávú ütemező algoritmus ezt úgy oldja meg, hogy bizonyos (nem időkritikus) folyamatokat felfüggeszt illetve újraaktivál a rendszer terhelésének a függvényében. Folyamat felfüggesztése esetén a folyamat a háttértáron tárolódik, az operációs rendszer elveszi a folyamattól az erőforrásokat, melyeket csak a folyamat újraaktiválásakor ad vissza a felfüggesztet folyamatnak.

Rövidtávú ütemezés feladata, hogy kiválassza, hogy melyik futásra kész folyamat kapja meg a CPU-t. A rövidtávú ütemezést végző algoritmus gyakran fut le, ezért gyorsan kell lefutnia. Mivel gyakran lefut az algoritmus, ezért az operációs rendszer mindig a memóriában tartja az ütemező kódját. Az operációs rendszerek magja tartalmazza az ütemezőt.

Az általános célú és a valós idejű operációs rendszerek a CPU ütemezésben különböznek leginkább egymástól. Ennek az oka az, hogy a valós idejű operációs rendszereknek az eseményeket meghatározott időn belül le kell reagálnia, egy általános célú operációs rendszer esetében nincsenek ilyen jellegű követelmények.

A CPU ütemezéssel kapcsolatban a következő fogalmakat értelmezhetjük:

- **CPU löket (CPU burst):** A folyamatnak csak CPU és az operatív tár kell
- **Periféria löket (I/O burst):** Perifériás átvitelt hajt végre a folyamat, nincsen szükség CPU-ra

Ütemezés során a folyamatokkal a következő esemény következhet be:

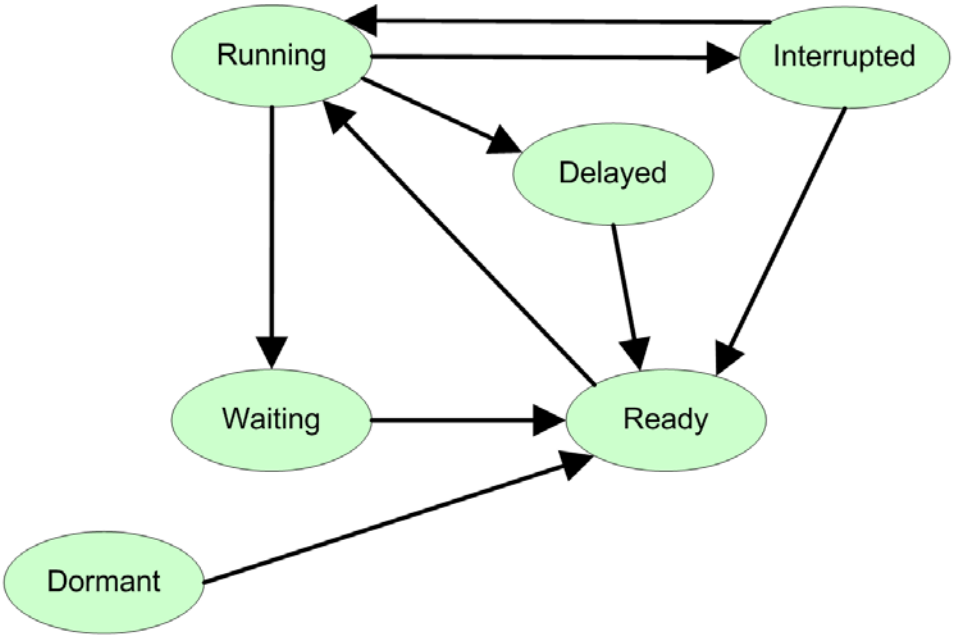
- A futó folyamat várakozni kényszerül (Például: I/O-ra, erőforrásra).
- A futó folyamat befejeződik.
- A futó folyamat lemond a CPU-ról.
- A futó folyamattól az eperációs rendszer elveszi a CPU-t.
- A folyamat aktiválódik, futásra késszé válik.

Az ütemezéssel és a programokkal kapcsolatban a következő alapfogalmakat értelmezhetjük:

- Task: Önálló részfeladat.
- Job: A task-ok kisebb, rendszeresen végzett feladatai.
- Process: A legkisebb futtatható programegység, egy önálló ütemezési entitás, amelyet az operációs rendszer önálló programként kezel. Van saját (védett) memória területe, mely más folyamatok számára elérhetetlen. A task-okat folyamatokkal implementálhatjuk.
- Thread: Saját memóriaterület nélküli ütemezési entitás, az azonos szülőfolyamathoz tartozó szálak azonos memóriaterületen dolgoznak.
- Kernel: Az operációs rendszer alapvető eleme, amely a task-ok kezelését, ütemezést és a task-ok közti kommunikációt biztosítja. A kernel kódja hardware függő (device driver) és hardware független rétegekből épül fel. A hardware függő réteg új processzorra és eszközökre történő adaptálását az operációs rendszer átportolásának nevezzük.

A Task-ok állapota a futás közben a következő állapotokat veheti fel:

- Dormant: Passzív állapot, amely jelentheti az inicializálás előtti vagy felfüggesztett állapotot.
- Ready: A futásra kész állapotot jelöli. Fontos a task prioritási szintje és az is, hogy az éppen aktuálisan futó task milyen prioritási szinttel rendelkezik, ezek alapján dönti el az ütemező, hogy elindítja e a taskot.
- Running: A task éppen fut.
- Delayed: Ez az állapot akkor lép fel, mikor a task valamilyen időintervallumig várakozni kényszerül. (Rendszerint szinkron időzítő szolgáltatás hívása után következik be.)
- Waiting: A task egy meghatározott eseményre várakozik. (Ez rendszerint valamilyen I/O művelet szokott lenni.)
- Interrupted: A task-ot megszakították, vagy a megszakítás kezelő rutin éppen megszakítja a folyamatot.



1.1. ábra: A Task állapotok változásai

A task-ok állapotát és tulajdonságait a Task Vezérlő Blokk (Task Controll Block – TCB) írja le, amely a memóriában lévő adatszerkezet, fontosabb tagjai a következők:

- Task ID: Egy egész szám, amely a task-ot azonosítja.
- Context: Program Counter, a regiszterek és flag-ek elmentett értékei. (A task futásának helyreállításához szükségesek ezek az információk.)
- Top of Stack: Egy mutató, amely megadja a task-hoz tartozó verem tetejét
- Status: Egy egész szám, amely utal a task aktuális státuszára.
- Priority: A prioritás aktuális értéke, amely a futás közben megváltoztatható.
- I/O Information: Milyen perifériákat és I/O-kat foglalt le és használ a task. A nem használt perifériákat minden esetben fel kell szabadítani.

A ütemezési algoritmusoknak két fő típusa van, ezek a kooperatív (nem preemptív) és a preemptív algoritmusok. Először a kooperatív multitask-ot valósították meg nagy gépes környezetben. A működési elve és alapötlete a kooperatív algoritmusoknak az, hogy egy adott program vagy folyamat lemond a processzorról, ha már befejezte a futását vagy valamilyen I/O műveletre vár. Ez az algoritmus addig működik jól és hatékonyan, amíg a szoftverek megfelelően működnek (nem kerülnek végtelen ciklusba) és lemondanak a CPU-ról. Ha viszont valamelyik a program/folyamat nem mond le a CPU-ról vagy kifagy (és ez miatt nem mond le a CPU-ról), akkor az egész rendszer stabilitását képes lecsökkenteni vagy akár képes az egész rendszert kifagyasztani. A kooperatív algoritmus ezért soha nem fordul elő valós idejű operációs rendszerek esetében.

A preemptív algoritmusok esetében az operációs rendszer részét képező ütemező algoritmus vezérli a programok/folyamatok futását. A preemptív multitask esetén az operációs rendszer elveheti a folyamatoktól a futás jogát és átadhatja más folyamatoknak. A valós idejű operációs rendszerek ütemezői minden esetben preemptív algoritmusok, így bármely program

vagy folyamat leállása nem befolyásolja számottevően a rendszer stabilitását. Az ütemező algoritmusok operációs rendszerek rendeltetése alapján más rendszerjellemzőkre vannak optimalizálva. Az ütemezési algoritmusok teljesítményét a következő szempontok alapján tudjuk osztályozni:

- CPU kihasználtság (CPU utilization): Azt mondja meg, hogy a CPU az idejének hány százalékát használja a folyamatok utasításainak végrehajtására.
- CPU üres járása (Idle): A CPU idejének hány százalékában nem hajt végre folyamatot. (Ilyenkor indíthatóak hosszú távú (long term) ütemezésben szereplő folyamatok.)
- Átbocsátó képesség (Throughput): Az operációs rendszer időegységenként hány folyamatot futtat le.
- Körülfordulási idő (Turnaround time): A rendszerbe helyezéstől számítva mennyi idő alatt fejeződik be egy process
- Várakozási idő (Waiting time): Egy munka (vagy folyamat) mennyi időt tölt várakozással
- Válaszidő (Response time): Időosztásos (interaktív) rendszereknél fontos, azt mondja meg, hogy a kezelői parancs/beavatkozás után a rendszer első válaszáig eltelt idő.

Az ütemezési algoritmusokkal szembeni követelményeket különbözőképpen tudjuk csoportosítani. Rendszerenként változhat az, hogy a megvalósításkor melyik követelményt választják fontosnak és melyiket kevésbé. Az algoritmusokkal szemben támasztott fontosabb követelmények a következők:

- Optimális: Legyen optimális a rendszer viselkedése, azaz valamilyen előre meghatározott szempontok figyelembe vételével működjön a rendszer.
- „Igazságos”: Ne részesítse előnybe azonos paraméterekkel rendelkező process-ek közül semelyiket sem.
- Prioritások kezelése: Legyen képes az algoritmus arra, hogy process-eket folyamatokat a prioritásuk alapján egymástól megkülönböztessen
- Ne „éheztesse ki” a folyamatokat: Minden process kapjon valamennyi processzor időt, ezáltal biztosítva azt, hogy folyamatos legyen a futás
- Viselkedése legyen megjósolható: Minden esetben legyen a rendszer viselkedése előre kiszámítható, hogy a mérnökök előre modellezni tudják, hogy a rendszer hogyan fog viselkedni.
- Minimális rendszeradminisztrációs idő
- Graceful degradation: Ez a rendszer túlterhelése esetén fontos szempont, mert a rendszer viselkedés szempontjából az a fontos, hogy „fokozatosan romoljon le” a rendszer teljesítménye. A valós idejű operációs rendszerek esetében kritikus milyen csökkenés engedhető meg, mert a rendszernek tartani kell a valós idejű rendszer specifikációjában meghatározott időket.

1.2.6. Klasszikus ütemezési algoritmusok

Az ütemezési algoritmusokat csoportosíthatjuk felépítésük és működésük alapján. A különböző operációs rendszerek használhatóságát nagyban befolyásolja az ütemező algoritmus működése. A jegyzet ezeket az algoritmusokat nem tárgyalja részletesen a rendelkezésre álló hely

szűkössége miatt. A klasszikus ütemezési algoritmusok közül a jegyzet a következőket tárgyalja:

- Egyszerű algoritmusok
  - Legrégebben várakozó (First Come First Served, FCFS):
  - Körforgó (Round-Robin, RR)
- Prioritásos algoritmusok
  - Statikus prioritás
  - Legrövidebb (löket)idejű (Shortest Job First, SJF)
  - Legrövidebb hátralévő idejű (Shortest Remaining Time First, SRTF)
  - Legjobb válaszarányú
- Többszintű algoritmusok
  - Statikus többszintű sorok (Static Multilevel Queue, SMQ)
  - Visszacsatolt többszintű sorok (Multilevel Feedback Queues, MFQ)
- Többprocesszoros ütemezés

Legrégebben várakozó (First Come First Served – FCFS): Az új folyamatok a várakozási sor végére kerülnek, mindig a sor elején álló folyamat kezd futni. A process-ek nem szakíthatóak meg (Nem preemtív az ütemező, így valós idejű rendszerhez nem használható.) Az algoritmus előnye az, hogy egyszerűen megvalósítható. Az algoritmus hátránya az, hogy egy hosszú ideig futó process feltartja az egész rendszert (Konvojhatás)

Körforgó (Round-Robin – RR): Az időosztásos operációs rendszerek algoritmusainak alapja. Csak időszeleteket kapnak a process-ek (time slice), amelyek után az ütemező átadja a vezérlést egy másik process-nek, így az algoritmus preemtív módon üzemel. Abban az esetben, ha a CPU löket kisebb, mint az időszelet, akkor a process lefut és átadja a vezérlést egy másik process-nek. Abban az esetben, ha a CPU löket nagyobb, mint az időszelet, akkor az időszelet után felfüggesztésre kerül a process és az ütemező átadja a vezérlést egy másik process-nek.

Prioritásos ütemező algoritmusoknál a folyamatokhoz az ütemező hozzárendel egy prioritás értéket és a legnagyobb prioritású folyamat lesz a következő futtatandó. Ezeknél az algoritmusoknál megkülönböztethetünk statikus és dinamikus prioritásos algoritmusokat. A statikus prioritásos algoritmusoknál a folyamatok kiéheztetése léphet fel, ezért a folyamatokat öregíteni (aging) kell.

Legrövidebb (löket)idejű (Shortest Job First – SJF) algoritmus a dinamikus prioritásos algoritmusok közé tartozik. Az algoritmus a várakozó folyamatok közül a legrövidebb löketidejűt indítja el.

Legrövidebb hátralévő idejű (Shortest Remaining Time First – SRTF) algoritmus szintén dinamikus prioritásos algoritmus. Ha egy új folyamat érkezik, akkor az ütemező megvizsgálja a process-ek hátralévő löketidejét és a legrövidebb hátralévő idejű process-t indítja el.

A legjobb válaszarányú algoritmus is dinamikus prioritásos algoritmus. Ez az SJF algoritmus egy változata, a várakozó folyamatok közül nem a várakozási idő alapján választ, hanem egy speciális algoritmus segítségével.

A többszintű algoritmusok esetében a process-ek több sorban várakoznak (például: rendszer, megjelenítés, batch folyamatok stb.). Minden sorhoz prioritást rendel az ütemező algoritmus. A sorokon belül különböző kiválasztási algoritmusok is használhatóak. A többszintű

algoritmusoknak kettő fő típusa van: statikus többszintű sorok (process nem kerülhet át másik sorba) és a visszacsatolt többszintű sorok (process átkerülhet másik sorba).

A hatékony többprocesszoros ütemezés a mai processzorok esetében elengedhetetlen, mivel a jelenleg piacon kapható számítógépek már több maggal rendelkeznek sőt, még a beágyazott alkalmazásokhoz fejlesztett számítógépek is. A többprocesszoros ütemezést több CPU-val rendelkező rendszerekben vagy több magos/szálas CPU-k esetében lehet használni. Az ütemezési algoritmusokat kettő csoportra bonthatjuk: heterogén és homogén rendszerek. Heterogén rendszer esetében egy folyamat csak 1 CPU-n futhat.

Homogén rendszer esetében az induló folyamat a rendszer közös sorába kerül. Homogén ütemezés esetében beszélhetünk aszimmetrikus és szimmetrikus rendszerről. Aszimmetrikus rendszer esetén egy közös (meghatározott CPU-n futó) ütemező van, míg szimmetrikus rendszer esetében minden CPU-nak saját ütemezője van.

1.2.7. Task-ok közötti kommunikáció

Mivel a rendszer működése közben a task-ok egymással párhuzamosan futnak ezért gondoskodni kell arról, hogy egyazon I/O-t, perifériát vagy memória területet két vagy több task ne használjon egyszerre, mert abból hibás rendszerműködés alakulna ki. A taszkok közötti kommunikációra a következő módszerek állnak rendelkezésre a programozók számára:

- Szemafor (semaphore), mely 1 bit információ átadására alkalmas.
- Események (event flags), melyek több bit információ kicserélésére is alkalmasak.
- Postaláda (mailbox), amely komplexebb struktúra átadására szolgál.
- Sor (queue), amely több mailbox tömbében tartalom átadására szolgál.
- Cső (pipe), amely direkt kommunikációt tesz lehetővé két taszk között.

A szemafor az egy absztrakt adattípus, amelyet leginkább közös erőforrásokhoz való hozzáférés kontrollálására (kölcsönös kizárás) használnak. Alkalmas ezen kívül még esemény bekövetkeztének jelzése, két task tevékenységének összehangolására és kettő vagy több task szinkronizálására is. Szemafor típusai a következők lehetnek:

- Bináris szemafor (binary semaphore), amely egyszerű igaz-hamis jelzésre szolgál. Csak egyetlen erőforrás vezérlésére használható.
- Számláló szemafor (counting semaphore): A szemaforhoz egy számot rendelünk, működés közben a szemafor wait() művelete blokkol, ha a számláló 0 érékűre változik. Ellenkező esetben eggyel csökkenti a számláló értékét. A szemafor signal() művelete eggyel növeli a számlálót.
- Erőforrás szemafor (resource semaphore): Csak az a taszk engedhető el, amelyik lefoglalta az adott perifériát. Közös erőforrás védelmére jó, de taszkok közötti szinkronizációra nem alkalmas.
- Mutex: Egy bináris szemafor, mely kibővített tulajdonságokkal rendelkezik.

A következő példa egy szál létrehozását, elindítását, leállítását mutatja, a kommunikáció közben mutex-et használ fel a program a változó eléréséhez.

```
// A szálat tartalmazó függvény forráskódja
UINT ThreadProcProximity (LPVOID pParam)
{
    CClass &MainClass = *((CClass *)pParam) ;

    while (1)
    {
            ::WaitForSingleObject (MainClass.m_ProxMutex,
                                   INFINITE) ;
            if (MainClass.m_TerminateThread)
                return (TRUE) ;
            Card = MainClass.LoadActCard(i);
            //..

            if (MainClass.m_TerminateThread)
                return (TRUE) ;
            ::ReleaseMutex (MainClass.m_ProxMutex) ;
            Sleep(SLEEP);
    }
    return (TRUE);
}
//Az adatokat tartalamzó osztály
class CClass
{
    HANDLE m_ProxMutex;
    BOOL m_TerminateThread;
    BOOL m_RunThread;
    int StopThread();
    int StartThread();
    CClass();

    DWORD WaitMutex(void) {return ::WaitForSingleObject
                                    (m_ProxMutex, INFINITE);}
    DWORD ReleaseMutex(void) {return ::ReleaseMutex
                                          (m_ProxMutex);}
    // ...
};
```

```
//Az osztály konstruktora
CClass::CClass(const char ReaderNum)
{
    m_RunThread = false;
    m_TerminateThread = false;
    // ...
    m_ProxMutex = ::CreateMutex (NULL, FALSE, NULL) ;
    // ...
}

// A szál elindítását végző függvény
int CClass::StartThread()
{
    if (!m_RunThread)
    {
            ::WaitForSingleObject (m_ProxMutex, INFINITE) ;
            m_TerminateThread = false ;
            PurgeComm(m_hCom, PURGE_TXCLEAR);
            PurgeComm(m_hCom, PURGE_RXCLEAR);
            AfxBeginThread (ThreadProcProximity, this) ;
            m_RunThread=true;
            ::ReleaseMutex (m_ProxMutex) ;
    }
    return 0;
}

// A szál leállítását végző függvény
int CClass::StopThread()
{
    try
    {
        ::WaitForSingleObject (m_ProxMutex, INFINITE) ;
        m_TerminateThread=true;
        m_RunThread=false;
        ::ReleaseMutex (m_ProxMutex) ;
        return 0;
    }
    catch (CException* e)
    {
```

```
        e->Delete();
        return 1;
    }
}
```

Az event flag-ek egy-egy esemény bekövetkezésekor állnak 1-es állapotba, egy task várhat több eseményre is. Az események között különböző logikai kapcsolat állhat fenn, például: AND, OR stb.

A pipe a task-ok közötti kommunikáció megvalósítására használható, egy task több pipe-on is kommunikálhat egy időben. Az alábbi példa pipe-ok létrehozását mutatja.

```
// pipe létrehozása a gyerek process számára OUT
if ( ! CreatePipe(&g_hChildStd_OUT_Rd,
                  &g_hChildStd_OUT_Wr,
                  &saAttr, 0) )
    ErrorExit(TEXT("StdoutRd CreatePipe"));

// Leíró ellenőrzése
if ( ! SetHandleInformation(g_hChildStd_OUT_Rd,
                            HANDLE_FLAG_INHERIT, 0) )
    ErrorExit(TEXT("Stdout SetHandleInformation"));

// pipe létrehozása a gyerek process számára IN
if (! CreatePipe(&g_hChildStd_IN_Rd,
                 &g_hChildStd_IN_Wr,
                 &saAttr, 0))
    ErrorExit(TEXT("Stdin CreatePipe"));

// Leíró ellenőrzése
if ( ! SetHandleInformation(g_hChildStd_IN_Wr,
                            HANDLE_FLAG_INHERIT, 0) )
    ErrorExit(TEXT("Stdin SetHandleInformation"));

// Gyerek process elindítása
    CreateChildProcess();
```

1.2.8. Rendszerhívások (kommunikáció a kernellel)

Egy komlex rendszer működése során a projectspecifikus feladatokat mindig a „felhasználó” programok hajtják végre, melyeknek folyamatosan kommunikálniuk kell a különböző erőfor-

rásokkal. A modern operációs rendszerek esetében nem megengedhető az, hogy egy-egy program saját maga kezelje az erőforrásokat. Az erőforrások kezelése mindig az operációs rendszer feladata, a programok az operációs rendszer speciális funkcióinak a meghívásával képesek csak a rendszereszközökkel kommunikálni. Nem megengedhető az operációs rendszer megkerülése. Ahhoz, hogy a processzor az operációs rendszer magjában lévő kódrészletet hajtson végre, a következő esemény valamelyikének kell bekövetkeznie: rendszerhívás, ütemező-időzítő működése, megszakítás végrehajtása.

Rendszerhívásokat (API) az alábbi események válthatnak ki: task kontextusának mentése, kernelmódba való átkapcsolás, felhasználói hívás végrehajtása vagy visszatérés történik (user mód és kontextus visszaállítás). A rendszerhívások lehetnek szinkron és aszinkron hívások. Szinkron hívás esetén a hívó task blokkolt (delayed) állapotba kerül, amíg a kernel el nem végzi a hívott feladatot. Aszinkron hívás esetén a hívó task folytatja a munkáját és a munka végeztével fut le a rendszerhívás.

Megszakításoknak két fő csoportja van: az ütemező (timer) megszakítás és a külső megszakítás. A timer megszakítás általában hardveres megoldáson alapszik, a megszakítás hatására a következő események következnek be: időzítő események feldolgozása, a jelenleg futó task időszámlálójának a módosítása és a készenléti lista (Ready List) aktualizálása.

A külső megszakítások vagy külső esemény hatására következnek be vagy valamely folyamat generálja őket szoftveres úton. A külső megszakítások kiszolgálása lehet azonnali és lehet ütemezett is, attól függően, hogy mi generálta a megaszakítást. A külső megszakítások ki-be kapcsolhatóak (maszkolhatóak).

1.3. Valós idejű rendszerek, ütemezés, időzítések

A valós idejű rendszerek tárgyalásánál fontos definiálni azt, hogy egy rendszert mikor nevezhetjük valós idejű rendszernek. Egy rendszer akkor valós idejű, ha a rendszer interaktivitása elég egy bizonyos feladat azonnali elvégzéséhez. Ebből a definícióból már sejthető a valósidejű rendszerek fő követelménye, viszont értelmeznünk kellene azt, hogy mit értünk „azonnali elvégzésen”, ezt minden esetben meghatározott formában rögzíteni kell.

Egy rendszer valós idejű, ha egy valós időskálához kötött idő(zítési)-követelményeket támasztunk. Egy valós idejű rendszer számára előírható a reagálási idő, időzített események egymásutánja.

A követelmények szigorúsága alapján kettő féle valós idejű rendszer írható elő: a Hard real-time és a Soft real-time rendszer. A Hard real-time rendszerek esetében szigorú követelmények vannak előírva, és a kritikus folyamatok meghatározott időn belül feldolgozásra kerülnek. Soft real-time rendszer esetében a követelmények kevésbé szigorúak és a kritikus folyamatokat a rendszer mindössze nagyobb prioritással dolgozza fel.

A valós idejű rendszereknél fontos értelmeznünk a következő időzítésekkel kapcsolatos definíciókat:

Válaszidő: Egy esemény által kiváltott időzítő (trigger) impulzus és az eseményt lekezelő program indulása között eltelt idő.

Idő követelmények: Átlagos válaszidőre vonatkozó korlátozást és minden egyes válaszidőre vonatkozó előírást is előírhat

Határidő teljesített: Ha egy eseményt a megadott válaszidőkön belül elkezdte a rendszer feldolgozni. (A kezdés időpontja nem determinisztikus.)

Hibás rendszerviselkedés: Ha a válaszidők az előírt időhatáron kívül vannak.

Ha össze szeretnénk hasonlítani a Hard real-time és a Soft real-time rendszerek időkezelési filizófiáit, akkor azt tapasztaljuk, hogy a Soft real-time rendszerek az időtúllépéseket sokkal dinamikusabban kezelik. A Soft real-time rendszerek esetében előre meghatározott mértékben és gyakorisággal el lehet térni a határidőktől, úgy hogy az nem jelent hibás rendszerviselkedést. A Hard real-time rendszerek esetében viszont a határidők megsértése semmilyen esetben sem engedélyezett.

A valós idejű rendszerekkel szemben támasztott követelményeket kielégítő operációs rendszereket real-time operációs rendszernek (RTOS[2]) hívjuk. Ilyen valós idejű operációs rendszerek például a:

- QNX
- RTLinux
- RTAI
- VxWorks
- OSE
- Windows CE/eXP

Egyes rendszerekhez léteznek realtime modulok, amelyek képesek beépülni a gépen futó (operációs) rendszerbe, aminek a segítségével a számítógép képes valós idejű rendszerként üzemelni. Ilyen modulok például a NI LabVIEW Real-Time Modul és a Windows Real-Time Modul(ok) stb.

Ha általánosságban megnézzük az operációs rendszerek fő feladatait, akkor azok közül ki tudjuk választani azokat a feladatokat, amelyek fontosak egy valós idejű operációs rendszer számára. Egy általános célú operációs rendszer fő feladatai a következők:

- File kezelés
- Tárgazdálkodás
- Felhasználói felület
- Perifériák kezelés
- **Ütemezés**
- Hálózatkezelés
- **Erőforrásokkal való gazdálkodás**
- Programok és állományok védelme

[2] RealTime Operating System

1.4. Biztonságkritikus rendszerek

Biztonság kritikus felhasználásnak, rendszernek különböző alkalmazási területeik lehetnek, például a nukleáris erőművek, vasút, autóipari alkalmazások vagy a légi közlekedés. A felhasználási területtől függően szigorú rendeletek szabályozzák a biztonsági követelményeket. Ezért a megbízhatósági és biztonsági jellemzők fontos kérdést jelentenek a biztonsági tanúsítási folyamat közben.

Ez még hangsúlyosabbá vált és elsődleges fontosságú az autóipari és a gazdasági ágazatokban, mivel egyre növekvő számban vannak számítógépes alapú rendszerek. Ezek kritikus biztonsági funkciókat valósítanak meg, mint a kormányzás és a fékezés. Ezért több ajánlás és tanulmányok alapján kidolgoztak több tanúsítási szabványt: ARP 4754, RTCA/DO-178B (melyet a repüléstechnika területén használnak) vagy az EN50128 (amely a vasúti ágazatban alkalmazott). Ezek a szabványok szigorú iránymutatásokat adnak a biztonsági szempontból kritikus beágyazott rendszerekről. Mindazonáltal, ezek a szabványok alig ültethetőek át a járművek szoftver alapú rendszerei számára.

A probléma a rendszer szétbontásával (partícionálásával) oldható meg, amely során a szoftvert szétbontjuk kritikus és nem kritikus rendszerre, a kritikus részeket a megvalósítás során úgy kell megvalósítani több példányban, hogy azok eltérő szoftver komponenseket használjanak, így megoldható a rendszerben az aktív redundancia. Ezt hasonlóan meg tudjuk oldani a hardver tervezése során is, hogy ha valamilyen részegysége az áramkörnek meghibásodik, akkor még bizonyos vezérlési funkciók elérhetőek lesznek, így a rendszer csökkentett funkcionalitással még működőképes marad. (Ha funkciócsökkenés lép fel valamilyen szoftver vagy hardver egység meghibásodása esetében, akkor azokat az eszköz specifikációjában megfelelően dokumentálni kell. Bizonyos biztonságkritikus rendszerek esetében meghibásodás esetén nem megengedett a rendszer funkcionalitásainak a csökkenése, ezeknél a rendszereknél egymástól működési elvben is különböző megoldást kell keresni és implementálni.)

Az autóipari szektorban a Motor Industry Software Reliability Association (MISRA), amely megába foglalja az autóipari beszállítók és gyártók jelentősebb szereplőit, és kidolgozta az IEC 61508-at, amely az autóipari biztonságkritikus fedélzeti szoftverfejlesztést szabályozza, a szabvány az elektronikus és a programozható elektronikus áramköröket tartalmazó egységekre vonatkozik.

A szabvány feladata az, hogy támogatására a tanúsítási eljárást az autóiparban, és azt a gyártók számára még egyszerűbbé tegye. Jelenleg is fejlesztik azt az IEC szabványt, ami a közeljövőben fog megjelenni, amely arra szolgál, hogy az autóipari igényeket a lehető legjobban kielégítse.

Az ISO nemzetközi szabvány (ISOWD 26262) tervezete jelenleg fejlesztés alatt áll az EU-ban, az Egyesült Államokban, és Japánban. A szabvány következő lépése abból áll majd, hogy az ISO Association tagjai a felhasználás során keletkező tapasztalatait felhasználva pontosítják majd a szabvány különböző pontjait. Az ISOWD 26262 szabványt alkalmaznak a működési biztonság területén, amelynek célja, hogy minimálisra csökkentsék az esetlegesen hibás rendszer veszélyét. Az ISO tervezete a funkcionális biztonságot szavatolja a felhasználók számára: „… A jármű hibás működés eredményeként nem kerülhet olyan előre nem látható végállapotba, melynek az eredménye és viselkedése ésszerűen előre nem látható

rendellenes használatot okozna.” Ez a meghatározás a termék és a rendszer teljes életciklusára vonatkozik!

A biztonsági ellenőrzés hatékonysága nagyban függ a rendszer kialakításának a kezdeti fázisától (különösen a veszélyelemzéstől és a kockázatértékeléstől). Az ellemzést és ellenőrzést el kell végezni a fejlesztés során (funkcionális, biztonsági követelmények, hardver és szoftver, valamint rendszer evolúció), a működés közbeni szolgáltatások és a megsemmisítés (újrafelhasználás) közben is. Minden fázisban vizsgálni kell, hogy a biztonsági értékelések és veszélyességi elemzések helytállóak.

Miután a rendszerfunkciókat egy olyan biztonsági folyamat határozza meg, amely egy előre elkészített listát tartalmaz különböző vezetési helyzetekről és a hozzájuk tartozó üzemzavarokról. A lista egyértelműen meghatározza azt, hogy a járműnek hogyan kell viselkednie bizonyos helyzetekben és bizonyos meghibásodások esetében. A listát úgy kell elkészíteni, hogy a viselkedés megjósolható legyen azokban az esetekben is ha valamilyen vezérlési funkció végrehajtása közben egy vagy több részegysége meghibásodik a rendszernek. A listát úgy kell elkészíteni, hogy ne legyen olyan állapot/meghibásodás/bemeneti jel kombináció, amelyet a végrehajtási lista ne írna le megfelelően. A listát mindig úgy kell elkészíteni, hogy minden egyes közlekedési helyzetben fent lehessen tartani a jármű biztonságos és jósolható viselkedési módját.

Minden ilyen helyzetet az események függvényében kell kiértékelni, a kár (egészségügyi és gazdasági) súlyossága, ezt jelenleg egy emberi személy (a vezető) képes leginkább eldönteni és értékelni, ezért az elektronikus rendszerek számára a legfontosabb dolog a járműviselkedés ellenőrizhetőségének és irányíthatóságának fenntartása. Ezek alapján a rendszereket jellemzi egy úgynevezett Autóipari Biztonsági Integritás Szint (Automotive Safety Integrity Level – ASIL).

Ha megnézzük az IEC61508-as szabványt minden alkalmazott SIL (Softver In the Loop) tesztnek két biztonsági része van:

- a funkcionális követelmények tesztelése, azaz a biztonsági integritás attribútumok figyelése miközben az ECU-ra (Electronic Control Unit) nem adnak hibás jeleket (a hiba események bekövetkezésének egy bizonyos küszöbérték alatt kell lennie (például: kisebb, mint 10^{-8})),
- az implementáció során ha egy rendszer megvalósít egy funkciót, akkor meg kell győződni arról, hogy a rendszer többi tagja biztosítja-e az összes szükséges információt a funkció végrehajtásához.

A vezérlési funkciók mellett ellenőrzési tevékenységek is implementálva vannak a biztonságkritikus rendszerekben, például a hibamód és hatás elemzés (Failuremode and Effect Analysis – FMEA), vagy a hiba fa vagy esemény fa elemzés stb.

Ezeket a technikákat használják a rendszerek validációja és verifikációja közben is. Számos technikát lehet még a fejlesztésbe és tesztelésbe beilleszteni, amelyek a fejlesztési folyamat szakaszaitól függenek és fel lehet használni a használt modell ellenőrzésére, a teljesítmény értékelésére, ütemezés és időzítés elemzésére, hardver-in-the-loop (HIL), model-in-the-loop (MIL) és system-in-the-loop tesztek alkalmazásához stb.

1.5. Kommunikációs protokollok

A fejezet feladata a beágyazott rendszerekhez használható kommunikációs protokollok és kommunikációs módok rövid áttekintése. Sajnos a rendelkezésre álló hely szűkössége miatt a protokollok nem teljes részletességgel szerepelnek a jegyzetben. A fejezet leginkább azokra a protokollokra fókuszál, melyek FPGA-s, mikrokontorolleres környezetben szerepelnek. A bemutatás során először az olvasó azokat a protokollokat ismerheti meg, amelyek kis távolságokra (IC-k között) használhatóak, a fejezet végén pedig a nagyobb távolságokra használható kommunikációs módokat. (A számítógéphálózatokhoz használt protokollok bemutatása nem szerepel a jegyzetben.)

A kommunikációs protokollokat különböző szempontok alapján csoportosíthatjuk: használható maximális távolság, gyorsaság, hibatűrés, átviteli közeg stb. A protokollok fontosabb tulajdonságai alapján tudjuk kiválasztani, hogy egy adott feladathoz melyik protokollt lehet használni, hogy az információátvitel zavartalan legyen.

1.5.1. I²C busz

A Philips fejlesztette ki egyszerű kétirányú, kétvezetékes buszrendszert hatékony IC-k közötti vezérlésre azért, hogy mind a rendszertervezők, mint a készülékgyártók kihasználhassák a busz alkalmazásában rejlő előnyöket, valamint maximalizálhassák a hardver hatékonyságát. A buszt Inter IC-nek, vagy csak röviden I²C- busznak nevezik. Napjainkban már nem csak a Philips gyártmányú IC-k használják az I²C- buszt.

Minden I²C-busszal kompatibilis eszköz tartalmaz egy on-chip interfészt, ami az I²C buszon lehetővé teszi a többi eszközzel a kommunikációt. Ez a tervezési koncepció számos illesztési problémát old meg digitális vezérlésű áramkörök tervezésekor, ezzel megkönnyítve a tervezők munkáját.

Az I²C- busz legfontosabb jellemzői közé tartozik, hogy csak két buszvezeték szükséges a működéséhez: egy soros adatvonal (SDA) és egy soros órajel (SCL). Minden buszhoz csatlakoztatott eszköz programból címezhető egy egyedi címmel, melynek egy része a gyártó által megadott, a másik részét pedig áramköri huzalozással lehet beállítani. Az SDA és az SCL vezeték egy felhúzó ellenálláson keresztül a pozitív tápfeszültségre van kötve. Ha a busz szabad, mindkét vezeték magas logikai szintű.

A kommunikációt soros, 8 bit-es, kétirányú adatforgalom jellemzi, melynek sebessége normál üzemmódban 100 Kbit/s, gyors üzemmódban pedig 400 Kbit/s.

A chipben beépített zavarszűrőt találunk, mely az adatvonalon lévő zavarokat szűri ki, megőrizve ezzel az adatintegrítást. A buszra csatlakoztatható integrált áramkörök számát csak a busz kapacitása korlátozza, ami maximum 400pF lehet.

I²C buszos csatlakozás lehetőséget nyújt egy olyan prototípus kifejlesztésére, ahol a módosítás és a továbbfejlesztés megvalósítható az IC-k egyszerű rákötésével, vagy levételével.

Nagy előny az I²C buszos eszközök használatánál, hogy nem kell megtervezni a busz interfészt, mert a chip már tartalmazza azt, ezáltal a tervezési idő is csökken. Jóval egyszerűbb a hibakeresés és a hibadiagnózis, a problémák azonnal felderíthetőek a buszon lévő forgalom monitorozásával, amelyhez sajnos speciális eszköz szükséges. Ugyanazok az IC típusok, sok

eltérő alkalmazásban használhatóak. A tervezési idő is lerövidül, mert a tervezők hamar rutint szereznek a gyakran használt funkcionális blokkok I²C- busszal kompatibilis IC-ként való alkalmazása miatt.

Egy beágyazott rendszer egy vezérlője általában legalább egy mikrokontrollerből és további perifériaeszközökből áll, mint például memóriák, I/O bővítések, kijelző stb.

Egy fejlett rendszer megvalósításához, soros buszstruktúra alkalmazása ajánlott. Annak ellenére, hogy a soros buszok nem rendelkeznek olyan átviteli sebességgel, mint a párhuzamos buszok, de kevesebb vezetéket és IC lábat igényelnek, így hatékonyabb az alkalmazása.

Ahhoz, hogy az eszközök kommunikálni tudjanak egy soros buszon, rendelkezniük kell a protokoll adatküldési formátumának az implementációjával, amely megakadályozza az adatvesztést. Az implementáció során figyelembe kell venni a következőket:

- A gyors eszközöknek tudniuk kell kommunikálni a lassú eszközökkel.
- A rendszer nem függhet a csatlakoztatott eszközöktől, máskülönben a változtatások és javítások nem lehetségesek.
- Egy előre definiált eljárásnak kell lennie arra, hogy melyik eszköz és mikor vezérli a buszt.
- Ha különböző eszközök különböző sebességgel kapcsolódnak a buszra, akkor azonosnak kell lennie a busz órajel forrásának.

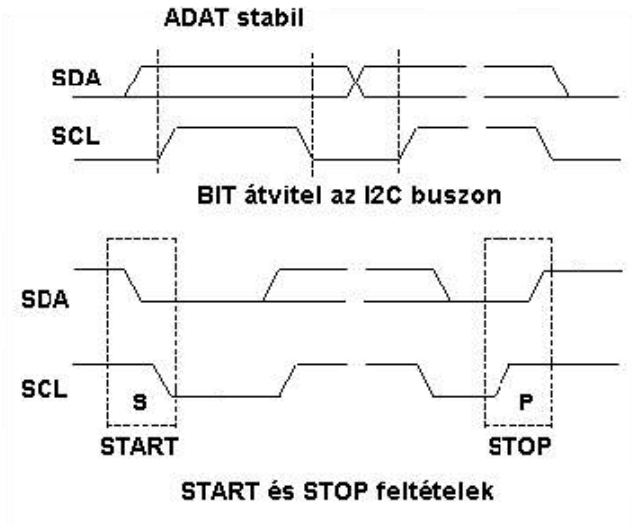
Ezeket a kritériumokat mind magába foglalja az I²C- busz specifikációja. Az I^2C busz hardveresen nem igényel sok alkatrészt. Minden eszköz egy egyedi címmel rendelkezik és működhet akár küldő, akár fogadóként is az eszköz funkciótól függően. Például egy kijelző meghajtó IC csak fogadó, míg egy memória IC tudja az adatokat fogadni és küldeni is.

Amikor az eszközök adatátvitelt valósítanak meg, tekinthetjük őket master-nek és slave-nek. A master az az eszköz, amely megkezdi az adatátvitelt a buszon és generálja az órajeleket az átvitel lebonyolításához. Az ezalatt megcímzett eszközöket slave-nek nevezzük.

Az I^2C buszon egyszerre több mikrokontroller is lehet, amelyek mindegyike képes lehet vezérelni a buszt. Ilyenkor előfordulhat az is, hogy egyszerre több master próbálja meg kezdeményezni az átvitelt, ennek az elkerülésére találták ki az arbitrációs eljárárást, amely az I^2C interfészek buszon történő ÉS kapcsolásával valósul meg.

Ha több master próbál információt küldeni a buszon, akkor az első olyan master elveszti az arbitrációt, amelyik logikai magas (1-es) értéket akar küldeni, miközben a többi logikai alacsony (0-ás) értéket. Az arbitráció ideje alatt az órajel a masterek órajeleinek a szinkronizált kombinációja, amely az SCL vezetéken létrejött huzalozott ÉS kapcsolat segítségével valósul meg. Az I²C buszon az órajel generálása mindig a masterek feladata.

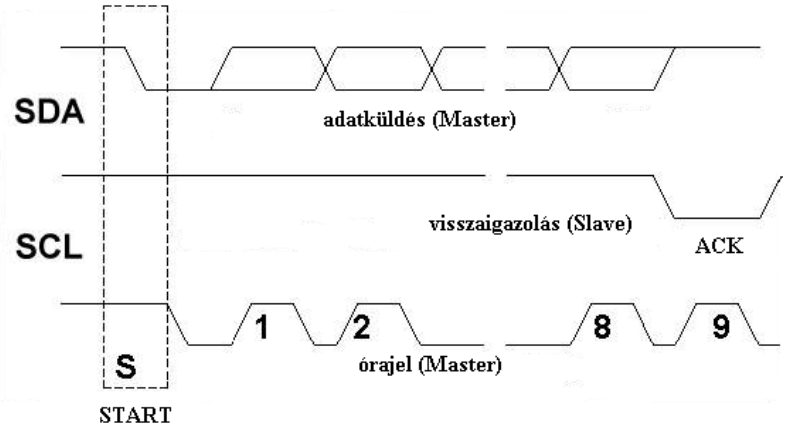
A huzalozott ÉS függvény megvalósításához az eszközök buszra csatlakozó kimeneti fokozatainak nyitott drain-nel vagy nyitott kollektorral kell rendelkeznie. Egy bit átvitele a következő módon történik. Kiinduláskor magas feszültség szinten lévő SDA adatvonalra kerül az átviteli érték, a logikai alacsony vagy magas értéknek megfelelő 0 V vagy 5 V-os feszültség. Az SCL vonal magas szintje alatt érvényes az adat, vagyis a vevő csak ekkor olvashatja, az adat csak az SCL vonal alacsony szintje alatt változhat, ilyenkor a vevő nem olvashat a buszról.



1.2. ábra: Egy bit átvitele, START és STOP feltétel a buszon

START jel akkor küldhető a buszon, ha a busz inaktív, azaz az SDA és az SCL vezeték is magas szintű. A busz aktív lesz a START bit küldése után (SCL magas állapotában az SDA vonalon egy magas állapotból alacsony állapotba történő átmenet van (SCL=1, SDA=1-ből 0 értékűre vált). A STOP bit küldése a következő módon történik: az SCL magas állapotában az SDA vonalon egy alacsonyból magasba való átmenet van (SCL=1, SDA = 0-ból 1 értékűre vált). A START és STOP biteket csak a mester eszköz hozhatja létre. A busz csak a START és STOP bitek küldése között aktív, a STOP jel után újból szabaddá válik a busz és inaktív állapotba kerül.

A busz adatátvitel byte szervezésű. Az átvitt byte-ok száma tetszőleges lehet. Az adónak visszaigazolást kell kapnia arról, hogy a vevő sikeresen fogadta az elküldött adatot. A slave egység minden sikeresen fogadott byte vétele után egy alacsony szintű nyugtázó (ACK, Acknowledge) bitet küld. Az órajelet ebben az esetben is a master egység generálja, ilyenkor a master az SDA vonalat nagy impedanciásra állítja, hogy olvasni tudja a slave nyugtázását. A nyugtázáskor a slave egység lehúzza az SDA vonalat. Az átvitel mindig a legmagasabb helyértékű (MSB, Most Significant Bit) bittel kezdődik.



1.3. ábra: Egy byte átvitele az I^2C buszon

Az adatbiteket a master, az ACK bitet a slave egység küldi. Ha a slave nem képes egy byte-ot fogadni, akkor az ACK bit küldése elmarad, és az SCL vonalat alacsony szinten tartja és egy várakozó állapotba kerül a busz. Az adatvonalon az információáramlás közben a master és a slave egység is adóként viselkedik, az órajelet viszont minden esetben a master egység generálja. (Ha az információáramlás iránya olyan, hogy a slave-től kérdezi le a master az adatokat, akkor a slave küldi az adatokat a master egység pedig nyugtázza a vételt. Az órajelet ebben az esetben is a master egység generálja.)

Két esetben fordul csak elő az adatátvitel során, hogy nem szűkséges nyugtázás:

- A mester a vevő egység, ekkor az adónak valahogy jelezni kell a byte sorozat végét, ezt úgy teszi meg, hogy a küldőnek nem küld nyugtázást (ACK). Az ACK jelhez kapcsolódó órajelet természetesen a master generálja, de az SDA vonalat nem húzza le alacsony szintre. Ezt hívjuk negatív nyugtázásnak (NACK).
- A slave akkor nem küld nyugtázó (ACK) jelet, ha nem képes újabb adat byte-ok fogadására.

Adatforgalom a buszon az eszközök között

Az I²C buszon lévő minden eszköznek saját címe van, így egyértelműen beazonosíthatóvá válik minden egység. A cím hossza 7 bit, amellyel maximum 128 eszköz címezhető.

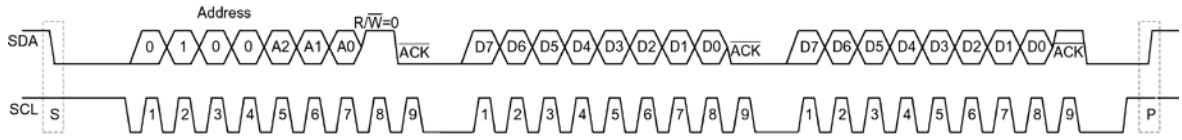
Az adatátvitelt a master kezdeményezi azzal, hogy a buszon elküld egy START bitet, ez után elküldi a buszra a slave egység címét, amelyikkel kommunikálni szeretne. Ha a slave felismeri az (egy byte hosszúságban elküldött) saját címét, akkor azt egy ACK jellel nyugtázza, ezután képes a slave adatot küldeni vagy fogadni. A nyolcadik, a legkisebb bit (LSB, Low Significant Bit) határozza meg a szolgával történő adatcsere irányát. Az alacsony logikai szint az írást jelenti (W), ilyenkor a master küldi az adatokat. A magas lokai szint az olvasást (R) jelenti.

A buszra kapcsolt eszközök címei két csoportba sorolhatók:

- programozható címmel rendelkező eszközök, amelyek általában a mikrokontrollerek
- a fix címmel rendelkező periféria áramkörök címei

Az eszközök címe két részből állhat:

- típus címből[3], amit a gyártók rendelnek az eszközökhöz
- egy hardver címből, amely az eszköz címző lábainak alacsony és magas szintre kötésével állítható.



1.4. ábra: Adatok küldése az I2C buszon

[3] A típus cím az azonos (típusú) tokokra jellemző cím és mindig megegyezik. Ezzel a címmel jelentkezik be a slave eszköz ill. ezzel a címmel szólítja meg a master eszköz a Slave -et adatcsere előtt.

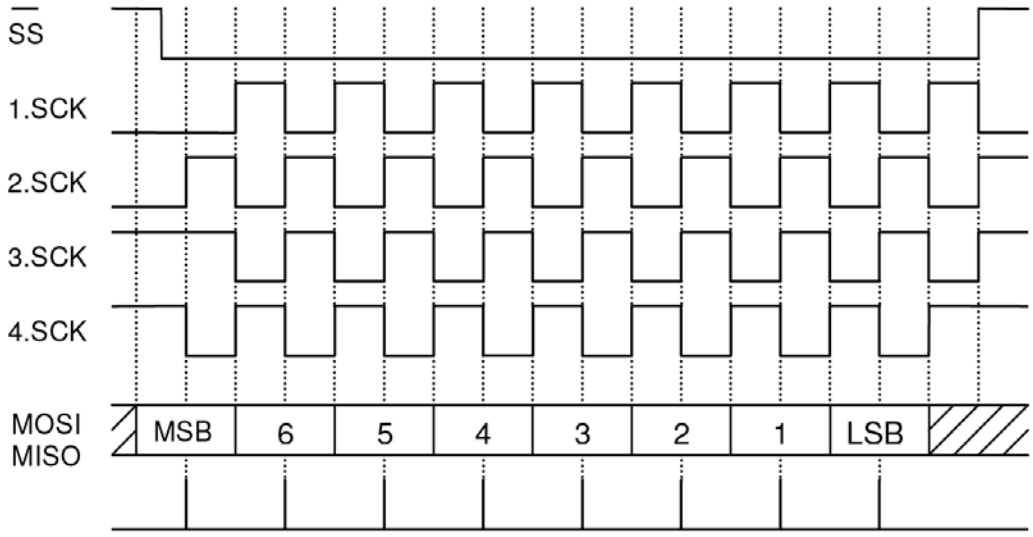
Az ábra bal oldalán látható az „S” betűvel jelölt START bit, amelyet a megfelelő hosszúságú szünet után cím mező követ, melyen jól látható a fix és a konfigurálható rész, majd az elküldött byte a kommunikáció irányát jelző bittel végződik. Ezt követi a nyugtázás (ACK), majd a kettő adatbyte következik. A komminikáció a „P” betűvel jelölt STOP bittel végződik.

1.5.2. SPI busz

Az SPI (Serial Peripherial Interface = Soros Periféria Illesztő) kommunikációs eljárást a Motorola által fejlesztette ki, az SPI busz egy nagysebességű master-slave alapokon nyugvó szinkron adatátviteli rendszer. Alkalmazása lehetővé teszi egy processzor vagy mikrokontroller és több kiegészítő áramkör, vagy más kontroller vagy CPU egységek összekapcsolását. Az órajel jellemzői, úgymint polaritás (CPOL), fázis (CPHA) szoftveresen állíthatóak. A kiegészítő áramkörök slave jellegét a szabvány rögzíti. Egy adatátviteli ciklus nyolc órajel alatt megy végbe, amely során a master küld egy adat sorozatot a megcímzett slave eszköznek és az viszont. Ezt a kétirányú adatátvitelt egyetlen órajel vezeték szinkronizálja. A busz átlagos sebessége 2 MHz-re is növelhető, azonban léteznek olyan eszközök is, amelyeknél a 4-5 MHz is elérhető. A szabvány a busz kialakításához négy vezetéket ír elő:

- adatvonal bemenetet (MOSI, Master Out Slave In),
- kimenet (MISO, Master In Slave out),
- órajel (SCK, Serial Clock),
- slave egység kiválasztását lehetővé tevő (CS, Chip Select).

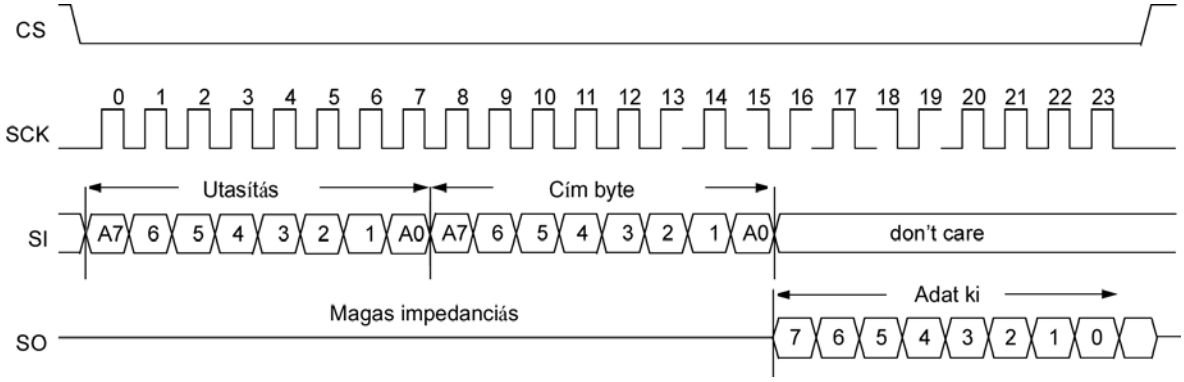
Az SPI kommunikáció során a master állítja elő a szinkron órajelet és a chip select-tel kiválasztja azt az eszközt, amellyel kommunikálni szeretne. Ezt követően kezdődik meg az adatátvitel, amely történhet az órajel felfutó és lefutó élére is, amelyet az eszköz dönt el, mert a mikrovezérlők túlnyomó többsége mindkét üzemmódot támogatja. Az SPI alkalmazásának legfőbb előnye az I^2C buszos kommunikációval szemben a gyorsaság és egyszerűbb szoftveres kezelés. Hátrányaként szerepel viszont, hogy a kommunikációhoz 4 darab adatvonal szükséges és a minden egység számára be kell húzalozni egy chip select vezetéket is.



1.5. ábra: SPI busz lehetséges vezérlési lehetőségei, fázis és polaritás alapján

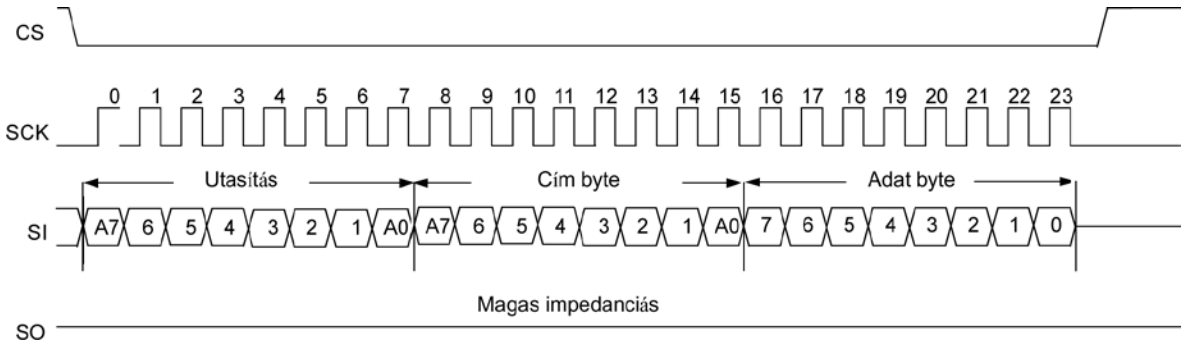
	CPOL	CPHA
1.SCK	0	0
2.SCK	0	1
3.SCK	1	0
4.SCK	1	1

Az ábrán látható, hogy az órajel 1. és 2. állapotában alacsony, a 3. és 4. pedig magas alapállapotú. Az adatokat az 1. és 4. esetben az adatot felfutó él hatására értelmezzük, míg a 2. és 3. esetben lefutó él bekövetkezése esetén olvassuk ki. A master eszközök többsége mindegyik üzemmódot támogatja, amíg a legtöbb slave eszköz csak egyik módban képes kommunikálni.



1.6. ábra: Olvasás az SPI buszról (IC regiszterének olvasása)

Az ábrán látható, hogy a master a kommunikációt a chip select alacsony szintre húzásával kezdi. Az elküldött első byte tartalmazza azt, hogy milyen utasítást szeretnénk végrehajtani az IC-vel, ezt követően kell megadni az IC regiszterének a címét. Ezek után az órajel léptetésével bitenként kiolvasható a kért információ. (Természetesen több adatbyte is kiolvasható.)



1.7. ábra: Írás az SPI buszra (IC regiszterének beállítása)

Az ábrán egy IC regiszterének az írása látható, a chip select alacsony szintre húzása után kell beállítani az utasítás kódját, majd el kell küldeni a címet tartalmazó byte-ot, ezek után kell küldeni az adatbyte-okat.

1.5.3. SPI és I^2C összehasonlítása

Az SPI és az I^2C adatátviteli eljárás használata megfelelő olyan alkalmazásoknál, ahol a perifériák elérése időben egymás után történik (például: memóriák, perifériák stb.). Azonban az SPI busz alkalmazása praktikusabb az úgynevezett adatfolyam jellegű kommunikációt igénylő folyamatoknál (például: processzorok egymással való kommunikációja, mintavételezés stb.).

Az elérhető sebesség szempontjából az SPI további előnyeként említhetjük a nagyságrenddel nagyobb, néhány MHz-es működési sebességet, míg az I^2C esetében ez csak KHz-es tartományba esik. Az SPI busz valódi előnyét a full duplex kommunikáció jelenti, amely során egyidejűleg küldhetünk és fogadhatunk adatokat, az I^2C busz félduplex megoldásával szemben.

Az I^2C előnye a kezelendő slave eszközök számában rejlik, amely maximum 128 egységet jelent. Az SPI busz több slave-vel történő használata több kiegészítő hardvert és összetettebb programkódot igényel. Az SPI busz esetében minden esetben el kell huzalozni a chip select lábakat is.

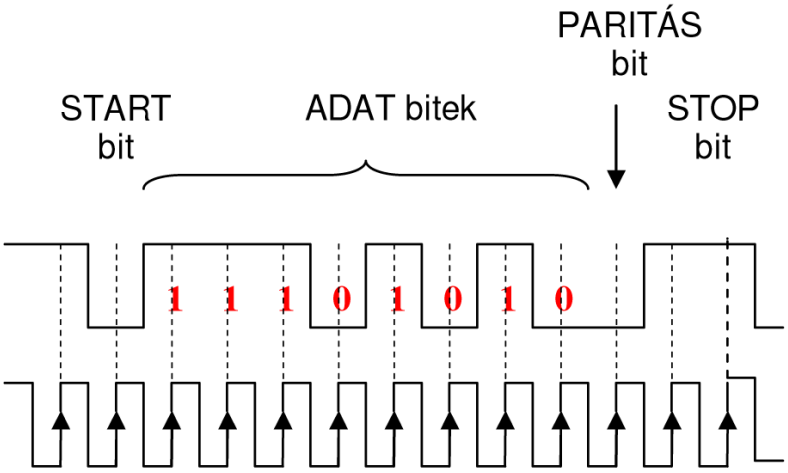
Az SPI legnagyobb hátránya az, hogy nem rendelkezik semmilyen nyugtázó mechanizmussal, vagyis a master eszköz semmilyen jelzést nem kap arról, hogy az adatátvitel sikeres volt vagy a slave eszköz egyáltalán létezik. Erre a legjobb megoldás az, hogy az IC beállított regiszterét kiolvassuk az írás után, így le tudjuk ellenőrizni az írási kommunikációs ciklust.

A pont-pont jellegű kapcsolatok esetén sokkal kezelhetőbb és hatékonyabb az SPI busz alkalmazása.

1.5.4. Aszinkron soros kommunikáció

Nagyobb távolságokra történő adattovábbítás esetén fontos szempont a kábelezés kialakítása. Kicsi mérettartományban IC-k között ez nem jelent különösebb problémát, de nagyobb buszhosszúság esetében már nem mindegy, hogy hány érrel rendelkező kábelt kell alkalmazni. Az eddig vizsgált szinkron kommunikációs módoknál szükség volt egy órajelet továbbító vezetékre, aminek a segítségével az adatátvitelnél a kommunikáció szinkronizálható volt. Az aszinkron kommunikáció ötlete az, hogy ez a szinkronizáló vezeték milyen módon spórolható meg.

Az átvitel során az adó bármely időpillanatban kezdeményezheti az adatátvitelt. Ennek azonban az a feltétele, hogy egy azonos, sebességgel kell az adónak és a vevőnek is működnie, ha ez nem teljesül, akkor az átvitt adatok értelmezhetetlenek, a sebességet szabvány rögzíti. A kommunikáció menetét a következő ábra szemlélteti. Az ábrán látszik, hogy a kommunikáció a START bittel kezdődik, amely egy magas alacsony állapot átmenet, ezt a 8 adat bit és a paritás bit (páros vagy páratlan számúra egészíti ki az adatként előforduló egyesek számát) követ, majd végül a STOP bit zárja a kommunikációt. Az időzítő a baud rate-nek a frekvenciájával, azonban egy fél órajel ciklussal eltolva mintavételezi a biteket, biztosítva az egyértelmű mintavételezést. A STOP bitet minimum három órajel szünetnek kell követnie, mielőtt a következő adat elküldhető.

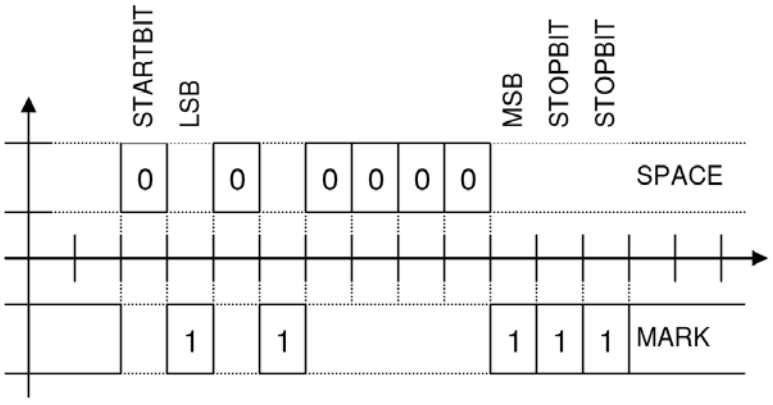


1.8. ábra: Aszinkron soros kommunikációs protokoll elvi felépítése

1.5.5. RS 232

Az RS-232 (Revised Standard 232) aszinkron soros kommunikációs szabványt az Electronic Industries Association (EIA) fejlesztette ki. Az RS-232 kialakítás napjainkban is gyakran alkalmazott szabványosított adatátviteli eljárás, amelyet lehet 9 illetve 25 pólusú csatlakozóval használni. A szabvány megalkotása során figyelembe vették azt, hogy az interfész bármely vezetékeinek összekötése során az eszközben kár ne keletkezzen.

Az RS-232 rendszer kétirányú, full duplex kommunikációs csatorna, ahol a jel egy a logikai villamos GND szinthez képest létrejövő feszültségként jelenik meg. Inaktív állapotban a jel a GND szinthez képest mérve negatív, míg aktív állapotban a jelszint pozitív.



1.9. ábra: Az „á” betű átvitele (ASCII kódja = 10000101B)

A bináris adatjelet +3V és +15V feszültség tartományban definiált logikai alacsony (SPACE), és a -15V és -3V tartományban értelmezett logikai magas (MARK) szint valósítja meg. A kimeneti jelszint +15V és -15V között váltakozik, ahol a +3V és a -3V közötti tartomány a tiltott sáv, amelynek szükségessége a zajelnyelésben nyilvánul meg. Az átvitt bitek időtartama nem lehet tetszőleges, értékét a szabvány szabályozza, amelynek értéke a bitidő

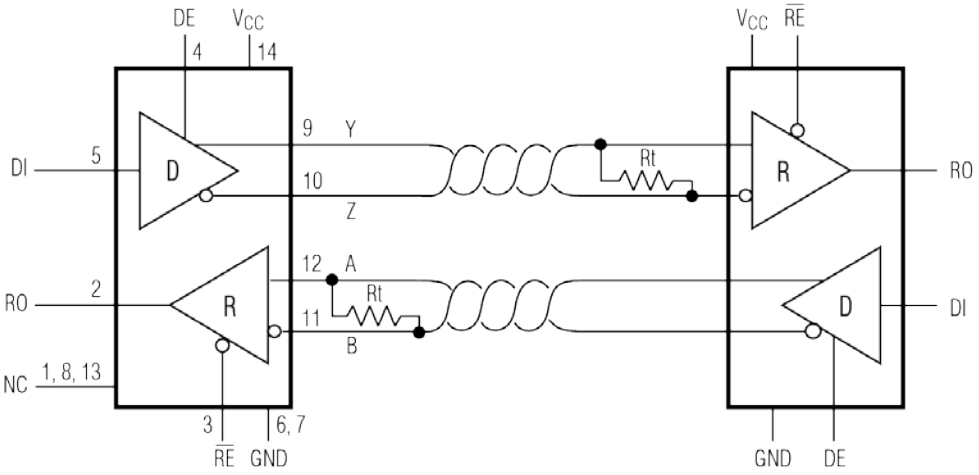
szabványsorból választható az alkalmazás igényeinek megfelelően. Az egy bit átviteléhez szükséges időtartam reciproka az úgynevezett baud rate, melynek tipikus értékei: 1200, 2400, 4800, 9600 bit/sec).

Az RS-232 kialakítás az aszimmetrikus volta miatt rendkívül nagy érzékenységgel bír a külső elektromos zavarokra, amely csökkenthető a kábel árnyékolásával, illetve kiküszöbölhető a jelek szimmetrikus RS-422 vagy RS-485 rendszerre alakításával.

A jelek átviteléhez alacsony kapacitású kábel szükséges, mivel a szabvány szerint (EIA/RS-232) egy ér maximális kapacitása 2500 pF lehet. Megfelelő kapacitású kábel (átlagos kábelek kapacitása 100 pF/m, a jobb minőségűeké, pedig 40-50 pF/m) esetén a szabvány által elérhető maximális távolság 15 m lehet. A kábel minőségén kívül másik, a rendszer kapacitását számottevően befolyásoló tényező a túlfeszültség-levezető védődiódák kapacitása, melyek kapacitása 500-1000 pF is lehet egyenként, amely jelentősen csökkenti a használható kábel hosszúságát. A 2500 pF maximális kábelkapacitást még tovább csökkenti a vevő 20 pF értékű bemeneti kapacitása.

1.5.6. RS-422

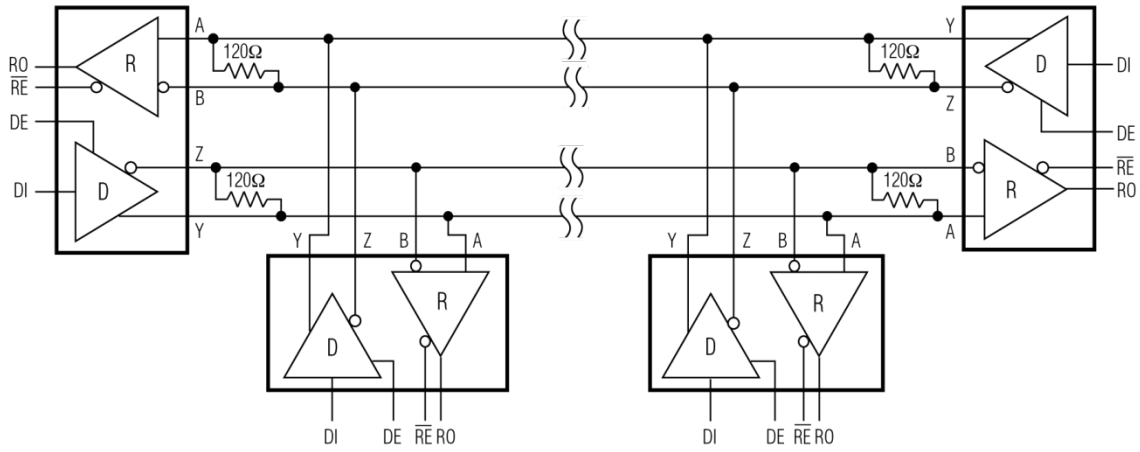
Az RS-422 egy szimmetrikus pont-pont közötti adatátvitelre kialakított rendszer, melyet az RS-232 rendszernél nagyobb távolságok áthidalására és nagyobb adatsebesség elérésére terveztek. Az ilyen rendszernek megfelelő áramkörök két, egymással nem azonos föld potenciálú vezetékkel rendelkeznek. Az RS-422 adatátvitel segítségével hálózatok is létrehozhatóak maximálisan 10 eszköz részére úgy, hogy 1 adó és 10 vevő csatlakozik a buszhoz. A szabványban kizárólag jelkarakterisztikák vannak definiálva, a csatlakozó típusok és bekötési módok nincsenek meghatározva. A szabvány 1200 m-ben határozza meg az adatátviteli távolság maximumát, amely 115 KBaud sebesség esetén reálisan mindössze 1000 m körüli, amennyiben meghajtóként egy személyi számítógép kommunikációs port-ját alkalmazzuk.



1.10. ábra: Az RS-422 kommunikációra használható meghajtó (MAX489) áramkör bekötése 2 eszköz esetén a buszt lezáró ellenállásokkal

Az RS-422 adó minden kimenetén megjelenő jelszint ±7V nagyságú feszültség. A jelszint 200 mV-ra csökkenését a vevő még érvényes jelként fogadja. Az átvitt jel két állapota a következő módon valósul meg. Ha a meghajtó 'A' kivezetése negatív a 'B'-hez képest a vonal logikai magas szintre (MARK), ellenkező esetben, vagyis ha a meghajtó A kivezetése pozitív a B-hez képest, akkor a vonal logikai alacsony (SPACE) szintre kerül. Az RS-422 rendszernél a meghajtó mindig engedélyezett állapotban van.

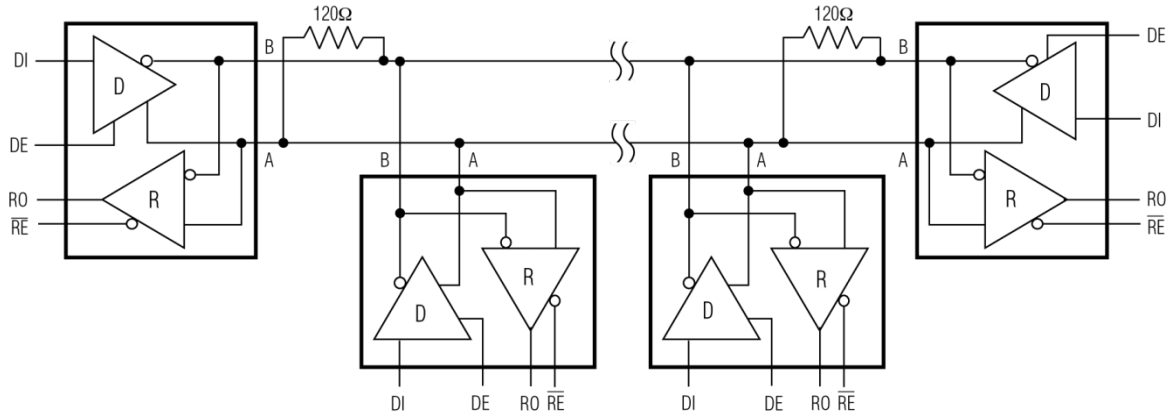
Az RS-422 rendszer kialakításához sodrott érpárú vezeték használata szükséges, amely az állóhullám mentes jelátvitelt biztosítja. A vezeték végét egy adott értékű ellenállással kell lezárni. A rendszerben csak egy meghajtó van a hálózaton, ezért a lezáró ellenállást az utolsó vevőhöz lehető legközelebbi helyre kell elhelyezni a kábelen.



1.11 ábra: Az RS-422 kommunikációra használható meghajtó (MAX489) áramkör bekötése 2 eszköz esetén mindkettő vonalon full-duplex kommunikációval, a buszt lezáró ellenállásokkal

1.5.7. RS 485

A szabvány nem különbözik nagymértékben az előbbiekben ismertetett RS 422-től. A szabvány nem tartalmaz egyértelmű utasításokat a csatlakozó felületeket és a bekötést illetően, azonban szigorú előírásokat közöl a jelek karakterisztikájával kapcsolatosan. Ez a fajta kommunikáció is master-slave alapú. A szabvány során sodrott érpáron keresztül, és szimmetrikusan történik az adatátvitel. A számunkra fontos információt a vezetékek közötti potenciálkülönbség előjele szolgáltatja. A vonalpáron megengedett egynél több adó és vevő csatlakoztatása is. Kialakítható fél-duplex (half-duplex) és tejesen-duplex (full-duplex) összeköttetés, az előbbihez két az utóbbihoz négy vezeték alkalmazása szükséges. A buszra csatlakoztatott eszközök maximális száma 32 lehet, ha ezt meghaladó számút kívánunk a buszhoz rendelni, akkor a vonalerősítők alkalmazása elkerülhetetlen. A szabványt ipari környezetben alkalmazzák és a maximálisan áthidalható távolság 1200m, jelerősítések alkalmazása nélkül.



1.12 ábra: Az RS-485-ös kommunikációra használható meghajtó (MAX481) áramkör bekötése 4 eszköz esetén a buszt lezáró ellenállásokkal

A kommunikáció az úgynevezett multi-drop elven működik, amelyet úgy alakítanak ki, hogy a master adat kimenete csatlakozik az összes slave eszköz adat bemenetére, azonban a slave adat kimenetek villamosan közösítve kapcsolódnak a master adatbemenetére. Ebben a tulajdonságban tér el az előbb ismertetett RS 422 és RS 232 szabványoktól, amelyek pont-pont összeköttetéseket hivatottak szolgálni. A szabvány lehetőséget nyújt busz rendszer kialakítására, mivel azonban egyidejűleg csak egy master adhat jelet, meg kell oldani a vezérlő jel átadását.

Alkalmazható kábelek

A gyakorlatban a fent említett soros kommunikációs hálózatok kialakításához csavart érpár szükséges, amely lehet árnyékolt (STP) és árnyékolatlan (UTP). A vezetékek csavarását az egymásra gyakorolt hatásukból kialakuló interferencia elkerülése indokolja. Az alkalmazott vezetékek általánosságban 4 egymástól eltérő csavarszámú érpárt tartalmaznak, amely az érpárok egymásra hatását akadályozza meg. A csavarások száma azonban befolyásolja a sebességet is, amely a nagyobb szám esetén nagyobb érték. A kábelek ára a méterenkénti csavarások növekedésével arányosan nő. A kábelek három legfontosabb jellemzője a vezeték-átmérő, a kapacitás és a hullámimpedancia, amely lehet 100 illetve 120 Ohm.

1.5.8. MODBUS protokoll

A Modicon cég által kifejlesztett nyílt forráskódú protokoll, amely ipari felhasználásra készült. Egy master/slave típusú adatátvitel, amely előre definiált kódokból épül fel. A felhasználók két üzemmód közül választhatnak, az egyik az ASCII, a másik pedig az RTU, továbbá bármely állapot esetén a Baud-rate és a paritás típusa állítható. Ezeknek a beállításoknak, a hálózatot használó minden eszköz esetében meg kell egyeznie. Az ASCII módot választva, a nyolcbites adatot kettő ASCII kódnak megfelelő karakterként továbbít a protokoll. Ennek a módnak a legnagyobb előnye, hogy két karakter között akár egy másodperc nagyságú szünetek is megjelenhetnek az adatátvitel során anélkül, hogy zavart okoznának. Formátuma a következő:

Kód:	hexadecimális (ASCII karakterek 0-9,A-F)
Egy byte felépítése:	1 start bit, 7 adat bit (LSB először), 1 paritás bit, 1 stop bit ha alkalmazunk paritást és 2 bit ha nem
Hibaellenőrző:	Longitudinális Redundancia Check (LRC)

Az alábbi ábrán látható a küldendő üzenet keretrendszere. Az üzenetek mindig a kettőspont (:=3A hex) karakterrel kezdődnek és a „kocsi vissza-soremelés” (CRLF=0D és 0A hex) karakterrel végződnek.

START	CÍM	UTASÍTÁS	ADAT	LRC	STOP
1 karakter :	2 karakter	2 karakter	n karakter	2 karakter	2 karakter CRLF

A másik említett üzemmód az RTU (Remote Terminal Unit), amely során az adatok kettő darab négybites hexadecimális karakterként kerülnek továbbításra. Előnye a másik üzemmóddal szemben abban rejlik, hogy ugyanazon Baud-rate mellett többlet adatátvitel jelentkezik a nagyobb adatsűrűség következtében.

Kód:	8 bit bináris, hexadecimális 0-9, A-F
Egy byte felépítése:	1 start bit, 8 adat bit (LSB először), 1 paritás bit, 1 stop bit ha alkalmazunk paritást és 2 bit ha nem
Hibaellenőrző:	Ciklikus Redundancia Check (CRC)

Az RTU üzemben az üzenetek 3,5 karakter időnyi szünettel kezdődnek és végződnek, amelyet a Baud rate-nek megfelelő néhány karakteridővel könnyedén megvalósíthatunk (T1-T4). Fontos tudni azonban, hogy az egész adatátvitel folyamatosan kell, hogy történjen, mert ha egy 1,5 karakteridőnél hosszabb megszakítás van a jelsorozatban akkor a fogadó eszköz azt feltételezi, hogy a következő fogadott byte egy új üzenet cím mezőjével egyezik meg Hasonlóképpen, ha egy új üzenet 3,5 karakter időnél előbb kezdődik az eszköz azt feltételezi, hogy az előző üzenet folytatása. Az alábbi táblázatban láthatjuk az RTU üzenet felépítését.

START	CÍM	UTASÍTÁS	ADAT	CRC	STOP
T1-T2-T3-T4	8 bit	8 bit	nx8 bit	16 bit	T1-T2-T3-T4

LRC ellenőrző összeg

Az ASCII módban kerül alkalmazásra. Előállítása úgy történik, hogy a kettőspont ’:’ start és ’CLRF’ stop karakter kivételével az átvitt 8 bites bájtokból képzett kettes komplemens értékek összegződnek.

CRC ellenőrző összeg

Az RTU módban továbbított jelsorozatok ellenőrző összege ez a bájt. A képzése során az üzenet egészét megvizsgálja, és a paritás bit megléte irreleváns az ellenőrzés szempontjából. A CRC kód egy 2 bájtos, 16 bit nagyságú bináris összeg. Az értékét mindig a küldő eszköz képezi, amelyet a fogadó eszköz a vétel során számol, majd összehasonlít a kapott összeggel. Ha a két bináris szám eltérő, akkor hibaüzenet keletkezik. Képzése során a CRC regiszter

minden bitje először logikai 1-es állapotba kerül. Majd ehhez kapcsolódnak az egymást soron követő 8 bites adatok, a start, stop és paritást nem beleértve. A bájtot EX-OR függvénykapcsolatba hozza a regiszter aktuális tartalmával, majd az eredményt az LSB, legkisebb helyiértékű bit irányába eltolja és az MSB, legnagyobb helyiértékű bit helyére pedig 0 logikai értéket ír. Ezután megvizsgálja az LSB értékét, ha 1-es, akkor EX-OR kapcsolatba hozza egy előre definiált értékkel, ha pedig 0 volt, akkor semmilyen műveletet nem hajt végre az összegen. Ez a folyamat addig zajlik, amíg 8 léptetés nem történt. Ezután a következő 8 bit kerül EX-OR kapcsolatba a regiszter aktuális tartalmával, és ismétlődik meg nyolc, az előbb említett léptetés erejéig. A műveletsorozat végén keletkező érték, a CRC hibajavító összeg. Ez az egyik lehetséges megoldás. Létezik egy úgynevezett CRC kódtáblás megoldás is, amely nyilvánvalóan gyorsabb működésű, azonban lényegesen nagyobb memória területet szükségel a működéshez.

1.5.9. PROFIBUS

A Process Field Bus szavak összevonásából alkották meg a A PROFIBUS mozaikszót. A szabványt 1989-ben fejlesztette ki a német kormány másik 15 cég és kutatóintézet segítségével és a PROFIBUS International felügyeli, frissíti a szabványt. A társaság egy németországi székhelyű (Karlsruhe) non-profit szervezet, mely az európai ipari automatizálás piac nagy részét uralja. A cél egy olyan buszrendszer megalkotása volt, amely széles körben alkalmazható a gyártásban és folyamatirányításban használt intelligens terepi eszközök összekapcsolására.

A PROFIBUS univerzális nemzetközi szabványokon alapszik és közelít az OSI modell ISO 7498-as nemzetközi szabványához. Ebben a modellben minden rétegnek (összesen 7) pontosan meghatározott funkciója van. Az első réteg, mely a **fizikai réteget** (Physical Layer) jelenti, meghatározza az átvitel fizikai karakterisztikáját. Fizikai rétegnek a PROFIBUS megalkotói a már meglévő szabványok közül választották ki az RS-485-öt.

A második réteg az **adatkapcsolati réteg** (Data Link Layer), mely a buszelérési protokollt definiálja. A PROFIBUS csak az első és második réteget használja a 7 rétegű OSI modellből.

A fizikai réteg a PROFIBUS szabványban

A fizikai réteg maga a közvetítő közeg, ami összekapcsolja az egy hálózatban szereplő eszközöket. Ide értjük a teljes kábelhosszt, a kialakított topológiát, az interfészt, a csatlakoztatott állomások számát, az átviteli sebességet, amely 9,6 és 1500 kbaud között változhat. Fizikai rétegként az RS-485-öt alkalmazzák a PROFIBUS-nál.

Kábelek

Az RS-485-ös szabványnak megfelelően a fizikai réteg egy lineáris busz, amely mindkét végén ellenállással lezárt csavart érpárú vezeték, maximális hossza 1200 m lehet, a csatlakoztatott állomások száma pedig 32. Lehetőség van kiterjedtebb topológia (hosszabb kábel, csillag struktúra) kialakítására és több állomás csatlakoztatására, amennyiben Repeater állomásokat helyezünk el a hálózatba. Az RS-485-ös szabványnak megfelelően a PROFIBUS-DP

aszinkron, half-duplex adatátvitelt használ és az adatokat NRZ kódolású, 11 bites karakterek formájában továbbítja.

A szabvány „A” és „B” jelzésű jelvezetékeket definiál, ahol „A” az RxD/TxD-N jelnek, a „B” pedig az RxD/TxD-P jelnek felel meg. Logikai „1” forgalmazása esetén tehát az „A” vezetéken alacsony szintnek, míg a „B” vezetéken magas szintnek megfelelő feszültség jelenik meg.

Csatlakozók

Az eszközök egy 9 pólusú **D-Sub** csatlakozón keresztül csatlakoztathatóak, melynek hüvely változata az állomás eszközön, míg a dugó típusú a kábelvégen található. A szabvány fémházas csatlakozók használatát javasolja a zavarvédelem és tartósság érdekében. A csatlakozó lábainak elnevezése az alábbi táblázatban látható.

Láb	Megnevezés	Jelentés
1	SHIELD	Árnyékolás
2	M24V	-24V-os feszültség
3	RxD/TxD-P	Adat pozitív
4	CNTR-P	Control-P
5	DGND	Digitális GND
6	VP	Pozitív feszültség
7	P24V	+24V kimenő feszültség
8	RxD/TxD-N	Adat negatív
9	CNTR-N	Control-N

A PROFIBUS szabvány alábbi három verziója létezik:

- ***PROFIBUS-FMS*** (Fieldbus Message Specification): mely a kommunikációban használatos kliens-szerver modellre épül, automatizálási eszközökre átültetve.
- ***PROFIBUS-PA*** (Process Automation): terepi eszközök és adóállomások összekapcsolását teszi lehetővé egy folyamatirányító eszközzel. Támogatja a biztonságos adatátvitelt és megengedi az energiaátvitelt a kommunikációs vezetéken. A paraméterek és funkcionális blokkok definiálása a folyamatirányítás minden területére kiterjed.
- ***PROFIBUS-DP*** (Decentralized Peripherals): nagy sebességű be- és kimenetek vezérlésére használják, a szenzorok és beavatkozó szervek irányító beredezéshez való kapcsolásához.

A PROFIBUS-DP eszköztípusokat három osztályba lehet sorolni a hálózatban betöltött szerepük szerint:

- **Class 1 DP Master:** Egy class 1 típusú master eszköz általában egy központi programozható irányítóegység (PLC) vagy egy számítógépen futó speciális szoftver. Ez végzi a hálózati címek kiosztását a buszon és az adatcserét meghatározott periodusidő szerint a hozzá csatlakoztatott eszközökkel. Meghatározza a *baud rate*-et (melyet az eszközök automatikusan felismernek), a *token* üzenetek cseréjét vezérli a Master eszkö-

zök között. A Slave-ekkel aktívan kommunikál, míg egy Class 2 Master-rel csak passzívan (kérés alapján). Van saját leírófájlja, mely pontosan leírja működését.

- **Class 2 DP Master:** Ez a típus egy konfigurációs eszköz, mely lehet egy laptop vagy konzolgép. Fő feladata a Class 1 DP Master konfigurálása, de felügyeleti, és diagnosztikai feladatokat is ellát. Aktívan képes kommunikálni Class 1 Master típusú eszközökkel és a hozzájuk kapcsolt Slave-ekkel.
- **DP Slave:** Egy Slave eszköz passzív állomásként van jelen a buszon, amely azt jelenti, hogy csak válaszolni tud a Master kérésekre, valamint megerősítő üzenetet küldeni. Minden ilyen eszköz egy Class 1 DP Master-hez tartozik (amennyiben kapott hálózati címet), amelynek írási joga is van hozzá. Minden neki küldött kérésre válaszol, kivéve a broadcast és multicast típusúakra. A Slave-nek is van leíró fájlja, mely tartalmazza paramétereit és funkcióit.

1.5.10. CAN busz

A CAN használatakor az állomások (vezérlők, érzékelők és beavatkozók) egy buszrendszeren keresztül vannak összekötve, melyeken az információ továbbítása soros szervezésű. A busz egy szimmetrikus vagy aszimmetrikus két vezeték, amely lehet árnyékolt vagy árnyékolatlan is. A fizikai átvitel elektromos paramétereit szintén a CAN szabvány specifikálja.

A mai korszerű járművek többsége nagyszámú elektronikus vezérlő rendszert tartalmaz. A járműiparban az elektronikus vezérlők számának növekedése egyrészt a felhasználó biztonsági és kényelmi igényeinek, másrészt a környezetvédelmi megfontolásoknak (károsanyag kibocsátás és üzemanyag fogyasztás csökkentése) köszönhető. Ilyen vezérlőeszközök lehetnek például a motorban, a sebességváltóban, a kormánynál, valamint a blokkolásgátló (Antilock Braking System – ABS) és menet-stabilizátor (Electronic Stability Program – ESP) rendszerben. A kényelmet szolgáló eszközöknél pedig például a klímánál, és az audiorendszernél.

Ezen rendszerek funkcióinak bonyolultsága elkerülhetetlenné teszi a rendszerek elemei közötti adatcserét. A hagyományos rendszerekben az adatcsere dedikált adatvonalakon keresztül történik, de ezt a vezérlési funkciók bonyolultabbá válásával egyre nehezebb és drágább megvalósítani. A bonyolult vezérlőrendszerekben az összeköttetések száma tovább már nem volt növelhető. Egyre több olyan rendszert is kifejlesztettek a gépjárművek számára, amelyek több vezérlőeszköz együttműködését igényelték. (Motor vezérlése, menet-stabilizátor, automata sebességváltó, műszerfal-gombok.) Szükségessé vált a hagyományos pont-pont összekötésének lecserélése, amit úgy oldottak meg, hogy a rendszer elemeit egy soros buszrendszerre kötötték rá. Az addig használt soros buszrendszereknek viszont túl kicsi volt az átviteli sebességük, vagy a kommunikációs hibákat nem kezelték megfelelően. Mivel az autóipar számára nem volt megfelelő buszrendszer, ezért fejlesztette ki a Bosch a „Controller Area Network”-öt.

A CAN protokoll, amely az ISO OSI (Open Systems Interconnection) modell fizikai és adatkapcsolati rétegének felel meg és kielégíti a járműipari alkalmazások valósidejű igényeit is. Az egyszerű konfigurálhatóság, olcsó implementálhatóság, nagy sebesség és a központi hibaellenőrzés a CAN előnyös tulajdonsága, amely a CAN gyors elterjedését eredményezte.

A CAN rendszerek járművekben való használatának elsődleges célja az volt, hogy az vezérlő egységek központi vezérlő használata nélkül is tudjanak kommunikálni.

A CAN busz története

1990-ben a CAN specifikáció megalkotója, a Bosch GmbH., a CAN specifikációját nemzetközi szabványosításra nyújtotta be, majd ez után 1992-ben megalakult a CAN in Automation (CiA) független nemzetközi felhasználói és gyártói csoport, a különböző megoldások egységesítéséhez, valamint a CAN további technikai fejlődésének elősegítéséhez. A CiA leszűkítette az ISO OSI modell szerinti fizikai réteg specifikációját vezeték, csatlakozó és transceiver ajánlásra. Később a CiA kidolgozta a CAL-t (CAN Application Layer), amely az ISO OSI modellhez képest a CAN-ből addig hiányzó alkalmazási réteget képes pótolni. Később olyan további CAN alkalmazási rétegek definiálásával foglalkoztak, mint például a SDS (Smart Distributed System).

1993-ban a Nemzetközi Szabványügyi Hivatal (International Standardisation Organisation – ISO) kiadta az ISO 11898-as CAN szabványt, amely a protokoll standard formátumú (11 bites azonosítójú) üzenetein túl a fizikai réteget is definiálta, a maximális 1 Mbit/s-os átviteli sebességig.

Az egy rendszerben elküldhető üzenetek növekedésével szükségessé vált a kiterjesztett formátumú (29 bites azonosítójú) üzenetek specifikálása, amelyet az ISO 11898 kiegészítéseként jegyzett be a Nemzetközi Szabványügyi Hivatal 1995-ben. A CAN 2.0-ás specifikáció az alábbi fejezetekből és függelékből áll:

- CAN 2.0 „A fejezet” (Part A): Standard formátumú üzeneteket (CAN Specification 1.2 alapján)
- CAN 2.0 „B fejezet” (Part B): Standard és a kiterjesztett formátumú üzenetek
- CAN 2.0 Függelék: Útmutatást ad arra, hogyan kell megvalósítani a CAN protokollt a szabványnak megfelelően

A CAN specifikációkat leíró szabványok a következők:

- ISO 11898-1: a CAN adatkapcsolati rétegét írja le,
- ISO 11898-2: a CAN nagysebességű fizikai réteget definiálja,
- ISO 11898-3: a CAN alacsony sebességű, hibatűrő fizikai rétegét rögzíti.

A CAN busz alacsony költséggel implementálható, a CAN vezérlők ma már a közepes tudású mikrokontrollerekbe is bele vannak integrálva, a halózat kialakításához használható sodort érpárú vezeték kis költséggel beszerezhető. Szinte az összes mikrokontroller gyártó eszközválasztékában megtalálhatók a CAN kommunikációt támogató chipek, amely a CAN vezérlők árának a csökkenését eredményezte. A CAN alacsony kiépítési költsége nagyban hozzájárult a CAN busz gyors elterjedéséhez az autóiparban és egyéb területeken is.

A korszerű személygépkocsikba és teherautókba egyre nagyobb mennyiségű elektronikát építenek be, és az eszközök száma és bonyolultsága folyamatosan növekszik. Már kaphatók olyan gépjárművek, melyekben a kormánykerék áttétele a sebesség függvényében változik, vagy amelyben a kézifék elektronikusan működik. Már a középkategóriás a szériaautókban is megjelent olyan parkolássegítő rendszer, melynek segítségével az autó saját maga be tud parkolni egy adott parkolóhelyre.

A mai gépjárművekben szinte az összes elektronikus eszköz a CAN buszra van kötve. (Újabb gépjárművek esetében CAN buszra es FlexRay-re.) A közepes-nagyobb

funkcionalítással rendelkező autótípusok esetén a CAN busz terheltsége akkora, hogy a folyamatos információáramláshoz 2-3 CAN halózat szükséges. Ilyenek például azok a gépjárművek, amelyek ESP-vel (elektronikus menetstabilizátorral) vannak felszerelve. A modern gépjárműveknél a busz(ok) sebessége általában 1 Mbps. Azok az eszközök, amelyek kevesebb és a rendszer működése szempontjából nem annyira fontos adatokat küldenek, ritkábban küldik el az adatokat (például: klíma, rádió), azok az eszközök, amelyek fontos funkcionalitást látnak el sűrűbben küldik (például: motor, váltó, ABS vagy ESP vezérlő, gyorsulásmérő szenzorok, stb.).

Mivel a menetstabilizátor vagy a kipörgésgátló vezérlő bemenetére vannak a kerékszenzorok csatlakoztatva, így a vezérlő képes megállapítani a kerekek sebességét. Ezekből az adatokból tudja kiszámolni, hogy az autó valamelyik kereke megcsúszott vagy éppen kipörög. Ha megcsúszik, akkor szelep(ek) segítségével a megfelelő keréken csökkenti a fékkör olajnyomását, ha pedig kipörög a kerék, akkor a differenciálmű és a motor észleli a vezérlő által küldött adatokból, hogy szükséges valamilyen beavatkozást végezni.

Az ipari terepibusz rendszerek és a járművek buszrendszereinek az összehasonlítása sok hasonlóságot mutat. Fontos követelmény az alacsony költség, az elektromágneses zajjal terhelt környezetben való működés, a valósidejű működés és az egyszerű használat. Jól használható a gépekben vagy gyárakban az "intelligens" I/O eszközök és az érzékelők/beavatkozók hálózatba kapcsolására. Napjainkban az összes nagy PLC gyártó kínálatából kiválaszthatóak olyan típusok, amelyek a CAN buszra csatlakoztathatók. Az adatátvitel megbízhatóságán túl a csomópontokra eső alacsony költség is jelentős érv a CAN használata mellett.

A beágyazott rendszerekbe kerülő új eszközök kifejlesztése és javítása egyaránt bonyolult feladat, mivel a berendezések egymással logikai kapcsolatban vannak, és felhasználják egymás adatait. Ahhoz, hogy egy új terméket ki lehessen fejleszteni, egy olyan tesztrendszert szükséges kiépíteni, amely képes arra, hogy a kifejlesztendő termék számára a bemeneti adatokat biztosítsa, és képes az érkező adatok feldolgozására és ellenőrzésére. Mivel a CAN buszra csatlakozó egységek az adataikat CAN buszon fogadják és küldik, ezért a hatékony fejlesztőmunkához szükséges(ek) olyan eszköz(ök), amellyel a busz forgalmát monitorozni lehet.

A CAN általános jellemzői

A CAN protokoll egy *multi-master* protokoll, amelyben a csomópontok (*node*) üzenetek (*message frame*) segítségével kommunikálnak egymással. Mindegyik adat továbbítást végző üzenetkeretnek tartalmaznia kell az üzenet azonosítóját (*message identifier*) és adatküldés esetében az adatmezőt (*data field*).

Minden CAN *node* egyenrangú, nincsen kiválasztott busz-vezérlő (*bus master*). Minden *node* képes az üzeneteit önállóan továbbítani az adatbuszon (*data bus*). Egy *node* leállása esetén sem válik működésképtelenné a CAN busszal épített beágyazott rendszer, de a rendszer funkcionalítása csökkenhet.

Az üzenetek azonosítása egyedi azonosító (*identifier*) alapján történik, ez alapján lehet az üzeneteket priorítás alapján csoportosítani. Az üzenetazonosító határozza meg tehát az adott üzenet prioritását, valamint közvetlenül szerepet játszik a buszért való versengés eldöntésében is. A fontosabb információt hordozó üzeneteknek nagyobb a priorítása.

A buszon továbbított üzeneteket mindig minden csomópont megkapja (*broadcast*), és ellenőrzi. A csomópontok CAN vezérlői az üzenetek azonosítója alapján döntik el, hogy az üzenetet eltárolják-e a vezérlő pufferébe (*message filtering*). A filter konfigurálása mindig a beágyazott rendszer mikrokontrollerének/CPU-jának a feladata, a filter működés közben bármikor megváltoztatható.

A CAN busz prioritásos CSMA/CD+CR (Carrier Sense, Multiple Access/Collision Detection + Collision Resolution – Vivőjel érzékeléses többszörös hozzáférés ütközésérzékeléssel) médiaelérési technikát használja. Az adatot küldeni kívánó csomópontok várnak a busz felszabadulásáig, majd a üzenet kezdete bit (*start bit*) átvitelével megkezdik az adatküldést. A *start bit* szinkronizálja az összes csomopontot. Az üzenet-azonosító továbbítása a *start bit* küldése után kezdődik meg. Több *node* buszért való versengésekor ebben a szakaszban történik az ütközés feloldása, bitszintű arbitrációval. Ez a technika a nem-destruktív arbitrációs mechanizmus (*non-destructive arbitration*), mivel a magasabb prioritású üzenet nem sérül, így mindig a legmagasabb prioritású üzenet lesz továbbítva a buszon késleltetés nélkül.

A rendszertervezők és üzemeltetők számára fontos, hogy tudják azt, hogy az adott *node* még csatlakozik a buszhoz, azért az üzenetek globális nyugtázó mezővel rendelkeznek, amely jelzi a küldő *node*-nak, hogy legalább egy *node* hibátlanul vette az üzenet. Minden *node* nyugtázó jellel válaszol, ha nem észlelt semilyen hibát. Minden egyes elküldött üzenetre garantálja a szabvány, hogy azt minden *node* elfogadja vagy elutasítja (konzisztens adatátvitel). Ha bármely vevő hibát észlel a vétel során, akkor egy hibajelző üzenettel (*error frame*) azonnal megszakítja az átvitelt.

A bitszint domináns vagy recesszív lehet, a bitfolyam kódolása a *Non-Return-to-Zero* (NRZ) elv szerint valósítja meg a CAN. A *node*-ok szinkronizálásához a bitbeszúrás módszerét is alkalmazza a CAN. Öt egymást követő azonos értékű bit után egy ellentétes bitet illeszt be a küldő node a *frame*-ek küldése során, ezáltal a küldő *node* beszúrt egy le- vagy felfutó élt a bitidő szinkronizálásához. A CAN vezérlők sokféle hiba detektálásra képesek: vezeték szakadás, lezáró ellenállások hibája, testzárlat, egyéb zárlatok. A szabvány nem definiálja, hogy mi a teendő a fenti hibák esetén.

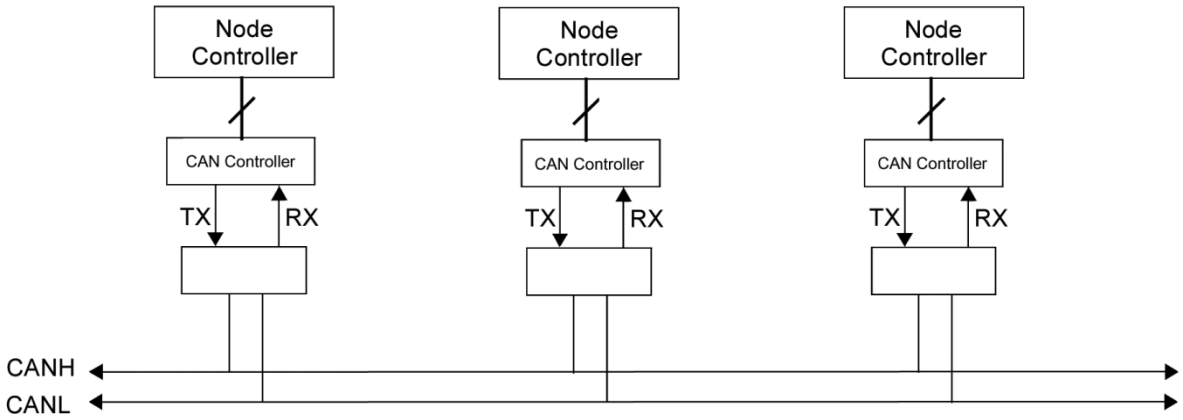
A kommunikáció adott esemény bekövetkezésének (új információ generálódott egy csomópontban) hatására is el kezdődhet. Az új információval rendelkező csomópont maga kezdi meg az átvitelt. Így jelentős kommunikációs időt takarít meg azokhoz a rendszerekhez képest, amelyekben a csomópontok minden ciklusban adott időszelettel rendelkeznek, amelyben az új információjukat elküldhetik. Ugyanis abban az esetben, ha nincs új információja egy csomópontnak, akkor ez az időszelet kárba vész, míg esetlegesen egy másik, új információval rendelkező eszköznek várnia kell, amíg sorra kerül. Ha sok *node* van a rendszerben és azok sokféle információval rendelkeznek, akkor a bus eseményvezérelt kezelése nem hatékony, mert esetlegesen 1 bit új információ elküldéséhez egy teljes üzenetet kell elküldeni, amely legalább 55 bit továbbításának az idejét veszi igénybe a standard CAN üzenet alkalmazása esetében.

Sok csomópontot tartalmazó időkritikus valós idejű rendszerek esetében csak a ciklikus információcsere alkalmazása elfogadható megoldás, ekkor egy belső időzítő *(timer)* segítségével időzítik a CAN üzenetek küldését, így detektálható az is, ha egy *node* meghibásodik és már nem képes kommunikálni.

Az eseményvezérelt kommunikációt kiegészítve a CAN lehetőséget biztosít „adatkérő üzenet" küldésére. Ezek segítségével egy *node* lekérdezheti a számára fontos információkat egy másik csomóponttól. Az adatkérő üzenet, és az arra adott válasz is külön üzenetet alkot. Összetettebb rendszerek esetében általában a *node*-ok állapotának (aktív/inaktív) lekérdezésére használják. Ciklikus komminikáció esetében az „adatkérő" üzenetet nem használják.

A szabvány különböző módszereket biztosít a CAN busz meghajtására, amelyek a következők:

- **Differenciális mód** használata esetében kettő jelvezeték és egy földvezeték (illetve referencia vezeték) szükséges. A logikai bitszintet a két vezetéken lévő jelek különbségéből határozza meg. Elektromos zavarok ellen védett.
- **Kiegyensúlyozatlan mód** használata esetén egy föld- és egy jelvezeték. Nagyon érzékeny a zajokra, csak erősen költségérzékeny alkalmazásokban alkalmazzák alacsony sebességen, vagy a már említett vezetékhibák ideiglenes áthidalására.

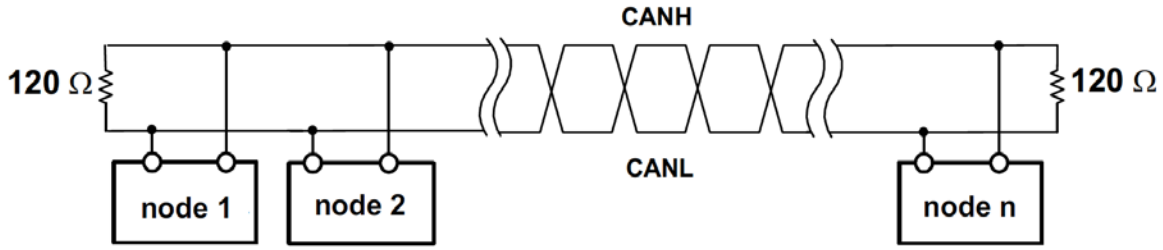


1.13. ábra: CAN-es csomópontok hardverének elvi felépítése

A CAN busz rendszere rugalmasan alakítható, mert a *node*-okat dinamikusan rákapcsolhatjuk, illetve leválaszthatjuk a buszról anélkül, hogy a többi *node* kommunikációját ez befolyásolná. Az összetett nagy bonyilultságú rendszerek tervezésében nagy szabadsági fokot nyújtanak a következő tulajdonságok:

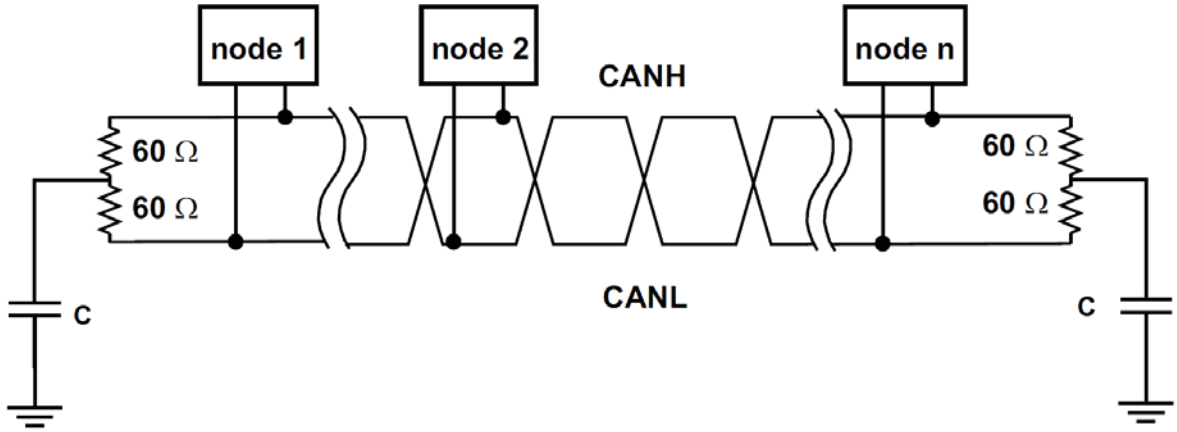
- A *node*-ok száma szabványos buszmeghajtók alkalmazása esetén egy rendszeren belül 32 lehet, speciális (nagyobb áramú) buszmeghajtók esetén akár 64-128 darab *node*-ra is bővíthető.
- Üzenetek száma a rendszerben standard üzenetformátum esetén 2048 (2^{11}), kiterjesztett üzenetformátum esetén pedig 536 870 912 (2^{29}) lehet.
- Az elküldött adatmennyiség üzenetenként 0 és 8 byte között változhat, amely megfelelő megkötésekkel elegendő a járművekben valamint beágyazott illetve automatizált gyártó rendszerekben történő alkalmazásokhoz, és garantálja a lehető legrövidebb buszelérési időt a nagy prioritású üzenetek számára.
- A maximális üzenethossz a beszúrt bitekkel együtt standard üzenetformátum esetén 130 bit, kiterjesztett üzenetformátum esetén pedig 154 bit lehet. 1Mbit/s-os buszsebesség mellett 130 μs és 154 μs a maximális adatküldési idő.
- Változó busz hosszúság.

Mivel a CAN áramhúrkot használ, ezért elektromágneses interferenciákra alacsony az érzékenysége. A CAN szabvány garantálja, hogy a küldő *node* által elküldött adatok megegyeznek a fogadó *node*-ok által fogadott adatokkal. A hibadetektáló mechanizmust figyelembe véve 90% feletti buszterhelés esetében statisztikailag 1000 év alatt egy olyan hiba fordulhat elő, amelyet a rendszer nem detektál.



1.14. ábra: CAN busz felépítése és lezárása standard mód esetén

Az busz felépítését mutató ábrán jól látszanak az ellenállások, amelyek villamosan összekötik a CAN High (CANH) és CAN Low (CANL) vezetéket. Az ellenállások a segítségével alakul ki az áramhurok.



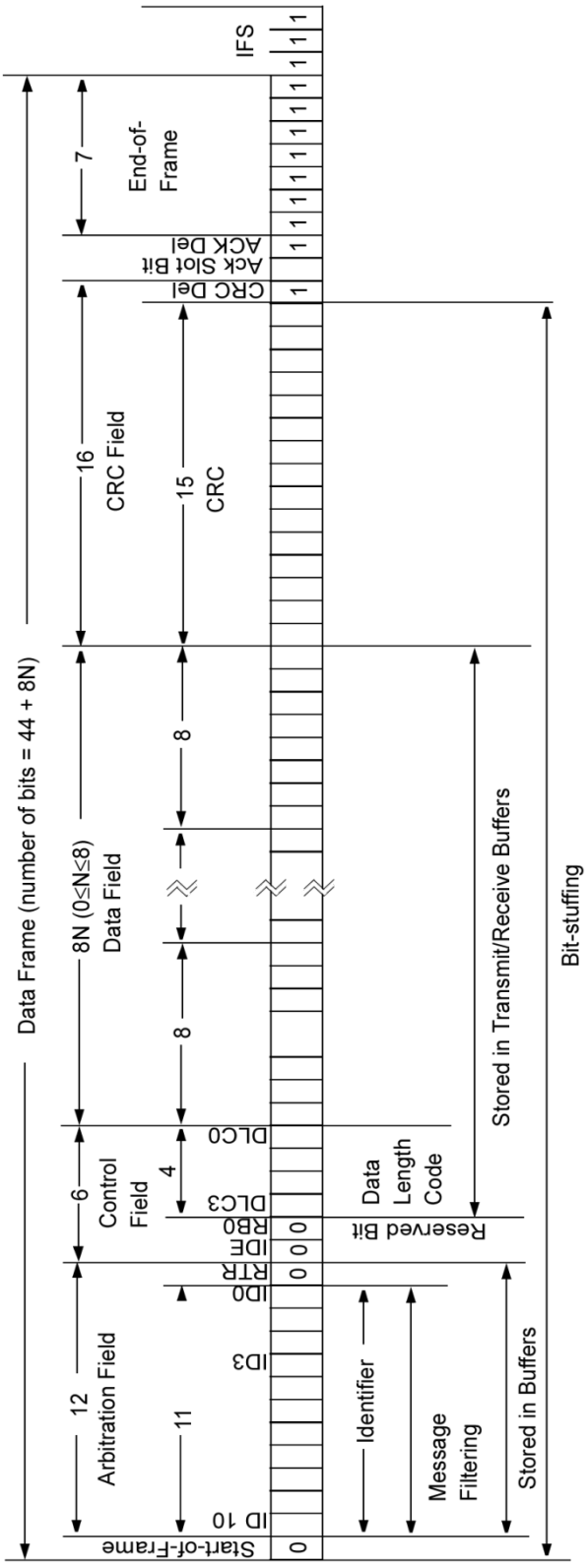
1.15. ábra: CAN busz felépítése és lezárása split mód esetén

A CAN Split (osztott) módon történő lezárása sokkal zavarvédettebb kommunikációt eredményez, elektromágneses zaj által szennyezet környezetben a Split lezárás alkalmazása ajánlott.

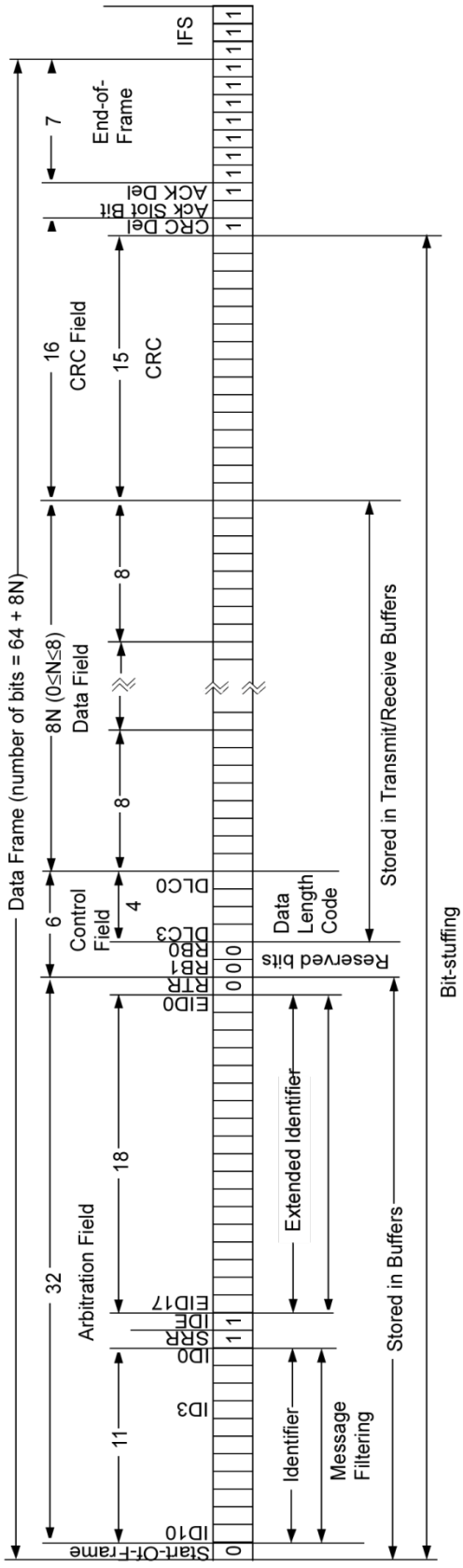
A CAN busz a következő hibadetektáló és hibakezelő mechanizmusokkal rendelkezik:

- 15 bites, 6-os Hamming-távolságú CRC-vel *(Cyclic Redundancy Check)*, amely 5 hibás bit felismerését teszi lehetővé
- a hibás üzenetet a küldő *node* automatikus újraküldi
- az ismétlődő hiba esetén a *node* lekapcsolása a buszról

Az adatátviteli sebesség a CAN busz hossztól függően 5 kbit/s és 1 Mbit/s között változhat. Az elméletileg elérhető legnagyobb buszsebesség 1 Mbit/s, amely maximum 25 m hosszú busszal garantál a rendszertervezők számára a szabvány. Ha ennél hosszabb busz alkalmazása szükséges, akkor csökkenteni kell a bitsebességet. 400 méteres buszhossz esetén a bitsebesség 100 Kbit/s-re csökken, 1000m hosszú busz alkalmazása esetén pedig csak 50 kbps bitsebesség implementálható, 1 km-nél hosszabb busz alakalmazása esetén jelismétlő(ke)t kell alkalmazni.



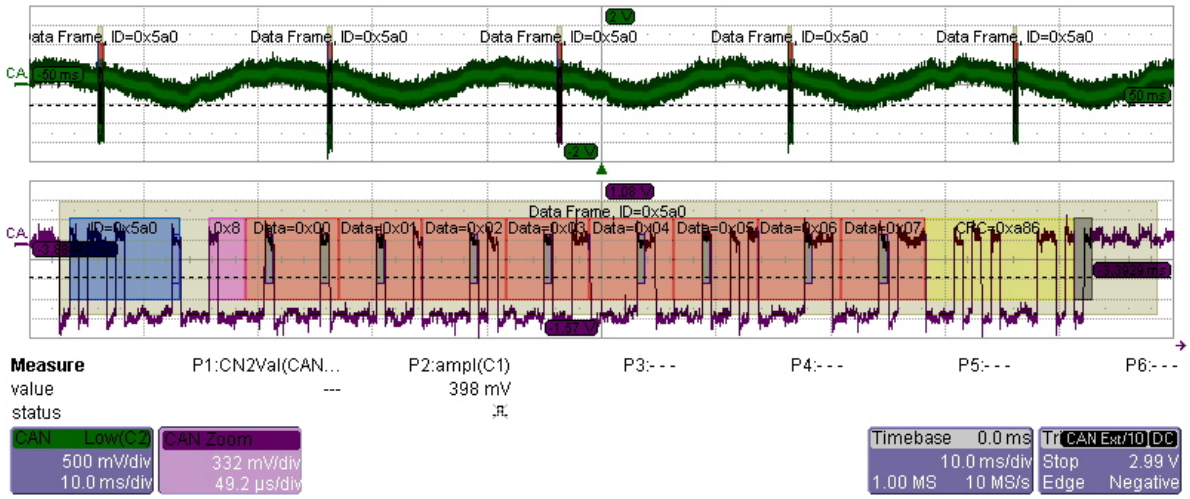
1.16. ábra: Standard adat frame felépítése



1.17. ábra: Extended adat frame felépítése

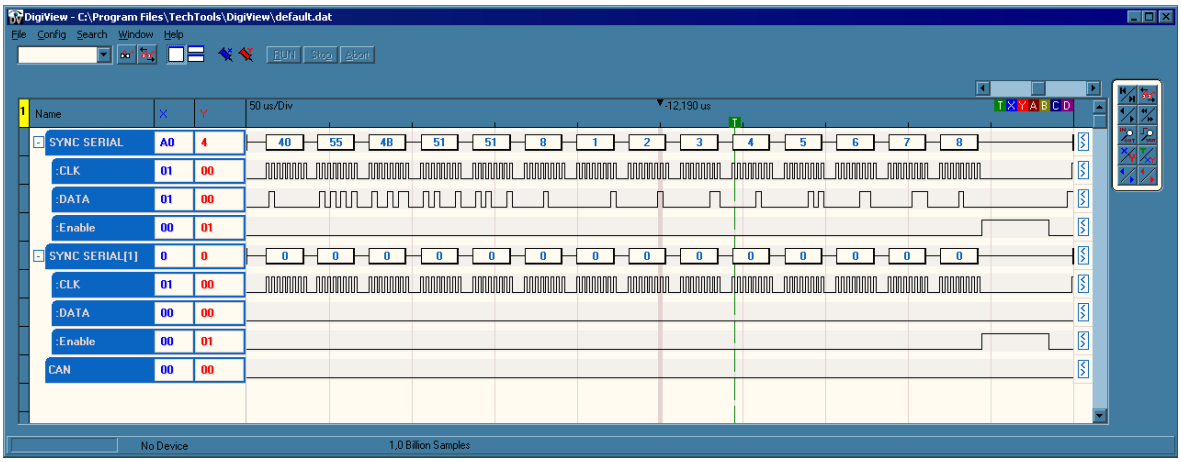
A frame-ek küldése látható a következő ábrán. A mérések LeCroy waveRunner 6050A Oscilloscope típusú eszközzel készültek. A frame-eket generáló PIC mikrokontrolleres program 20 ms-onként küldi a frame-eket a CAN buszra. A frame adatai a következők voltak:

- Azonosító: 0x5A0
- A frame adatbyte-jainak hossza: 8 byte
- Adatbyt-ok: 0,1,2,3,4,5,6 és 7



1.18. ábra: Standard adat frame (LeCroy-os mérés)

A következő ábrán az SPI buszos adatforgalom látható, amely feltölti a CAN vezérlő IC kimeneti buffer-ét a megfelelő tartalommal (a 8 elküldött adatbyte jól látható az ábrán, a byte-ok értékei: 1, 2, 3, 4, 5, 6, 7, 8):



1.19. ábra: Az SPI buszon elküldött adatbyte-ok figyelhetőek meg a logikai jelanalizátor program ablakában, miközben beállítja a Microchip MCP2515 IC regisztereit

Összefoglalva a CAN protokoll legfontosabb tulajdonságai a következők:

- Nem-destruktív arbitráció
- Multimaster
- Üzenetközpontú

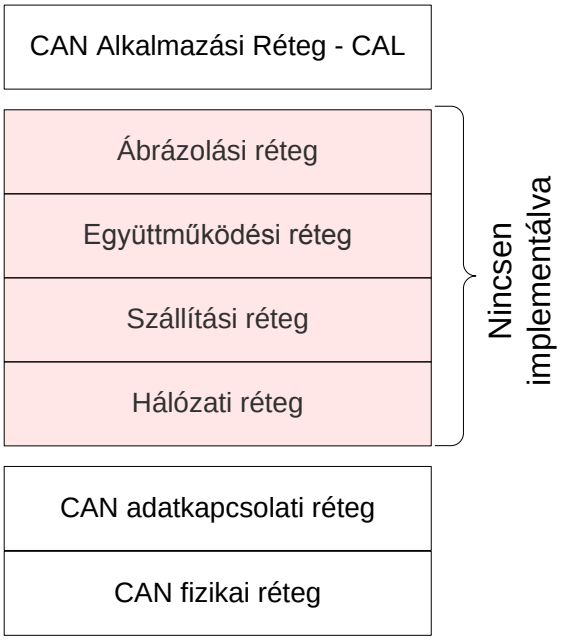
- Broadcast
- Adatkérés
- Eseményvezérelt
- Költséghatékony
- Gyors 1000Kbit/-os maximális sebesség (40m-es buszhossznál)
- Flexibilis
- Robusztus
- Nyugtázás
- Hiba detektálás
- Konzisztens üzenetátvitel
- Bitkódolás
- Bit szinkronizáció
- Nagy eszközválaszték

1.5.11. CANopen

A CANopen általános jellemzése

A CAN szabvány specifikációja az ISO OSI modell szerint a fizikai szintet és az adat bitek fizikai továbbítását és azok keretezését definiálja. Sok esetben az alkalmazások készítésekor a fejlesztési munkát megkönnyítené, ha lenne egy magasabb szintű szolgáltatásokat nyújtó szabványosított réteg.

A CiA a CAN alkalmazási réteget (CAL) definiálta, ez a réteg tulajdonképpen a kommunikációt végrehajtó rendszer és az alkalmazás közötti interfész.



1.20. ábra: Kibővített CAN modell

Sok új szolgáltatást nyújt a CAN alkalmazási rétege (CAN Application Layer – CAL), saját üzenetobjektumokat és üzenetformátumokat definiál, melyek segítségével megadhatóak az eszközök nyújtotta funkciók elérési módjai. Az üzenetekbe kerülő adatokra új adattípusokat és újfajta kódolást definiál.

A rendszertervezők munkáját a master-slave kommunikációt támogató eszközök is támogatják, a legfontosabb ilyen eszköz a hálózatirányító rendszer (Network Management – NMT), rétegirányító rendszer (Layer Management – LMT) és a dinamikus azonosító kiosztást végző (Distributor – DBT).

Az ISO OSI modell szerint a CAL-lal kibővített CAN modell már tartalmazza az ISO OSI modell legfelső szintjét, de a modell közbülső rétegei így is hiányoznak. A megbízható működéshez a hiányzó rétegek implementálása nem feltétlenül szükséges.

Az így létrejött CAN alkalmazási réteg egy jól használható kiindulási pontot nyújt a fejlesztésekhez. Sok új általánosan felhasználható szolgáltatást definiál, de a szolgáltatások felhasználási módját a rendszertervezőre bízza. A CAN alkalmazási réteg lényegében ugyanazt a problémakört hozta létre, mint ami a CAL kifejlesztését okozta. A CAL-lal kibővített rendszer még mindig túlságosan általános maradt, és még mindig nem volt eléggé könnyedén használható.

A CANopen specifikáicó az eszközök információinak a rendszerét határozza meg az elérés és az adattárolás szempontjából. Objektumokat, könyvtárakat és alkönyvtárakat definiál, amiket azok azonosító számával érhetünk el. A szabvány három fő kommunikációs objektumtípust határoz meg:

- Adminisztratív üzenetek, például a hálózat kezelését segítő üzenetek (NMT).
- Szerviz adat objektumok (SDO – Service Data Objects), melyeknek feladata az adatok írása és olvasása.
- Process adat objektumok (PDO – Process Data Objects), amelyek real-time adadtok cseréjére szolgál.

Az üzeneteket framekbe kell foglalni, amelynek egy részét a CAN vezérlő és driverei oldják meg, a felhasználónak csak a CANopen által elvárt protokolláris adatokat kell feltölteni a megfelelő tartalommal. Minden üzenet egy azonosítóval kezdődik, melynek a neve: COB-ID (Communication Object ID – Kommunikációs Objektum Azonosító) kezdődik. A COB-ID két fő részből áll és egy 1 bit hosszúságú mezőből:

- 4 bit a kívánt funkció kódja (NMT, PDO, SDO írás/olvasás, stb),
- 7 bit a *node* sorszáma
- 1 bit-es RTF (Remote Frame) mező

1.5.12. LIN

A Local Interconnect Network (LIN) sokkal több, mint "egy protokoll". A LIN alkalmazása egy egyszerű tervezési módszertant határoz meg, mivel a LIN definiál egy tools-interface-t és egy jel alapú API-t (Application Program Interface-t). A LIN egy nyílt kommunikációs szabvány, amely lehetővé teszi olcsó multiplex rendszerek gyors és költséghatékony megvalósítását. A LIN támogatja beágyazott rendszerek modell-alapú tervezését és validálását. A LIN protokoll alkalmazása esetén a fejlesztés elején a fejlesztők a project-specifikus fejlesztési

folyamatokra összpontosíthatnak, így gyorsabb és költséghatékonyabb a fejlesztés, mint a hagyományos fejlesztési módszerek alkalmazása esetében.

A LIN szabvány nemcsak a busz-protokoll meghatározása, de kiterjeszti annak hatályát az eszköz-interface, újrakonfigurálási mechanizmusok, és a diagnosztikai szolgáltatások segítségével. Ezáltal egy holisztikus kommunikációs megoldást nyújt az ipari, az autóipari, és a fogyasztói alkalmazások hatékony alkalmazásában, így a rendszermérnökök számára egy ideális megoldás a LIN választása.

A rendelkezésre álló dedikált eszközök automatizálják a tervezési és rendszerintegrációs folyamatok kulcsfontosságú tényezőit, amelyek által a LIN sikeres protokollá vált.

A LIN konzorciumot 1998 végén öt autógyártó: az Audi, a BMW, a Daimler Chrysler, a Volvo és a Volkswagen, az autóipari beszállítók közül a Volcano Communications Technology és a félvezetőgyártó Motorola hozta létre. A munkacsoport összpontosított arra, hogy a LIN specifikációja nyílt szabványa olcsó megoldást nyújtson a járműveknél, ahol a sávszélesség és sokoldalúság miatt a drágábban kiépíthető CAN busz nem szükséges.

A LIN szabvány tartalmazza az átviteli protokoll specifikációt, az átviteli közeget, az interface-t a fejlesztési eszközökhöz, valamint a kapcsolódási pontok szoftveres programozási lehetőségeit. A LIN támogatja a skálázható architektúra kialakítását és a hardveres és szoftveres átjárhatóságot a hálózati csomópontok szemszögéből. A LIN kialakításakor fontos szempont volt a kiszámítható elektromágneses kompatibilítás (Electromagnetic Compatibility – EMC).

A LIN kiegészíti a jelenlegi, autóiparban használt multiplex hálózatok tulajdonságait. Ez az a tényező, amely lehetővé teszi az autóiparban eddig használt hierarchikus járműhálózatok mellett a LIN alkalmazását, amellyel további minőségi javítás és költségek csökkenése érhető el a járműveknél. Ezáltal csökkenthető az egyre összetettebb elosztott és beágyazott rendszerek fejlesztési és karbantartási költsége is.

A LIN busz legfontosabb tulajdonságai:

- Egy mester egység van a buszon
- Több slave egység lehet a buszon
- Olcsó univerzális aszinkron adó vevő (UART – Universal Asynchronous Receiver Transmitter) vagy soros kommunikációs interfész (SCI – Serial Communication Interface)
- Önszinkronizálás a slave csomópontokba épített kvarc vagy kerámia rezonátor nélkül
- Determinisztikus jelátviteli szervezet, a jel terjedési ideje előre kiszámítható
- Jelzések alapú API (Application Program Interface)

A LIN hálózat egy mester és egy vagy több slave csomópontból áll. Az átvitelhez használt médiához való hozzáférést a master egység szabályozza, így nem szükséges arbitráció (arbitration) és ütközés detektálás (collision management). Legrosszabb esetben is garantált az adatok átvitele, csak késleltetett jelátvitel következik be.

A mai LIN 2.1 szabvány egy kiforrott specifikáció, amely több régebbi LIN változatot foglal magában (LIN 1.2 , LIN 1.3, és LIN 2.0). A LIN 2.0 egy jelentős technológiai lépés volt azáltal, hogy új slave szolgáltatásokat vezetett be, mint például a konfigurációs slave

csomópontokat. A LIN 2.1 szabványt a LIN 2.0 használata során szerzett tapasztalatok alapján alakították ki a jelentős autóipari felhasználók bevonásával.

A LIN 2.1-es szabvány a LIN 2.0-ban bevezetett diagnosztikai funkciók bővítésével vált teljessé, amelyek a konfigurációs folyamatot teszik még hatékonyabbá. A mai LIN-es csomópontok a LIN 1.2, LIN 1.3, és LIN 2.0 és a LIN 2.1-es szabványon alapulnak, amely azt eredményezi, hogy a csomópontok lefelé kompatibilisek egymással.

1.5.13. FlexRay protokoll

A klasszikus kommunikációs szabványok esetében rendkívül kötött a hardver kialakítása, a FlexRay esetében nincsenek ilyen jellegű kötöttségek. Sok különböző módja van, hogy kialakítsunk egy FlexRay cluster[4]-t. Lehet egy vagy két csatornás busz, csillag elrendezésű vagy akár ezek mindegyikét tartalmazó hibrid rendszer.

A FlexRay protokollt már évek óta alkalmazzák az autóiparban a magasabb árkategóriás gépjárművekben, például BMW X5. A FlexRay-t is tartalmazó autókben megtalálható a CAN is. A FlexRay-t leginkább a vezérlők közötti multimédiás információcserére alkalmazzák, mert az CAN-nel rendkívül körülményesen lenne csak megvalósítható a 8 byte-ban korlátozott elküldhető adat méret miatt.

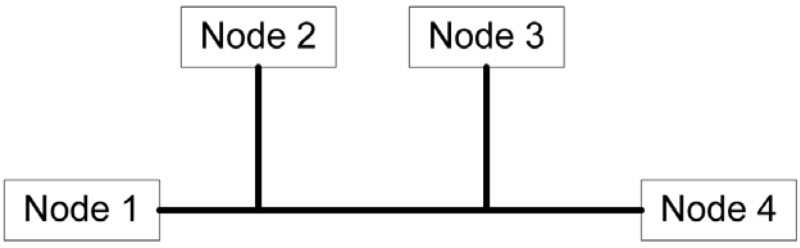
Pont-Pont közötti kapcsolat

Ha két *node* csatlakozik a rendszerünkhöz, akkor pont-pont közti összeköttetést kell alkalmazni, a két *node* közti maximális távolság 24 m lehet.

1.21. ábra: Pont-Pont közti kapcsolat

Passzív busz topológia

A csillag és aktív elemektől mentes struktúrákat hívjuk passzív busznak (*Passive Bus*). Legalább négy *node*-ra van szükség ehhez a topológiához, és a csatlakozók számának minimum kettőnek kell lennie. Két tetszőleges *node* közötti maximális távolság 24 m lehet.

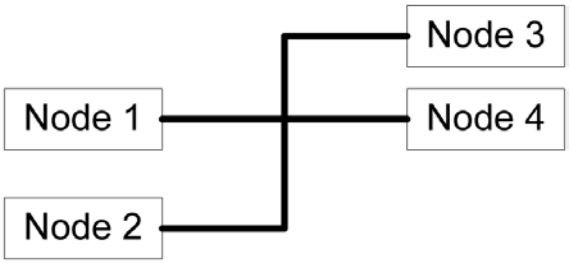


1.22. ábra: Passzív busz

[4] Node-okból álló bus és/vagy csillag topológiájú összetett kommunikációs rendszer.

Passzív csillag topológia

A passzív busz (*Passive Star*) egy speciális esete. Minden *node* egy csatlakozási ponthoz kapcsolódik. A hálózathoz legalább három *node*-nak kell csatlakoznia ennél a kialakításnál és maximum 22 *node*-ot tartalmazhat a passzív csillag. Két tetszőleges *node* közötti maximális távolság 24 m lehet.



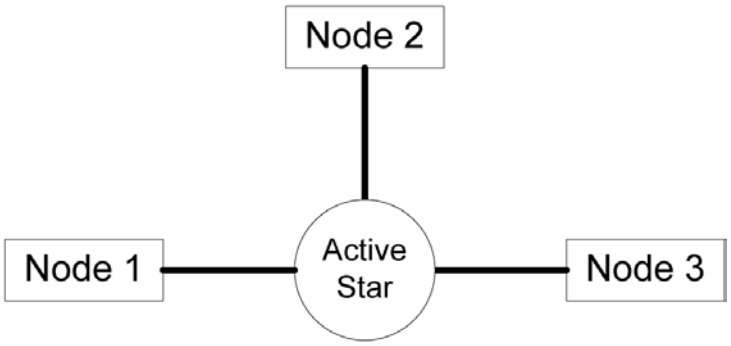
1.23. ábra: Passzív csillag

Aktív csillag topológia

Az aktív csillag (*Active Star*) pont-pont közötti kapcsolatot használ a *node*-ok és az aktív csillag között. A csillaghoz kapcsolódó ágak minimális száma kettő, és egy ág maximális hossza maximum 24 m lehet.

Az aktív csillag feladata, hogy a hozzá csatlakozó *node* információját a többi ágra továbbítsa. A csillag minden ágához tartozik egy küldő és egy fogadó áramkör, így az ágak egymástól elektronikusan függetlenek.

Ha az aktív csillagnak csak két ága van, akkor azt degenerált csillagnak vagy hub-nak hívunk, és a teljes busz hossz megnövelésére használhatjuk. Azért is érdemes használni, mert növeli a hiba behatárolhatóságát két passzív busz között.



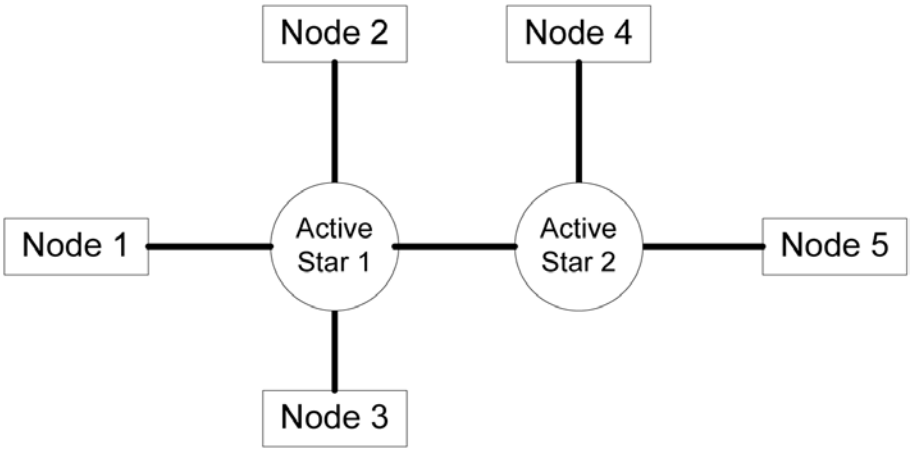
1.24. ábra: Aktív csillag

1.25. ábra: Hub felépítése

Kaszkád aktív csillag

Két aktív csillagot kaszkád csillagnak nevezünk, ha pont-pont alapú kapcsolatban vannak egymással. Ezt a kapcsolatot kiterjeszthetjük passzív csillaggá/busszá, hogy a későbbiekben elérhetővé váljon node-ok, vagy újabb aktív csillagok fogadására.

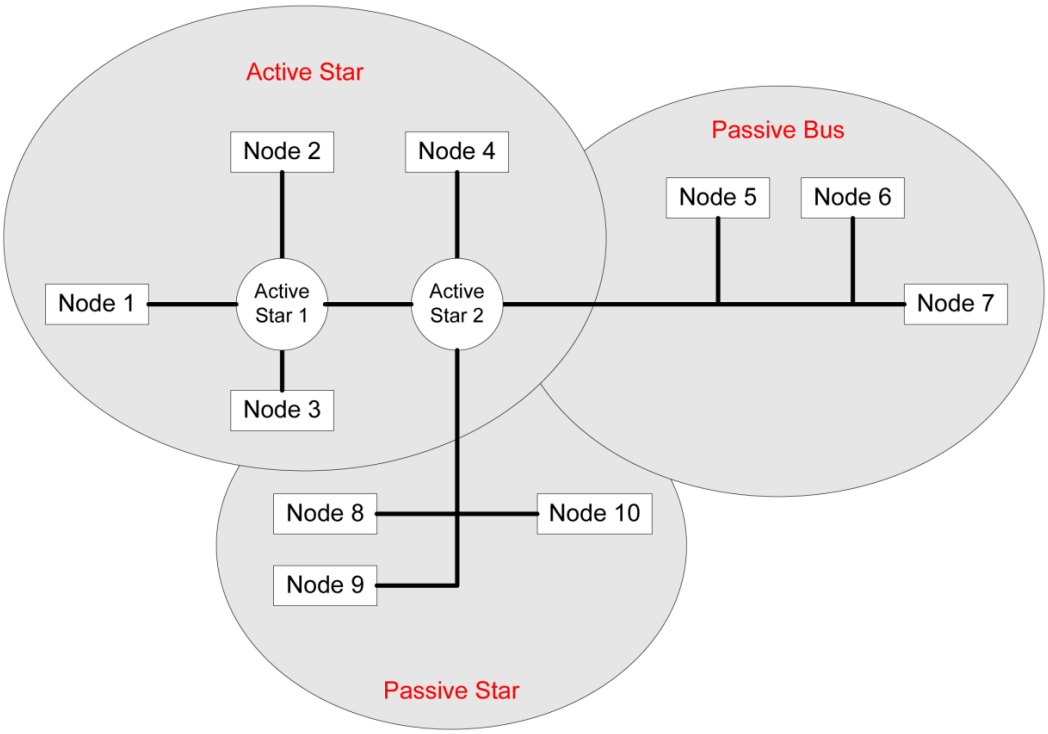
Az elküldött adatfolyam maximum két aktív csillagot érinthet míg célba ér. Ha a csillag nem formálja át a fogadott aszimmetrikus adatfolyamot, az csökkentheti a rendszer robusztusságát.



1.26. ábra: Kaszkád aktív csillag

Hibrid topológiák

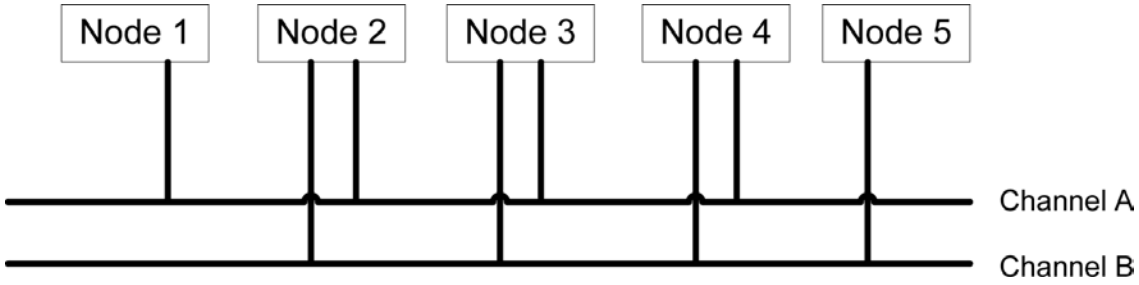
A FlexRay rendszer lehetőséget biztosít az eddig bemutatott topológiák együttes alkalmazására, ami által tágabb határokat nyit az egyes topológiákhoz képest. Sokféle hálózat alakítható ki ezzel a módszerrel, megkötést egyedül az eddigi topológia típusokra vonatkozó korlátozó feltételek adnak.



1.27. ábra: Hibrid topológia

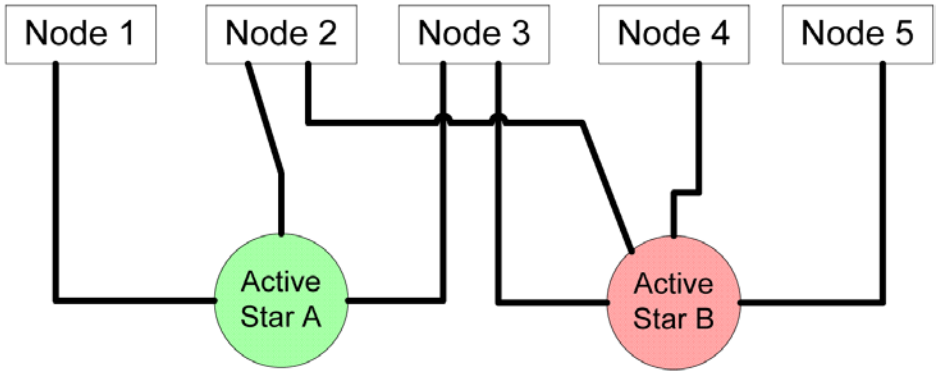
Kétcsatornás topológiák

A FlexRay kommunikációs modul megadja a lehetőséget, hogy akár két csatornát is kiszolgáljunk. Ezzel megnövelhetjük a sávszélességet, vagy javíthatjuk a rendszer hibatűrő képességét.



1.28. ábra: Kétcsatornás busz topológia

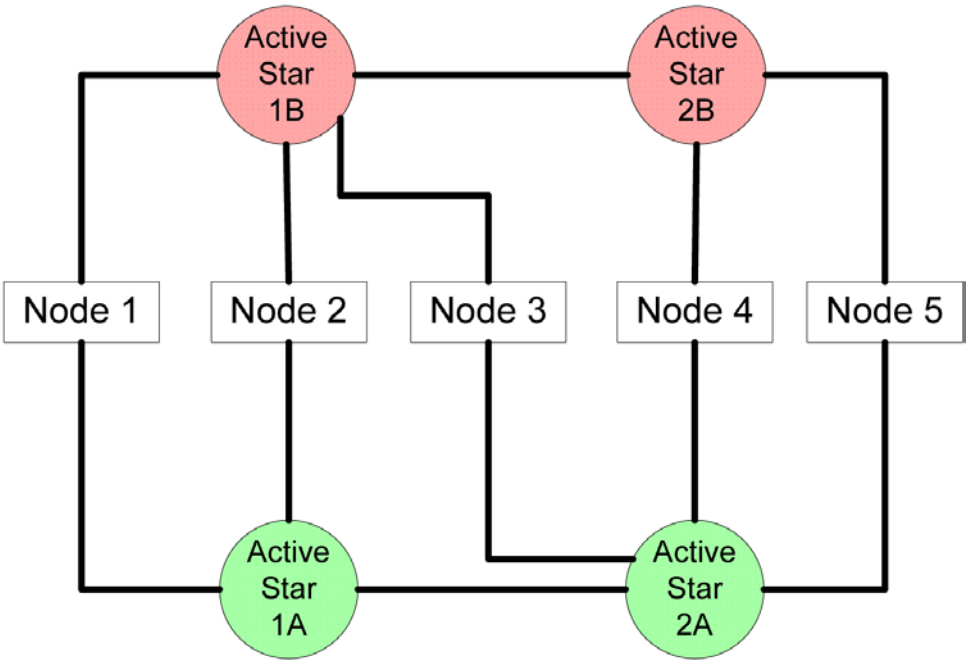
A kétcsatornás rendszerben, ahol az egyik csatornát „A”-val másikat „B”-vel jelöljük a node-ok csatlakozhatnak vagy csak az „A” vagy csak a „B” csatornához, de akár mind a kettőhöz is. Amennyiben csatlakozik az egyik csatornára, akkor az ott lévő bármelyik node-dal tud kommunikálni.



1.29. ábra: Kétcsatornás aktív csillag topológia

A kétcsatornás rendszerek minden más tulajdonsága megegyezik a fenti egycsatornás esetekkel a topológia, és korlátozó feltételek tekintetében is.

Ha szükség van arra, hogy egyszerre több cluster-hez is csatlakozzon egy eszköz, akkor azt csak különböző kommunikációs vezérlőkön keresztül teheti meg. Nem megengedett egy vezérlőnek, hogy csatlakozzon egy cluster-hez az „A” és egy másik cluster-hez a „B” csatornán.



1.30. ábra: Kétcsatornás aktív kaszkád csillag topológia

1.5.14. MOST

A MOST egy multimédiás hálózat fejlesztés eredménye, amelyet az 1998-ban létrehozott MOST Cooperation hozott létre. A konzorcium a fontosabb autógyártókból és alkatrész-beszállítókból áll.

A MOST pont-pont közötti audió és videó adatátvitelt biztosít különböző adatátviteli sebességgel. A MOST Így képes támogatni különböző a végfelhasználói alkalmazásokat, mint például rádiók, globális helymeghatározó rendszer (GPS), navigáció, videó kijelzők, és a szórakoztató rendszerek. MOST fizikai rétegként egy műanyag optikai szálat (Plastic Optical Fiber – POF) használnak, amely jobban ellenáll a különböző elektromágneses zavaroknak és nagyobb átviteli sebességre képes, mint a klasszikus rézkábel.

A jelenleg a nagyobb autógyártók közül a BMW és a Daimler-Chrysler alkalmaz MOST hálózatokat az audió és videó jelek továbbítására. Így oldják meg például a felső kategóriás BMW-kben a hátsó kamera képének a műszerfalon elhelyezkedő kijelzőre való juttatását. Az autóba épített beágyazott vezérlő pedig a kijelzett képre berajzolja, hogy a tolatás közbeni kormány pozícióval milyen íven haladna az autó és jelzi a vezető számára ha balesetet okozna. Az automatikus parkolási funkció megvalósításához is általában a MOST hálózaton küldött információkat használják az autógyártók.

Jelenleg MOST már a harmadik verzióját készítik, amely alkalmas lesz a csatornán szabványos ethernet-es frame-eket is küldeni.

1.6. Monitorozás, diagnosztika, validáció, mérés

A beágyazott rendszerek adatainak a monitorozása, mérése és tesztelése sokkal bonyolultabb és összetettebb folyamat, mint egy-egy külön álló eszközé. Ha egy eszköz vagy részegység

fejlesztési folyamatát nézzük, akkor a beagyazott rendszerek fejlesztése megoldhatatlan az eszköz monitorozása és diagnosztikája nélkül.

Az adatmonitorozásnak különböző szintjeit/módjait különböztethetjük meg. Az adatok monitorozásán érthetjük azt is, hogy a rendszerünk a környezetéből fogadja a különböző információkat és azok alapján a specifikációnak megfelelő vezérlési feladatokat végzi el. Mivel a beágyazott rendszerek szoftverei a legtöbb esetben beágyazott szoftverek, ezért fontos a szoftverek megfelelő tesztelése is. A szoftverkomponensek a működéshez szükséges beérkező információkat eltárolják a memóriába, melyet valamilyen formában tesztelni kell. A kérdés, ami felmerülhet az a következő: Jó információt lett eltárolva? Jó helyre lett eltárolva? Ezekre a kérdésekre a teszteléssel lehet választ találni. A nagyobb rendszerek esetében a monitorozás és a diagnosztika nyújthat ehhez segítséget. Monitorozáson ebben az esetben az eszköz (ECU) belső adatainak a kiolvasását értjük. A legtöbb processzor lehetőséget ad a fejlesztőknek arra, hogy debug üzemmódban a kiolvassák a kontroller memóriáját és regisztereit, így megállapítható az, hogy a beérkezett adatok a meghatározott memóriaterületre (változóba) kerültek e. Ezzel a módszerrel működés közben kiolvashatók a változók értékei is, így kényelmesebbé és hatékonyabbá teszik a hibakeresést. A hátránya ennek a módszernek az, hogy speciális hardvert igényel.

Nagyobb funkcionalitással rendelkező ECU-k esetén speciális diagnosztikai felület áll a fejlesztők és felhasználók rendelkezésére. Általában a gyártók a végfelhasználóknak csak speciálisan korlátozott felületet biztosítanak, és nem engednek meg olyan hozzáférést a végfelhasználóknak, mint amilyen a fejlesztőknek van. A sorozatban gyártott intelligens termékek esetében a gyártósor végén az eszközök a beépített diagnosztikai funkciók felhasználásával képesek lehetnek akár saját magukat letesztelni és speciális körülmények között kalibrálni is. Ezzel a gyártósor végén kiszűrhetőek bizonyos hibák és ezzel a módszerrel még a gyártáskor végrehajtott feladatokat is le lehet egyszerűsíteni. A gyártáskor felhasznált diagnosztika specifikációját a gyártók általában nem szokták kiadni, mert azzal akár az elektronikus egység akár tönkre is tehető.

Napjainkban egyik gyártó sem engedheti meg magának, hogy hibás terméket készítsen, mert annak komoly anyagi következményei lehetnek a gyártó számára. Ilyen tekintetben az autóiparban a legszigorúbbak az előírások. Az autógyártók szinte a 0.0%-os selejtarányt várják el beszállítóiktól. Ez csak megfelelő tesztekkel érhető el.

A kifejlesztett szoftverek és hardver prototípusok esetében ez ilyen formában mért arány nem értelmezhető. Az ilyen hardver- és szoftvereszközök esetében az eszköz viselkedését a specifikációban szereplő elvárt viselkedéssel kell összevetni. Ez az összehasonlítás nem egyszerű feladat, ha az eszközműködésnek a leírása több ezer oldal. Az autóiparban jelenleg használt irányelv alapján minden funkciót és viselkedést le kell tesztelni és ezeken a teszteken meg kell felelnie az eszköznek.

Mivel a tesztek elvégzése egyre nagyobb emberi és eszköz erőforrást vett igénybe és bizonyos teszteket nem lehet valós ipari körülmények között elvégezni, ezért szükséges volt új technikákat kitalálni a beágyazott rendszerek tesztelésére. Erre szolgálnak a Software- in the-loop (SIL) és a Hardware-in the- Loop (HIL) technikák, amelyekkel a tesztelés költségeit lehetett csökkenteni. A Software- in the- loop (SIL) technikával az elkészített szoftvert illesztjük be egy speciális szoftverkörnyezetbe, amely megfelelően szimulálja a kódrészlet számára a bemeneteket. A Hardware-in the- Loop (HIL) technikával a széria (hardvertől esetleg mini-

málisan különböző) hardverre töltjük bele a kifejlesztett programot és egy speciálisan kialakított tesztgép segítségével szimuláljuk a hardver számára a megfelelő bemeneteket. A kezdetekben nem volt visszacsatolás a tesztelendő eszköz számára, ezeket a teszteket open-loop teszteknek szokták hívni. Később a tesztgépekre elkészítették a különböző rendszerek matematikai modelljeit, így már megfelelően vissza lehetett csatolni a beavatkozás eredményét, ezek már closed-loop tesztek voltak.

A jelenleg iparban használt teszteket a következőképpen tudjuk csoportosítani:

- Mechanikai tesztek: Az eszköz mechanikai igénybevételét (gyorsulás, ütések, rezgés)) hatásait vizsgáló tesztek. Különböző hőmérsékletváltozások hatásai üzem közben (maximális / minimális üzemi hőmérséklet és hősokk teszt).
- Elektronikus tesztek: Az eszköz villamos tulajdonságait (túlfeszültség, alacsony feszültség, tranziens jelenségek, földzárlat hatásai a bemeneteke és kimeneteken) vizsgáló tesztek.
- Kommunikációs tesztek: A különböző speciális kommunikációs helyzetek tesztelése (Érkező információk megfelelő eltárolása, kimeneti információk megfelelő elküldése, kommunikációs gyorsasági / ismétlési / timeout tesztek, zárlatok és szakadások tesztelése.)
- Diagnosztikai tesztek: Feladata a felhasználók üzemeltetését elősegítő diagnosztikai felület tesztelése. Ide sorolhatóak a belső diagnosztikai tesztek és a hibák esetén bekövetkező hibaeltárolások tesztelése is. (Mivel ezek valamilyen kommunikációs platformon működnek, ezért akár oda is sorolható.)
- Funkcionális tesztek: Feladatuk, hogy megvizsgálják azt, hogy bizonyos helyzetekben a specifikációnak megfelelő feladatot végzi e a vezérlő. (Például egy gépjármű esetében mikor kell ABS / ESP algoritmusát futtatni, vagy mikortól lesz egy ABS beavatkozással induló megcsúszás olyan mértékű, amelynél már az ESP funkcionalitás szükséges.)
- Biztonsági tesztek: Valamilyen hibaesemény esetében a rendszer a specifikációban meghatározott módon való működését vizsgálja.
- Valós ipari körülmények között lefolytatott tesztek.

1.7. Irodalomjegyzék az 1. fejezethez

Nicolas Navet, Francoise Simonot-Lion: *Automotive Embedded Systems Handbook*, CRC Press, 2009, ISBN: 978-0-8493-8026-6

Richard Zurawski: *Networked Embedded Systems*, CRC Press, 2009, ISBN: 978-1-4398-0761-3

Microsoft Corporation MSDN Library, http://msdn.microsoft.com

ROBERT BOSCH GmbH (1991). *CAN Specification 2.0.* Robert Bosch GmbH, Stuttgart

ETSCHBERGER, K. (2001). *Controller Area Network.* IXXAT Press, Weingarten

Ferencz Balázs, Gerencsér Zoltán: Újrakonfigurálható szoftver kifejlesztése CAN alapú kommunikációs hálózatok adatforgalmának monitorozására és analízisére, Veszprémi Egyetem, 2005

FARSI, M. and BARBOSA, M. (2000). *CANopen Implementation: applications to industrial networks*. Research Studies Press Ltd., Baldock

ISO-WD 11898-1 (1999). Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling.

ISO-WD 11898-2 (1999). Road vehicles – Controller area network (CAN) – Part 2: High speed medium access unit

ISO-WD 11898-3 (1999). Road vehicles – Controller area network (CAN) – Part 3: Low speed medium access unit

Maxim: MAX481/MAX483/MAX485/MAX487–MAX491/MAX1487 Specification, http://www.maxim-ic.com

Microchip: *MCP2515 Specification*, Microchip Technology Incorporated, http://www.microchip.com

Modicon: Modicon Modbus Protocol Reference Guide, MODICON Inc. (1996)

2. fejezet

Programozható logikai kapuáramkörök, FPGA-k felépítése, fejlesztőkörnyezetek bemutatása

2.1. Bevezetés, programozható kapuáramkörök, FPGA-k felépítése, fejlesztőkörnyezetek bemutatása

Ebben a fejezetben a programozható logikai áramköröket vizsgáljuk meg, illetve a ma oly széleskörűen alkalmazott Xilinx FPGA áramkörök általános felépítését, különböző fejlesztő környezeteit ismerhetjük meg röviden. Ebben az oktatási segédletben egy konkrét FPGA-s hardveren (Digilent Nexys-2) és egy gyártó specifikus integrált fejlesztőkörnyezetben (Xilinx ISE™/WebPack) történik a feladatok tervezése és a megvalósítása VHDL leíró nyelv főbb nyelvi konstrukcióinak elsajátítása segítségével. Természetesen más gyártók (pl. Altera, Actel, Lattice stb.), akár eltérő felépítésű FPGA fejlesztő platformjait is választhattuk volna. Az ismertetendő hagyományos VHDL leíró nyelveken kívül az utóbbi évtizedben már magas szintű C-szintaxist használó nyelvek, illetve szimbólum szintű modulokból felépülő integrált fejlesztő környezetek is megjelentek, melyek fő előnye a gyors prototípusfejlesztés mellett, a rugalmasság és a tervek átültethetőségének lehetősége.

2.1.1. Xilinx FPGA-k általános felépítése

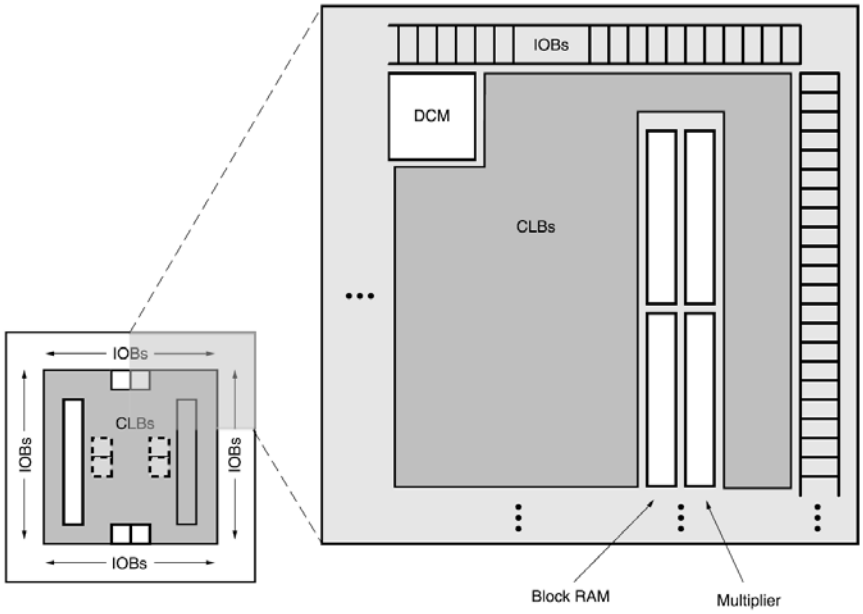
Egy általános Xilinx FPGA [*XILINX*] architektúra (amely elnevezését „felhasználó által tetszőlegesen programozhatónak”, vagy „újrakonfigurálható számítási eszközként” is definiálhatjuk) egy olyan programozható logikai eszköz, amely logikai cellák, makró cellák, illetve programozható kapcsolók és összeköttetések reguláris 2D-os tömbjéből épül fel. Az FPGA-k esetén az „in-field” kifejezés utal arra is, hogy „helyben”, azaz a felhasználó által többször programozható eszközökről van szó, szemben a gyárilag előre programozott, de a felhasználó által nem módosítható áramkörökkel. Egy logikai cella egyszerű logikai függvények megvalósítására használható, a makró cella speciális ún. dedikált építő elemeket is tartalmazhat, míg a programozható kapcsolók biztosítják a kapcsolatot a logikai cellák és makró cellák között az összeköttetéseken keresztül.

A Xilinx gyártó által kibocsátott FPGA-k között két különböző kategóriát lehet elkülöníteni egymástól:

- az egyik a Virtex™ architektúra, amely a nagyméretű, nagy-teljesítményű, de egyben drága FPGA családot jelent, míg
- a másik a Spartan™ architektúra egy költség-hatékonyabb, kisebb teljesítményű megoldást biztosít programozható logikai hálózatok kialakításához.

Mindkét családon belül további alkategóriákat lehet megkülönböztetni: Xilinx Virtex™ – Virtex™-6 sorozatok, illetve Spartan™-3 – Spartan™-6 sorozatok [XILINX], amelyek logikai cellák, és makró cellák kialakításában és sűrűségében térnek el leginkább egymástól. Mivel a kisebb, Spartan™ FPGA sorozat jobban igazodik az oktatási célú felhasználáshoz (emellett természetesen ipari és tudományos célokra is alkalmazhatóak), ezért erre a sorozatra fókuszálunk a továbbiakban. Ezekről, és más FPGA típusokról részletesen a gyártó hivatalos oldalán található adatlapokon olvashatunk [XILINX_DEVICES].

A Xilinx Spartan-3 FPGA koncepcionális felépítését és fontosabb építő elemeit az alábbi 2.1. ábra szemlélteti [SPARTAN3]:



2.1. ábra: Xilinx FPGA általános felépítése: jobb oldalon a chip egy áramköri részletének felnagyított blokkszintű belső felépítése látható a fontosabb építő elemekkel [SPARTAN3]

A Xilinx Spartan-3 FPGA fontosabb építő elemei a következők:

Logikai cellák:

- CLB (Configurable Logic Block): Konfigurálható Logikai Blokkok, amelyekben többbemenetű (4-bemenetű) LUT-ok segítségével realizálhatók a logikai függvények, és ezek kimeneteit szükség esetén egy-egy D flip-flopban tárolhatjuk el. A CLB-k továbbá multiplexereket és összeköttetéseket, valamint dedikált logikai kapukat is tartalmaznak. A CLB-ken belül ún. *slice*-okat, szeleteket különítenek el: egyet a logikai (SLICEL), egyet pedig az elosztott memória erőforrások lefoglalásához (SLICEM).

A slice-okat szokás az Xilinx FPGA-kon belüli logikai elemek mérőszámaként is tekinteni. A SLICEM-en belül egy n-bemenetű LUT egy 2^n × 1-bites memóriának is tekinthető, amelyek összekapcsolásából egy nagyobb méretű elosztott memóriát ki lehet alakítani, de a LUT konfigurálható akár egy 16-bites shift-regiszterként is.

Makró cellák:

- IOB: I/O Blokkok, amelyek a belső programozható logika és a külvilág között teremtenek kapcsolatot. Mai FPGA-k már többmint 30 különböző I/O szabványt támogatnak.
- DCM/PLL: Digitális órajel kezelő áramkör, amely képes a külső órajelből tetszőleges fázisú és frekvenciájú belső órajelek előállítására (alapfrekvencia szorzása / osztása segítségével), vagy (DCI) digitálisan vezérelhető impedancia illesztésre stb.
- PI: Programozható összeköttetés hálózat (vagy kapcsoló mátrix hálózat), amely a felsorolt építőelemek között a teremt kapcsolatot. A programozható összeköttetések beállítása, programozása során az SRAM-cellás megvalósítást alkalmazzák, amelyet bekapcsolás után inicializálni kell (de természetesen más programozási módok is létezhetnek).

A fenti általános erőforrásokon túl az FPGA-kon a további beágyazott makró cellákat, vagy más néven dedikált blokkokat, ún. *primitíveket* is találhatunk. Ezek a Blokk-RAM, a MULT, illetve a beágyazott „hard-processzor" magok lehetnek.

- BRAM (Blokk-SelectRAM) szinkron SRAM típusú memóriák, melyek nagy mennyiségű (~×100Kbyte – akár ~×10Mbyte) adat/program kód tárolását teszik lehetővé – FPGA típusától függően. Egy-, vagy dupla-portos memóriaként is lehet őket definiálni, azaz akár írni, és olvasni is lehet egyszerre a különböző IO adat portjaikon keresztül, a megfelelő engedélyező jelek kiadásával. A kapacitásuk egyenként 18Kbit (jelölése BRAM18k), azaz kb. 2 Kbyte (a paritás bitek használata nélkül).
- MULT18×18 (vagy DSP48 Blokkok nagyobb FPGA-kon, pl Spartan-3A, Virtex-4-5-6): beágyazott szorzó blokkokat jelentenek, amelyek segítségével egyszerűbb szorzási műveletet, vagy a DSP blokk esetén akár bonyolultabb DSP MAC (szorzás-akkumulálás), aritmetikai (kivonás) és logikai műveleteket is végrehajthatunk 2'komplemens számokon nagy sebességgel – szintén az FPGA típusától és kiépítettségétől függően.
- „Hard-processzor" mag: bizonyos Xilinx FPGA családok (főként a nagyobb méretű, de egyben drágább típusok) a hagyományos „beágyazható" soft-processzor magok (mint pl. PicoBlaze, MicroBlaze) mellett, rendelkezhetnek fixen beágyazott hard-processzor mag(ok)kal (pl. IBM PowerPC) is, amelyek igen nagy sebességű (~ több 100 MHz), komplex vezérlési funkciókat biztosítanak a különböző FPGA-n belüli beágyazott rendszerek tervezése esetén. Ezek a magok elkülönülten helyezkednek el a felsorolt általános logikai, és dedikált erőforrások mellett, így nem foglalják azokat, szemben a soft-processzor magokkal.

  Megjegyeznénk, hogy más gyártók (pl. Altera, Actel, stb.) FPGA-s eszközei hasonló funkcionalitású építő elemeket tartalmaznak, általában más néven, és kismértékben eltérő belső struktúrával.

Az FPGA-s rendszerek a nagyfokú flexibilitásukkal, nagy számítási teljesítményükkel, és gyors prototípus-fejlesztési – ezáltal olcsó kihozatali költségükkel – igen jó alternatívát teremtenek a valós-idejű beágyazott rendszerek tervezéséhez és megvalósításához. Ezt jól tükrözi

az általános célú mikroprocesszorok és az FPGA technológiák fejlődési üteme közötti hasonlóság, mely utóbbi is a fokozatos méretcsökkenést (scaling-down hatás) követve fejlődik [DISSZ], [MOORE]. Emiatt a Moore törvény értelmében nemcsak a hagyományos mikroprocesszorok, hanem az FPGA-kra is igaz az állítás: 1-1,5 évente az egységnyi szilícium felületre eső tranzisztorszám (programozható logikai eszközök esetén az ekvivalens logikai kapuszám) megduplázódik. Éppen ez mutatkozik meg az FPGA-k más építő elemeinek, pl. a dedikált belső memóriák és a szorzók számának növekedésében. A mai modern, nagyteljesítményű FPGA-kon (pl. Virtex-6) akár 2000 szorzó áramkör, 2-4 hard-processzor mag, illetve közel, akár 40 Mbyte belső BRAM memória is helyet foglal. Ezekről, és más Xilinx FPGA típusokról részletesen a gyártó hivatalos oldalán található adatlapokon olvashatunk [XILINX_DEVICES].

2.1.2. Digilent Inc. Nexys-2 fejlesztőkártya

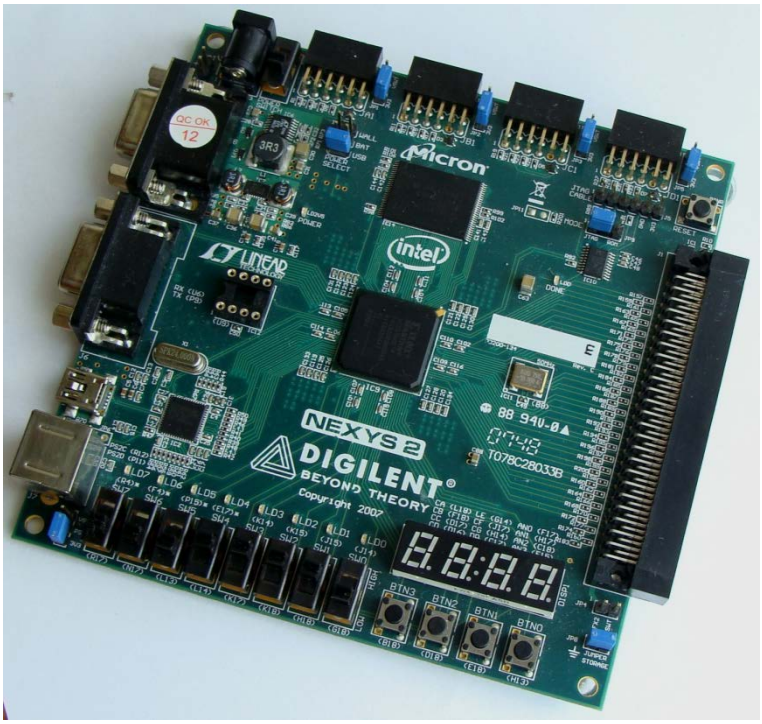
A jegyzetben a feladatok megoldása során egy olcsó, kisebb teljesítményű, de a jelen fejlesztési, oktatási céloknak tökéletesen megfelelő Spartan™-3E FPGA-t tartalmazó [SPARTAN3], 500K, illetve 1200K erőforrással is kapható Digilent Nexys-2 [NEXYS2] fejlesztő kártyát használunk. (Itt a „K” jelölés mindig az ekvivalens logikai kapuk számát jelenti 1000-es nagyságrendben). A tokozáson lévő szabványos jelölés szerint az XC3S500E megnevezés egy Xilinx Spartan-3E típusú és 500.000 ekvivalens logikai kapuszámmal rendelkező FPGA-t jelent. Mivel kb. 1-2 nagyságrenddel kevesebb dedikált BRAM18k memóriát (20-28 db), illetve MULT18×18 szorzó (20-28 db) erőforrást tartalmaz, mint a mai nagyteljesítményű FPGA sorozatok (pl. Virtex™-5, Virtex™-6, Spartan™-6, [XILINX_DEVICES]), ezért ez egyben a kisebb sorozatok lehetséges alkalmazási területeit is meghatározza. Szintetizálható VHDL leírások hardveres verifikálásának bemutatására, valamint az egyszerűbb felépítésű beágyazott rendszerek implementálásához viszont éppen ez lehet a megfelelő választás. A Spartan-3E-s FPGA-ról további részletes információkat a „Spartan-3E FPGA Family: Data Sheet” adatlapon lehet olvasni [SPARTAN3].

A Nexys-2 fejlesztő kártyán található Spartan-3E FPGA 500K/1200K a következő 2.1. táblázat szerinti logikai illetve dedikált erőforrásokat tartalmazza:

2.1. táblázat: A Digilent Nexys-2 fejlesztő kártyákon lévő FPGA-s eszközök rendelkezésre álló erőforrásainak száma

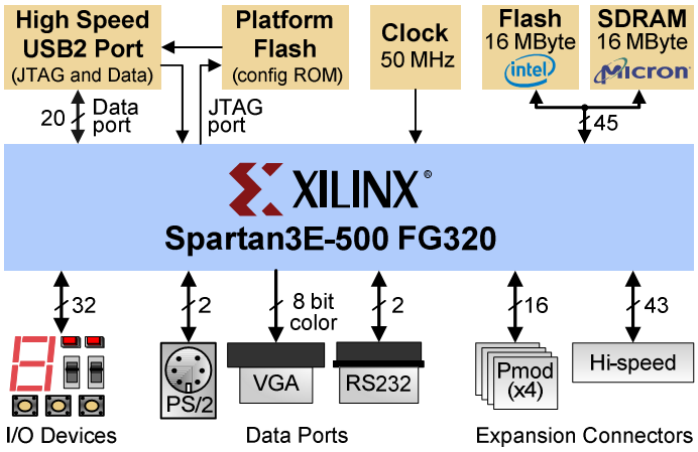
FPGA típus	CLB teljes (db)	Slice-ok (db)	4-LUT / FF (db)	BRAM 18k (db)	MULT 18x18 (db)	Elosztott RAM (bit)	DCM (db)	Tokozás IOB
XC3S500E	1164	4656	9312	20	20	74496	8	FG-320
XC3S1200E	2168	8672	17344	28	28	138752	8	FG-320

Az alábbi *2.2. ábra* a Digilent Nexys-2-es fejlesztőkártya felépítését szemlélteti:



2.2. ábra: Digilent Nexys-2 fejlesztő kártya fotója [DIGILENT]

A Nexys-2 kártya rendelkezik 16 MByte méretű külső Micron P-SDRAM típusú memóriával is, illetve egy szintén 16 MByte méretű Intel Flash RAM memóriával, amelyekben nagyobb mennyiségű adatot is el lehet tárolni, miközben pl. az FPGA-n az adatok feldolgozása megtörténne. A Nexys-2 kártya számos külső periféria illesztését is lehetővé teszi: pl. RS232, PS2 (bill. / egér), VGA kimenet, HighSpeed USB 2.0, ahogyan az *2.3. ábrán* is látható:



2.3. ábra: Nexys-2 fejlesztőkártya felépítése és külső perifériái [DIGILENT]

Az USB kapcsolatot egy Cypress USB-s mikrovezérlő biztosítja (high-speed módban). A kártya nemcsak USB-n, hanem a szabványos JTAG portokon keresztül is programozható, illetve a Flash-EEPROM-ban tárolt információval is inicializálható (amely szabadon konfigurálható). Ebben az esetben a „beégetett tartalommal” konfigurálva már nem szükséges a PC-vel összekapcsolni, ezért a Nexys-2 FPGAs kátya akár egy önálló működésre képes (ún. „stand-alone”) rendszerként is használható. Az előbb említett perifériákon kívül szabadon programozható kapcsolók (8), nyomógombok (8), LED-ek (8), és 7-szegmenses kijelzők (4) kaptak helyet, pl. a működés közbeni jelek, esetleges hibák jelzésére, külső beavatkozásra. Az eszköz ára 100 $ – 150 $ között mozog, az FPGA típusától függően [DIGILENT].

A kártyához a PMOD csatlakozók (bővíthető periféria modulok) segítségével további számos variálható analóg/digitális/kevert jelű periféria illeszthető, amelynek részletes listája a hivatkozáson keresztül megtekinthető [PMOD]. Áruk 10 – 50$ nagyságrend között mozog.

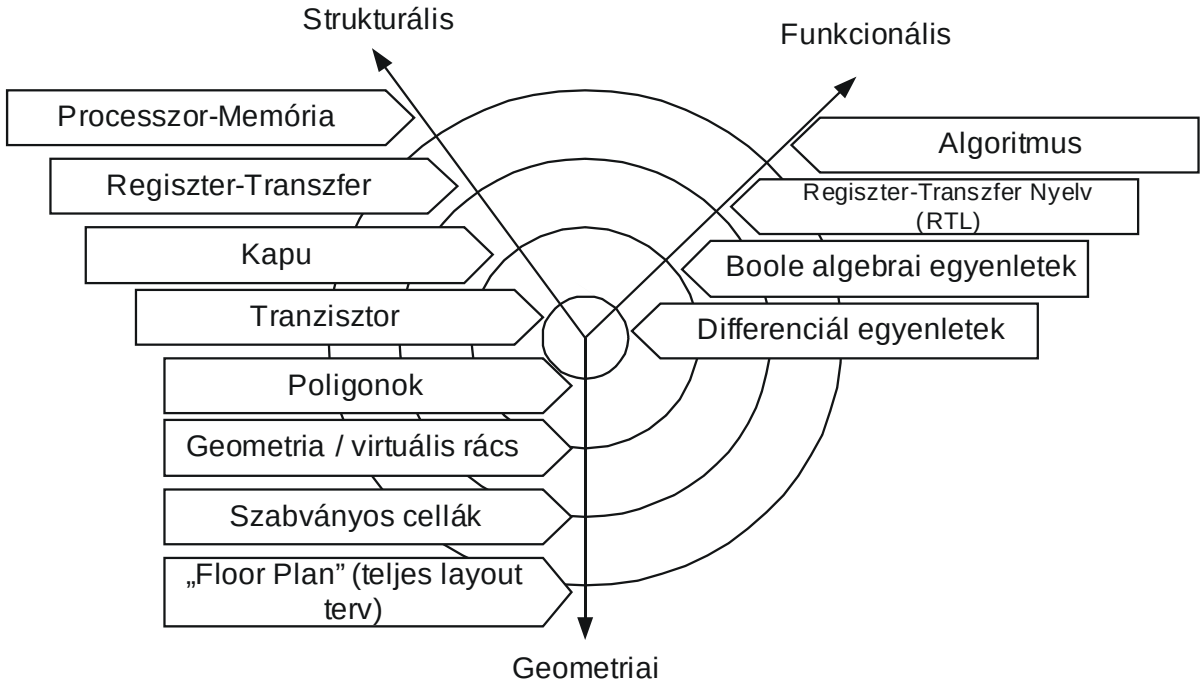
Amennyiben a kártyát videó jelek feldolgozására is fel kívánjuk felhasználni, akkor illeszthető egy külső videó dekódert tartalmazó bővítő kártya: V-DEC1 [VDEC], amely a több szabványos Composite, S-Video, Component Video bemenetek, illetve a PAL/NTSC/SECAM jelek kezelését is támogatja. A kisebb kártyákon, így a Nexys2-esen sem kell aktív hűtés használat közben.

Ezen bővítmények nagyban megnövelik a Nexys-2 fejlesztőkártya alkalmazási területeit: jelfeldolgozás, képfeldolgozás, autóipari alkalmazások stb.

2.1.3. Modellezés: tartományok és absztrakciós szintek

Egy adott rendszer tervezése során, különböző tervezési szempontok szerint dolgozhatunk, azaz a rendszer modellezését különféle tartományokon és azokon belül eltérő absztrakciós szinteken végezhetjük. Az ismert Gajski és Kuhn [GAJSKI] féle Y-diagram szerinti felírásban (lásd *2.4. ábra*) három lehetséges tartományt különböztethetünk meg*: funkcionális, strukturális és geometriai* tartományokat. (Ahogy a későbbiekben említésre kerül a VHDL leírások során, általánosan a viselkedési és a strukturális tartomány szerinti tervezést alkalmazzák.)

A **funkcionális tartományban** a rendszer viselkedésének leírását adjuk meg, de nem foglalkozunk annak részleteivel, ahogyan a funkciót megvalósítottuk. A funkcionális tartomány a ’legabsztraktabb’ megközelítést jelenti, mivel a teljes rendszer viselkedése megadható az algoritmikus szinten. A következő szint a regiszter-átviteli szint (Register-Transfer) vagyis a regiszter-memória-processzor elemek közötti transzformációk megadása. Az adatot egy regiszter, vagy egy memória adott cellájának értéke, míg a transzformációkat az aritmetikai és logikai operátorok jellemezhetik. Az adat és vezérlési információ áramlása definiálható akár a regiszter transzfer szintű nyelvi leírásokkal (RT Language) is, vagy hatékonyan szemléltethetők grafikus módon az adat-, és vezérlési- folyam gráfok segítségével (DFG, CFG). A harmadik szint a hagyományos logikai szint, ahol egy-egy funkció megadható a Boole-algebrai kifejezések, igazságtáblázatok segítségével. Végül az utolsó szinten az áramkör működését definiáló differenciál-egyenleteket kell definiálni, amely a hálózat feszültségében, és áramában történő változásokat hivatott leírni.



2.4. ábra: A modellezés tartományai és absztrakciós szintjei (Y diagram)

A **strukturális tartományban** viszont pontosan a rendszer belső elemeinek felépítését és azok között lévő összeköttetéseket vizsgáljuk. A strukturális tartomány legfelső szintjén kell a fő komponenseket és összeköttetéseit definiálni, amelyet processzor-memória kapcsolatnak (processor-memory switch) is hívnak. A második szint itt is a regiszter-átviteli szint, amely egyrészt az adatútból (data path), másrészt vezérlési szakaszokból (control path, sequence), szekvenciákból áll. A harmadik szint a logikai szint, amelyben a rendszer struktúrája az alapkapuk és összeköttetéseik segítségével építhető fel. A negyedik, legalacsonyabb szinten a tranzisztorok, mint az áramköri rajzolatok (layout) elemi egységeit kell tekinteni.

Végül a **geometriai tartomány** azt mutatja, ahogyan a rendszert elhelyezzük, leképezzük a rendelkezésre álló fizikai erőforrások felhasználásával (felület). Ebben a tartományban a legfelső hierarchia szinten, a szilícium felületen elhelyezett vagy ún. „kiterített” VLSI áramkört kell tekinteni (floor-plan): FPGA esetén tehát magukat a logikai cellákat és makrócellákat, valamint a közöttük lévő összeköttetéseket. A következő szintet a szabványos alapcellák (Standard Cell) könyvtárai képezhetik, amelyeket, mint technológiai adatbázist használhatunk fel a regiszterek, memóriák, vagy akár aritmetikai-logikai egységek megvalósításához. A harmadik szinten az egyedi tervezésű integrált áramkörök (ASIC) geometriája egy virtuális rácson adható meg. Végül az utolsó, legalacsonyabb szinten a poligonokat kell megrajzolni, amelyek csoportjai az áramkör egyes maszk-rétegeinek feleltethetők meg.

Manapság a számítógéppel segített elektronikus tervezői eszközök segítségével egy-egy tartományon belül nem kell minden szintet külön-külön pontosan definiálni, elegendő a tervezést a legfelsőbb absztrakciós szinteken elvégezni, amelyből az alacsonyabb szintek automatikusan generálódnak (EDA).

2.1.4. Magas-szintű hardver leíró nyelvek

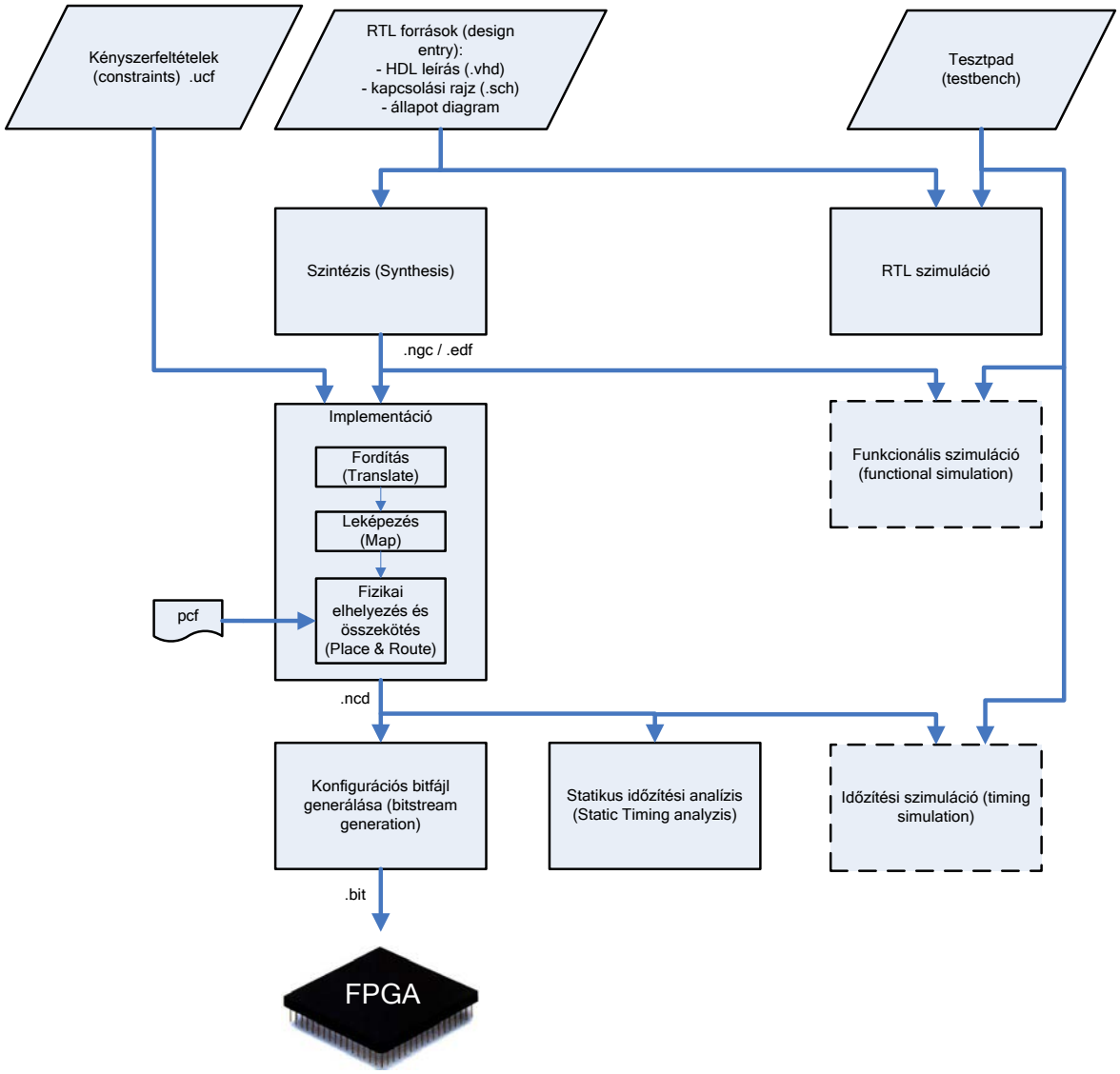
Napjainkban (2011) az FPGA-s eszközökön történő fejlesztésre már többféle lehetséges alternatíva is rendelkezésre áll:

- **Hagyományos/tradicionális magas-szintű hardver leíró HDL nyelvek** (pl. VHDL [*VHDL87*], Verilog [*VERILOG*] stb), vagy ezek kiterjesztésével megadott verifikációs nyelvek (pl. SystemVerilog),
- **Algoritmikus nyelvek vagy ún. „C→FPGA szintézist" támogató programnyelvek**: ANSI-C szintaktikát követő különböző magas-szintű nyelvek (pl. Agility Handel-C [*HANDELC*], ImpulseC [*IMPULSEC*], Mentor Graphics Catapult-C [*CATAPULTC*], System-C [*SYSTEMC*] stb.), illetve
- **Modell alapú integrált fejlesztő rendszerek** fejlett GUI támogatással (pl. The MathWorks MatLab [*MATLAB*], National Instruments LabVIEW [*LABVIEW*]).

Ezek a fő tervezési módszerek azonban a legtöbbször nem teljesen különülnek, vagy különíthetők el egymástól. A magas szintű C-alapú, illetve modell leíró nyelvek integrált fejlesztő rendszerei (IDE) általában a hagyományos VHDL/Verilog generálásának fázisáig, vagy a szabványos EDIF (Electronic Design Interchange Format) netlista generálásáig biztosítják, és egyben gyorsíthatják fel az algoritmus fejlesztés-tervezés menetét. Ettől a lépéstől kezdve viszont a kiválasztott FPGA gyártó által támogatott szintézis eszközökkel (EDA) folytatódik az FPGA-n lévő blokkok (primitívek) automatizált leképezése, elhelyezése majd összekötése (megjegyeznénk, hogy mindez manuális lépésekben is történhet). Ezeket hívják együttesen MAP → PLACE → ROUTE fázisoknak, amelyek az áramkör komplexitásának függvényében hosszabb-rövidebb idő alatt generálják a kimenetet, lásd következő alfejezetet. A kimenet végül egy generált bit-folyam lesz (pl. Xilinx .bit, vagy Altera .sof konfigurációs fájl), amelyet a Nexys-2 fejlesztő kártyára szabványos USB, vagy JTAG USB programozó kábelek segítségével tölthetünk le.

2.1.5. Tervezés folyamata (Xilinx design flow)

Az FPGA alapú rendszertervezés folyamatát a Xilinx ISE fejlesztő környezetben történő egymást követő lépéseken keresztül demonstráljuk. A fejlesztés egyszerűsített folyamatát a következő *2.5. ábra* szemlélteti:



2.5. ábra: FPGA-alapú rendszertervezés folyamata Xilinx ISE rendszerben

A *2.5. ábra* bal oldali ága az egymást követő fő tervezési lépéseket szemlélteti, amely során a cél a terv folyamatos tökéletesítése, illetve az FPGA konfiguráció előállítása. Konzisztens módon ehhez egy jobboldali párhuzamos ellenőrző ág (validáció) is kapcsolódik, minden szinten egy-egy szimulációs lépést feleltethetünk meg a tervezési folyamat egy-egy szintjével. Látható módon a fejlesztési folyamat egy magas absztrakciós szintű RTL HDL leírásból indulhat ki, amely egészen az alacsony eszköz-, vagy cella-szintű konfiguráció generálásáig tart. Itt kell azonban megemlíteni, hogy a tervezés kezdetén nem csak a hagyományos HDL leíró nyelveket, hanem kapcsolási rajzokat (séma – pl Xilinx Schematic Editor [*XILINX_SCHEMATIC*]), illetve a sorrendi hálózatok működését leíró állapot diagramokat is megadhatunk (pl. Xilinx StateCAD [*XILINX_STATE_CAD*]), amelyekből azután HDL leírás generálható. Ezek bemutatása túlmutat a jelenlegi jegyzet témakörén, de további információk

találhatóak a megadott irodalmi hivatkozásokon. Végezetül az elkészült konfigurációs fájlt (.bit) letölthetjük az FPGA-ra. A tervezés fontosabb lépései a következők:

1.) Moduláris, vagy komponens alapú rendszertervezés

- Lentről-felfelé (bottom-up), vagy fentről-lefelé (top-down) irányú tervezési metodika alkalmazása
- HDL leírások, kapcsolási rajzok, vagy állapot diagramok megadása a tervezés kezdeti fázisában (=design entry), illetve
- felhasználói-tervezési megkötések, kényszerfeltételek (user constraints – ucf) rögzítése (lásd később a szintézis és implementáció során).

2.) Ezzel párhuzamosan a tervezés egyes szintjein, illetve a legfelsőbb hierarchia szinten **HDL tesztkörnyezet, ún. 'tesztágy', vagy 'tesztpad' (=testbench) összeállítása**

- RTL / viselkedési szimuláció tesztvektorok, mint gerjesztések megadásával, amely még PC-n történik,

3.) Szintézis és implementáció:

- Szintézis: „logikai szintézis" során a HDL leírás általános kapu-szintű komponensekké transzformálódik (pl. logikai kapuk, FFs)
- Implementáció 3 fő lépésből áll: TRANSLATE (Compiler) → MAP (Fit) → PAR (Placer & Router / Assembler) lépéseinek sorozata. Ezeket és a zárójelben megadott kifejezéseket nemcsak a Xilinx fejlesztési folyamatában, hanem más FPGA gyártó integrált fejlesztő környezeteiben is közel hasonló módon nevezik. Ha az implementáció bármely adott lépésében hiba történt, akkor az implementáció további lépései már végre sem hajtódnak. Az adott lépésben meglévő hibát először ki kell javítani – ezt az Xilinx ISE üzenet ablakában megjelenített információk alapján láthatjuk – és csak utána tudjuk a további implementációs lépéseket végrehajtani. Implementáció lépései a következők:
  - TRANSLATE: több, esetleg eltérő hardver leíró nyelven megírt tervezői file összerendelése (merge) egyetlen netlist-fájlba (EDIF). A netlista fájl tartalmazza a komponensek, és összeköttetéseik szabványos szöveges leírását.
  - MAP: az elkészült logikai tervnek egy adott technológia szerinti leképezése (technology mapping), amelyet az előző, TRANSLATE lépésben generált EDIF netlistából kapunk: a kapukból pl. CLB-ket képez le
  - PAR: végleges „fizikai" áramköri tervet hoz létre a periféria kényszerfeltételeitől függően (.pcf – peripheral constraint file). Ebben a fázisban az előző leképezés (MAP) során kapott komponenseket az FPGA-n rendelkezésre álló és azonosítható fizikai cellákon helyezi el (pl. XnYm). Az elhelyezett fizikai komponenseket végül a routing eljárás különböző optimalizálási szintek és algoritmusok szerint 'huzalozza össze', az egyes tervezési megkötéseket, kényszerfeltételeket is figyelembe véve (.pcf). A PAR fázis befejezésével egy .ncd fájl keletkezik.

- Statikus időzítési analízis (STA): időzítési paraméterek (timing parameters) meghatározása (maximális órajel frekvencia, kapuk megszólalási idejének, vagy vezetékek jelterjedési késleltetések hatásának vizsgálata stb.)

4.) Bit-folyam (bit-stream), mint konfigurációs fájl generálása (.bit) és letöltése FPGA-ra (CLB-k, programozható összeköttetések beállítása, konfigurálása minden egyes inicializációs fázis során szükséges, köszönhetően annak, hogy a Xilinx FPGA-kat alapvetően az SRAM alapú technológia jellemzi).

A fenti *2.5. ábrán* szaggatott vonallal jelölt részeket, úgymint a funkcionális szimulációt, csak az FPGA szintézis után lehet végrehajtani, míg az időzítési szimulációt az implementációs lépések végén lehet elindítani. A funkcionális szimuláció magának a szintézis eljárásnak a helyességét ellenőrzi (.ncd, vagy .edf alapján), míg az időzítési szimuláció során a végleges fizikai netlista (.ncd) és a konkrét időzítési paraméterek vizsgálata történik. A köztes leírások komplexitásától függően ezekben a párhuzamos ágakban lévő szimulációs tesztek jelentős időt vehetnek igénybe.

2.2. Bevezetés a VHDL nyelv használatába. Nyelvi típusok, kifejezések, operátorok

A fejezetben a VHDL, mint magas-szintű hardver leíró nyelv kialakulásának hátterét, elméleti alapjait ismertetjük röviden. Bemutatásra kerülnek a VHDL nyelv alap típusai, kifejezései, operátorai, néhány szemléltető példával alátámasztva.

2.2.1 VHDL kialakulásának rövid háttere

A VHDL betűszó a „VHSIC HDL” (Very High Speed Integrated Circuits – Hardware Description Language) rövidítésből származik, melynek jelentése: Nagyon-nagy sebességű Integrált Áramkörök – Hardver Leíró Nyelve. A VHDL nyelvet eredetileg az amerikai Védelmi Minisztérium kérésére kezdték el kifejleszteni (DARPA program keretében), azzal a feltett szándékkal, hogy az ASIC áramkörök viselkedését dokumentálhassák. A VHDL szintaxisa az ADA programnyelven alapul [*ADA*]: erősen típusos, jól strukturált, objektum-orientált magas-szintű programnyelv. Akkoriban (1980-as évek) ez még egy nagy, komplex kézikönyvet jelentett, amely tele volt implementáció-specifikus részletekkel. További előrelépés volt, hogy ezeket a dokumentációkat (terv leírásokat) a logikai szimulátorok később be tudták olvasni, valamint elkészültek az első logikai szintézis eszközök, amelyek a kimenetükön digitális áramkörök fizikai megvalósításait definiálták. A VHDL nyelvvel szemben felmerült fontosabb elvárások a következők voltak:

- specifikációs nyelv (terv),
- képes legyen áramköri struktúrát leírni (milyen elemekből épül fel),
- képes legyen áramköri viselkedést (behaviour) leírni (mit kell csinálnia),
- funkcionális tulajdonságok megadása (hogyan lehet interfészt készíteni hozzá),
- szimulálhatóság (viselkedés),

- szintetizálhatóság (leképezni programozható logikai eszközökre),
- fizikai tulajdonságok (milyen sebességgel működhet).

A VHDL kezdeti verziója IEEE 1076-1987 szabványként vált ismertté [*VHDL87*], amely napjainkig számos módosításon, bővítésen ment keresztül: 1993, 2000, 2002, és a legújabb a 1076-2008-as verziók jelentek meg. A korai IEEE 1076-1987 változat az adattípusok kezelésének széles spektrumát biztosította: numerikus (egész, valós), logikai (bit, boolean), karakteres és fizikai mennyiségek (pl. idő, hosszúság), valamint az ezekből képzett tömbök (pl. bit_vector, string) definiálását is támogatta. Probléma a többértékű logikai kifejezések definiálásával volt, amelyet az IEEE 1164 szabványban orvosoltak: 9-értékű logikai kifejezések (jel erősség: pl. weak, strong, unknown stb.), illetve új skalár (std_ulogic) és vektor (std_ulogic_vector) típusok lettek bevezetve.

A VHDL jelenleg egyike a legfontosabb szabványos magas-szintű hardver leíró nyelveknek az EDA (számítógéppel segített elektronikai tervezés) területén. Ezt annak köszönhető, hogy a nyelv alapleírása teljesen eszköz (primitív) független maradt, ezért a megfelelő szintéziseszközök széles skálája készülhetett el hozzá. Mivel legáltalánosabban az IEEE 1076-1993-as szabvány használata terjedt el, amelyet a mai szintézis eszközök legtöbbje támogat, ezért a jegyzetben is ennek a VHDL-93 szabványnak [*VHDL93*] a használatát, szintézis szempontjából fontos nyelvi konstrukcióit mutatjuk be a példákon keresztül.

A VHDL-t, mintegy tervezendő új irányvonalként, nemcsak hardver leíró nyelvként, hanem hardver-verifikációs nyelvként is be kívánják vezetni a közeljövőben, hasonlóan a már létező SystemVerilog nyelvhez. Az utóbbi években megjelent a VHDL-AMS (Analog-Mixed Signal) szabvány is, amely digitális mellett már az analóg, illetve kevert-jelű áramköri tervezést is támogatja.

A VHDL kód teljességének és nagyfokú rugalmasságának következtében ugyanaz a feladat többféle nyelvi konstrukcióval és kódolási stílussal is leírható. A jegyzetben vázolt példaprogramok is kivétel nélkül ilyenek: egy lehetséges megvalósítását mutatják a problémának. Mindemellett – a HDL leírások kezdeti motivációjának köszönhetően – a digitális rendszertervek megalkotásakor és vizsgálatakor a viselkedési szimuláció megadására törekszenek, amely egyrészt szükségtelenül komplex lehet, másrészt az FPGA szintézis szempontjából rengeteg felesleges nyelvi konstrukciót is tartalmazhat. Ezt a szemléletmódot követi a legtöbb angol, illetve magyar nyelven megjelent szakkönyv és jegyzet is [*ASHENDEN*], [*BME*], [*KERESZTES*], azonban a viselkedési szimulációval ellenőrzött áramköri tervek, korántsem biztos, hogy közvetlen módon egy FPGA áramkörre szintetizálható HDL leírások szerint készültek, illetve hogy a tényleges fizikai erőforrásokon implementálhatók. Az elemi „építőkövekből”, sablonokból (template) történő építkezés egy jó kódolási gyakorlat lehet, ha szintetizálható leírást, végső soron fizikai FPGA magvalósítást kívánunk adni. Jelen jegyzetben ezt a módszertant fogjuk követni, azaz a nyelvi konstrukciók gyűjteményéből a legfontosabb szintetizálható készletet vizsgáljuk, más nemzetközi szakirodalmak megközelítéséhez hasonlóan [*CHU*], [*HASKELL*].

2.2.2. VHDL alapvető nyelvi konstrukciók

A VHDL kód nyelvi konstrukcióit a következő szemléletes példán keresztül mutatjuk be.

Feladat 1/a.

Tekintsük a következő 1-bites egyenlőség összehasonlító (komparátor) áramkört: két bemenete *A, és B*, valamint egy kimenetét jelöljük *EQ*. Az áramkör működését leíró igazságtábla a *2.2. táblázatban* adott: két bemenő független változó esetén ad logikai '1' értéket a kimeneten. A feladat során következő alapkapukat használhatjuk: NOT, AND, OR. A kimeneti függvény Boole-algebrai alakja a következő *(2.1)*:

$$EQ = A \cdot B + \overline{A} \cdot \overline{B} = \overline{A \oplus B} = A \odot B \qquad (2.1)$$

2.2. táblázat: 1-bites számok egyenlőségét összehasonlító áramkör (EQ) igazságtáblája

bemenetek		kimenet
A	B	EQ
0	0	1
0	1	0
1	0	0
1	1	1

Az áramkör kapu-szintű megvalósításának egy lehetséges VHDL leírása a következő:

```
-- Feladat 01 /a – 1bites egyenlőség összehasonlító áramkör
library ieee;
use ieee.std_logic_1164.all;

entity egyenloseg_1_bit is
   port(
      A, B: in std_logic;
      EQ: out std_logic
   );
end egyenloseg_1_bit;

architecture kapu_szintu of egyenloseg_1_bit is

begin
   -- Boole kifejezés
   -- de megadhatjuk xnor ekvivalencia operátorként is
   EQ <= (A and B) or (not A and not B);

end kapu_szintu;
```

Forráskód megadásának konvenciói

A VHDL nyelv a kis és nagybetűs kifejezések között nem tesz különbséget (nem case-sensitive). A forráskód szabadon formázható, a sorok elején a behúzásoknál tabulálást vagy üres karaktereket használva, de a tervező mérnököknek, ha csapatban dolgoznak, mindenképpen érdemes rögzíteni, hogy milyen előre meghatározott konvenció szerint kívánnak dolgozni. (Megjegyezném, hogy ezekre a formázásokra ma már automatikus ellenőrző rendszerek, ún. 'EDA linting' szoftver eszközök is léteznek, fejlesztő által definiálható szabályokkal).

Azonosítók

Az azonosítóknak, mint az egyes objektumok (például az A, B, EQ stb.) neveiben az angol abc betűit, számjegyeiket és az '_' (aláhúzást) is megadhatjuk. Azonban itt néhány fontos kitétel, hogy: az azonosító csak betűvel kezdődhet (számmal és aláhúzással nem) utána viszont bármi állhat, de több aláhúzás nem állhat egymás után és az azonosító sem lehet végén: pl „kapu_szintu" megengedett, viszont a fentiek értelmében a 0A, _A, A__, vagy A0_ azonosítók használata nem engedélyezett. (Megjegyezném, hogy a fenti példában ugyan nagy betűs azonosítókat definiáltunk a hagyományos Boole algebrai azonosságok felírását követve, azonban a későbbiek során a legtöbb esetben a konstans, illetve generic értékeknek, adunk meg általában csupa nagybetűs azonosítókat).

Foglalt szavak

Hasonlóan más program nyelvekhez, a VHDL nyelvben is vannak olyan foglalt szavak, vagy kulcsszavas kifejezések, amelyeket egy objektum azonosítójának nem adhatunk meg, mivel azt a VHDL nyelvi környezet egy foglalt utasításának értelmezi. A forráskód szerkesztő a foglalt szavakat vastagított betűvel jeleníti meg. Ilyenek lehetnek a következők:

2.3. táblázat: Foglalt szavak a VHDL nyelvben

abs	configuration	impure	null	rem	type
access	constant	in	of	report	unaffected
after	disconnect	inertial	on	return	units
alias	downto	inout	open	rol	until
all	else	is	or	ror	use
and	elsif	label	others	select	variable
architecture	end	library	out	severity	wait
array	entity	linkage	package	signal	when
assert	exit	literal	port	shared	while
attribute	file	loop	postponed	sla	with
begin	for	map	procedure	sll	xnor
block	function	mod	process	sra	xor
body	generate	nand	pure	srl	
buffer	generic	new	range	subtype	
bus	group	next	record	then	
case	guarded	nor	register	to	
component	if	not	reject	transport	

Megjegyzések

A VHDL nyelvben a kommenteket kettős '– –' gondolatjelek után tehetjük. Csak egy-soros komment engedélyezett. Ha több sorba szeretnénk megjegyzéseket tenni, akkor minden sor elején jelölni kell.

```
-- Feladat 1.1
```

vagy

```
-- ez egy megjegyzés
-- itt folytatódik a megjegyzés
```

Számok, szövegelemek

VHDL nyelven is lehetőség van a számok, vagy szövegelemek (literal-ok) megadására, különböző számrendszert használva. A decimális, valós számokat a különböző alapszámokkal definiált számrendszerekkel, illetve csoportosítókkal (pl. ezres csoportosító, vagy bináris számoknál 4-es bitminta csoportosító) is megadhatjuk a következő módon:

```
--decimális számok
23, 0, 146, 46E5

-- valós számok
23.1, 0.0, 3.14159

-- valós számok Exponenssel es Mantisszával (decimális értékben) jelölve
46.E5, 1.0E+12, 34.0e-08

--Számrendszerek, ahol n#x#    jelenti n a számrendszer alapját,
--alapszámát, x pedig az értékét
2#111111011#, 16#FD#, 16#0fd#, 8#0375#   --egészek
2#0.1000#, 8#0.4#, 12#0.6#       -- = dec (0.5) lebegőpontos
--értékek

123_456, 3.141_592_6, 2#1111_1100#    -- csoportosítok '_' használatával
--Számrendszerek, ahol n#xy#  jelenti az xy szám 'n'-es számrendszer beli
alapját,
```

Karakterek, karakter-füzérek és bit-füzérek

A VHDL karaktereit, karakter-füzéreit (string), illetve bitfüzéreit (bit-string) más nyelvekhez hasonlóan a következő módon definiálhatjuk:

```
-- Karakterek
‘A’, ‘z’, ‘,’, ‘1’, ‘0’, ‘_‘ – karakter kódok

-- karakter-füzérek
"A sztring"
"" –-ez az (empty) üres string

-- Bitfüzérek
B"0110001", b"0110_0111" –bináris szám, illetve ekvivalens formája
--használható csoportosító ’_’ karakterrel kombinálva is
O"372", o"00" –oktális szám,
X"FA", x"0d"  --hexadecimális,
D"23", d"1000_0003" --decimális
```

Objektumok

Egy VHDL modellben a névvel azonosított, és típussal megadott objektumokat következő 4 osztályba lehet sorolni:

- Konstansok (constants),
- Jelek (signals),
- Változók (variables),
- Fájlok (file).

Jelen jegyzetben az első kettő objektum típussal foglalkozunk részletesen a szimulálható, de egyben szintetizálható VHDL kódok megvalósíthatóságát figyelembe véve. A konstansok értékadás után már nem kaphatnak új értéket. A jelek és a változók közötti legfontosabb különbség az, hogy a jeleket használjuk különböző entitások, komponensek összekapcsolására, és így értéküket csak valamely esemény hatására változtatják meg, míg a változóknak új érték bármikor, a kezdeti értékadás után is megadható. Másik fontos különbség, hogy míg a konstans, és belső jelek bevezetését az architektúra leírás deklarációs részében lehet/kell megadni, addig a változókat csak az arhitektúra törzsén belül, pl. process()-ben lehet definiálni (lásd később).

Tervezői könyvtárak és csomagok

A Feladat-1.a első két sorában lévő utasítások:

```
library ieee;
use ieee.std_logic_1164.all;
```

megadásával biztosíthuk egyrészt, hogy a szabványos IEEE tervezői könyvtár (IEEE 1076 – VHDL [IEEE]), másrészt az IEEE könyvtárból meghívja a szabványos, előre definiált std_logic_1164 es csomagok (package) betöltődjenek a tervbe [IEEE1164]. Mind a csomagban, mind pedig a csomagokat tartalmazó könyvtárakban a forráskódhoz további előre definiált, vagy akár általunk egyedi módon definiát adattípusok, operátorok és függvények találhatóak, amelyek a fejlesztés során a 'use' kulcsszó segítségével tölthetők be, természetesen az adott tervezői könyvtár megadásával.

Entitás deklarációk

A magas szintű hardver leíró nyelveken definiált digitális áramköröket hierarchikusan egymásra épülő modulok, komponensek alkotják. Ezeket a modulokat, mint elemi egységeket a VHDL-ben entitásoknak nevezzük. Minden entitás rendelkezhet egy vagy több I/O porttal, amelyek az entitás és a külvilág között teremtenek kapcsolatot (így más hierarchia szinteken definiált entitások is összekapcsolhatóak segítségükkel). Egy entitás lehet a digitális tervezés legmagasabb hierarchia szinten definiált ún. 'top-level' modulja is, de lehet akár egy olyan komponens is, amely része egy másik modulnak.

A korábbi 1.a példa `egyenloseg_1_bit` nevű entitásának deklarációja a következő:

```
entity egyenloseg_1_bit is
   port(
      A, B: in std_logic;
      EQ: out std_logic
   );
end egyenloseg_1_bit;
```

ahol A, és B **in** módú, azaz bementeként, míg az EQ **out** módú, azaz kimenetként van deklarálva a portlistában. A példában mindegyik jel std_logic típusú (ieee.std_logic_1164 használata). A fenti példában ugyan nem szerepel, de kétirányú **inout** típusú jelek is megadhatók az entitások I/O port listájában.

Architektúra deklarációk

Az architektúra törzse a fenti példában a következő szerint kódrészlet szerint alakult:

```
architecture kapu_szintu of egyenloseg_1_bit is
   -- deklarációs rész: ide kerülhetnek pl. signal, const --
   --kifejezések deklarációi
Begin
   --itt kezdődik az architektúra törzse
   -- Boole kifejezés
   -- de megadhatjuk xnor ekvivalencia operátorként is
   EQ <= (A and B) or (not A and not B);

end kapu_szintu;
```

A fenti architektúra leírásban az áramkör pontos kapu-szintű strukturális felépítése van definiálva (amely lehetne még viselkedési is). Az architektúra törzsének neve `'kapu_szintu'` amely az `'egyenloseg_1_bit'` entitás architektúrális leírását tartalmazza. A VHDL megengedi több architektúra törzs használatát is egyetlen entitáshoz kapcsolódóan, amelynek pontos azonosítását az architektúra egyedi neve biztosítja. Ez a későbbiekben lesz fontos, amikor az alacsonyabb szinten definiált pl. 1-bites összehasonlító áramkört az akár eggyel magasabb hierarchia szinten lévő, például n-bites összehasonlító áramkör felépítésekor kívánjuk felhasználni. Ezt a folyamatot nevezzük példányosításnak.

Az architektúra törzsének deklarációs (bejelentési) része konstansok (const), illetve belső jelek (signal) deklarációját választható módon tartalmazhatja (a fenti esetben nem deklaráltunk ilyeneket). Az architektúra törzs utasításait a `begin` és `end` kulcsszók közötti részeken kell megadni, amelyeken belül minden utasítást egyidejű, párhuzamos vagy más néven konkurens végrehajtásúnak tekintünk. Egy hagyományos pl. ANSI-C nyelvtől eltérően, tehát azok az utasítások, amelyek egy architektúra törzsében definiáltak, azok ténylegesen egyidőben, párhuzamosan hajtódnak végre (kivételt képeznek ez alól az architektúra törzsén belül egy folyamatban, vagy `process`()-ben végrehajtandó szekvenciális utasítások). Ez egyben azt is jelenti, hogy az architektúra törzsében definiált, de bármilyen sorrendben felírt utasítások és hozzárendelések ténylegesen egyidőben fognak végrehajtódni.

A hozzárendelés a fenti egyenletben az EQ értékéhez rendeli a korábbi 2.2.-es igazságtáblázatban definiált (1-bites egyenlőség összehasonlító) áramkör bemeneteit, egyszerű and, or, not operátorok felhasználásával. Azaz, ha a hozzárendelés jobb oldalán lévő A és B bemeneti változók megváltoztatják értékeiket valamely esemény hatására, akkor a megfelelő utasítás (EQ) fog aktiválódni, és az utasítás jobb oldalán álló kifejezés kiértékelődik. Végül a jobb oldali kifejezés új értékét felveszi a baloldalon álló EQ kifejezés, de csak bizonyos véges propagálási (jelterjedési) idő múlva.

Feladat 1/b

Módosítsuk úgy az eredeti (*Feladat 1/a.* szerinti) 1-bites egyenlőség összehasonlító áramkörünk entitását, úgy hogy a hozzá tartozó két különböző (`kapu_szintu`, és `kapu_xnor`) architektúra leírásai a következő szerint legyenek adottak:

```
architecture kapu_szintu of egyenloseg_1_bit is
   -- architektúra #1
begin
   -- Boole kifejezés
   -- de megadhatjuk xnor ekvivalencia operátorként is
   EQ <= (A and B) or (not A and not B);

end kapu_szintu;
```

```
architecture kapu_xnor of egyenloseg_1_bit is
   -- architektúra #2
begin
   -- Boole kifejezés
   -- de megadhatjuk xnor ekvivalencia operátorként is
   EQ <= A xnor B;

end kapu_xnor;
```

Tehát a fenti kód szerint egészítsük ki a korábbi *Feladat 1/a* szerinti VHDL leírásunk architektúra részét (a file neve a régi maradjon: `egyenloseg_1_bit.vhd`)

2.2.3. Típus deklarációk

Amennyiben a VHDL nyelvben az objektumokhoz új típusokat akarunk bevezetni a **type** azonosítót kell használni a következő szerint:

```
Tipus_deklaracio <= type azonosito is tipus_definicio ;
```

Skalár típus deklarációk

Fontos megjegyezni, hogy amennyiben két azonos típus definiciós résszel rendelkező típust hozunk létre, akkor azok közvetlenül nem rendelhetőek egymáshoz, például:

```
type alma is range 0 to 100;
type korte is range 0 to 100;

--érvénytelen hozzárendelés!
alma := korte;
```

Egész (Integer) típus deklarációk

Az egész típus esetén az ábrázolható értékek számának tartománya: –2,147,483,647 -től +2,147,483,647 –ig terjed, amely VHDL esetén kibővíthető. Integer/egész típus definicióját a következőképpen lehet megadni:

```
integer_tipus_definicio <= range kifejezes ( to | downto ) kifejezes
```

ahol a tartomány (**range**) növekedő, ha **'to'** kulcsszóval adjuk meg, illetve csökkenő **'downto'** kulcsszó használata esetén. A következő példa az egész típusok, a későbbi konstansok, illetve változók közös használatát mutatják:

```
constant BITEK_SZAMA : integer := 32;
type bit_index is range 0 to BITEK_SZAMA - 1;

type day_of_month is range 0 to 31;
type year is range 2100 downto 0;
variable today : day_of_month := 4;
variable start_year : year := 2010;
```

Lebegő-pontos (floating-point) típus deklarációk

A VHDL lebegőpontos típusa az IEEE-754 szabványt követi: előjel, exponens, és mantissza részeket definiálva, egyszeres 32-bites, illetve dupla pontosságú 64-bites ábrázolást használva.

```
type bemeneti_ertekhatar is range -10.0 to +10.0;
type valoszinuseg is range 0.0 to 1.0;
…
variable p_50 : valoszinuseg := 0.5;
```

A `real` típus is egy előredefiniált 64-bites lebegőpontos érték, amelytől azonban eltérő lebegőpontos típusokat is definiálhatunk a használat során.

Fizkiai típus deklarációk

A fizikai típus deklarációk segítségével fizikai mennyiségek, mint például hosszúság, tömeg, idő, áram stb. mennyiségek adhatóak meg. Mivel a legtöbb feladatban egyedi késleltetéses módszert használunk, ezért az időbeli (time) fizikai típus deklarációt mutatjuk be a következő példán keresztül:

```
type ido is range --implementáció függő
units
 fs; --elsődleges m.egység
 ps = 1000 fs;
 ns = 1000 ps;
 us = 1000 ns;
 ms = 1000 us;
 sec = 1000 ms;
 min = 60 sec;
 hr = 60 min;
end units;

-- a kesobbiekben, peldaul egy process deklarációs részében valtozokent is
-- hivatkozhatunk a fizikai ido mennyisegre
variable periodus : ido;
```

A **unit** kulcsszó után közvetlenül elhelyezkedő [fs] mennyiséget elsődleges mértékegységnek nevezzük, amely egyben a legkisebb mértékegységet is jelenti az átváltáshoz.

Felsorolt (enumerated) típusok

A felsorolt, vagy 'enumerált' típust más programnyelvekhez hasonló módon, a VHDL nyelv magas absztrakciós szintjén alkalmazva akkor lehet nagyon hasznos, ha az egyes objektumok neveinek értékeket akarunk megfeleltetni (mintegy egyszerű kódolás), anélkül, hogy azokat az értékeket direkt módon hozzárendeltük volna a felsorolt típus elemeihez. A következő az álalános definíciós részt adja meg felsorolt típus esetén

```
enumeralt_tipus_definicio <= ( ( azonosito | karakter_elem ) { , … } )
```

Ha az ALU funkcionalitását kívánjuk felsorolt típusként megadni, akkor a következőt lehet akár leírni:

```
type alu_funkcio is (tilt, atereszt, osszead, kivon, szoroz, oszt);
…
variable alu_op : alu_function := atereszt;
```

A fenti kódrészlet szerint a `tilt = 0`, `atereszt = 1`, `osszead = 2`, `kivon = 3`, `. . . oszt = 5` értéken kerül definiálásra, anélkül hogy direkt módon megadtuk volna.

A felsorolt típusoknak másik fontos alkalmazási területe lehet például az FSM, véges állapotú automata modellek, vagy a szekvenciális hálózatok Mealy, Moore alapmodelljeinek állapot kódolása (lásd későbbi fejezetek).

```
type FSM_state is (start, state_1, state_2, … finish);
…
variable allapot : FSM_state := start;
```

Egy előredefiniált felsorolt típus a severity_level (megszorítási szint), amelyet a szimuláció során különböző visszajelzésekre, figyelmeztetésekre, hibákra stb. lehet használni.

```
type severity_level is (note, warning, error, failure);
```

Altípusok (subtype)

Az altípusok deklarációja során egy bővebb alap típus (base type) teljes értéktartományának bizonyos leszűkítését (szűkebb tartományát) defininálhatjuk, mint például:

```
subtype char is integer range -128 to 127;
subtype word is integer range 31 downto 0;

--a következők VHDL előre definiált altípusai
subtype natural is integer range 0 to highest_integer;
subtype positive is integer range 1 to highest_integer;
```

Attribútumok

Az előre definiált attribútumok csoportjaival a skálár típus/altípus tulajdonságait (sajátosságait) „kérdezhetjük le”, amelyet a típusnév után tett ' jellel adhatunk meg. A következő kódrészlet ezekre mutat néhány példát a korábban tárgyalt fizikai típus deklarációt is felhasználva:

```
type ellenallas is range 0 to 1E9
units
 ohm;
 kohm = 1000 ohm;
 Mohm = 1000 kohm;
end units ellenallas;

type ell_leszukites is range 21 downto 11;

ellenallas'left = 0 ohm    -- első (legbaloldalibb) eleme az ellenallas
-- neven definialt fizikai típusnak
ellenallas'right = 1E9 ohm -- utolsó (legjobboldalibb) eleme…
ellenallas'low = 0 ohm     -- legkisebb értéke
ellenallas'high = 1E9 ohm  -- legnagyobb értéke
ellenallas'ascending = true   -- értéke igaz, hogyha az ellenallas
 -- tartománya (range) növekvő, egyébként hamis
ellenallas'image(2 kohm) = "2000 ohm" -- ellenallas értéke bitfüzérben
-- ("string") kifejezve
ellenallas'value("5 Mohm") = 5_000_000 ohm  -- ellenallas "stringjének"
-- megfelelő fizikai mennyiség, érték

ell_leszukites'left = 21   -- legbaloldalibb érték (itt most a
--legnagyobb!)
ell_leszukites'right = 11  -- legjobboldalibb érték (itt most a
--legkisebb!)
ell_leszukites'low = 11
ell_leszukites'high = 21
ell_leszukites'ascending = false --!!range nem növekedő, hanem csökkenő,
-- ezért hamis
ell_leszukites'image(14) = "14"  -- érték -> string
ell_leszukites'value("20") = 20  -- strin -> érték
```

A következő példa a felsorolt típus attribútumaira mutat néhány példát:

```
type logikai_szint is (ismeretlen, alacsony, magas);

logikai_szint'left = ismeretlen
logikai_szint'right = magas
logikai_szint'low = ismeretlen
logikai_szint'high = magas
logikai_szint'ascending = true
logikai_szint'image(ismeretlen) = " ismeretlen "
logikai_szint'value("alacsony") = alacsony
```

2.2.4. Szabványos logikai adat típusok és operátorok, könyvtári csomagok

A VHDL egy erősen típusos nyelv, amely a hagyományos magas-szintű algoritmus nyelveknél, mint pl ANSI-C is kötöttebb. Ez a tulajdonság azt jelenti egyben, hogy egy objektum típuskonverzió nélkül csak akkor rendelhető egy másik objektumhoz, ha azok adattípusai, így érték-készlete teljesen megegyezik. Habár a VHDL nyelv igen gazdag az előre definiált adat típusokban, azonban jelen jegyzetben – a teljesség igénye nélkül – a teljes készlet csak egy kisebb részhalmazát, legfőképpen az IEEE `std_logic` típust használjuk (VHDL93 szabványból), amely alkalmas szintetizálható (és egyben szimulálható) leírások készítésére.

- **Bit típus:** a hagyományos TTL logikai szintek, ’0’, illetve ’1’ értéket veheti fel
- **Std_logic típus:** ez a típus az std_logic_1164 csomagban van definiálva, és 9 lehetséges értéket vehet fel.
  - A három szintetizálható alapérték:
    - ’0’ – logikai alacsony,
    - ’1’ – logikai magas (ezek a normál TTL szintek, mint bit típusnál), míg
    - ’Z’ – nagy-impedanciás állapot (tri-state), mellett:
  - további 6 érték alkalmazható. Ezekből kettő az
    - ’U’ – inicializálatlan (uninitialized), ill.
    - ’X’ – ismeretlen (unknown) amelyekkel szimuláció során találkozhatunk, például amikor a fix logikai szinteket definiáló jeleket (’0’ – ’1’) direkt módon összekötünk.
  - A maradék négy lehetséges érték a:
    - ’–’ (dont’t care, közömbös),
    - ’H’ (Weak ’1’),
    - ’L’ (Weak ’0’),
    - ’W’ (Weak unknown) a rezolúciós függvények segítségével alkalmazhatóak, ún feloldhatóak.

Az első három alapérték kivételével (’0’,’1’,’Z’), használatukra a jegyzet során nem térünk ki részletesen. Az irodalomjegyzékben szereplő leírásokban [*VHDL87*], [*VHDL93*], [*ASHENDEN*], [*KERESZTES*] ezekről bővebben és részletesen olvashatunk.

- **Std_logic_vector típus:** ez a típus az std_logic típusból definiált tömb, amelynek a bitszélessége változtatható, attól a feladattól függően. Például egy 8-bit szélességű buszvonal esetén, a hagyományos little-endian formátumot követve (azaz MSB:LSB bit sorrendben) egy 'a_little_endian_byte' nevű, bemeneti irányú jelet a következő módon deklarálhatunk (csökkenő sorrendet, azaz 'downto' kifejezést használva):

```
a_little_endian_byte : in std_logic_vector (7 downto 0);
```

Ugyanezt a jelet a hagyományostól eltérő, big-endian formátumban (LSB:MSB) a következő módon vezethetnénk be (növekvő sorrendet, azaz 'to' kifejezést használva):

```
a_big_endian_byte : in std_logic_vector (0 to 7);
```

Természetesen, a hagyományos magas-szintű programnyelvekhez hasonló módon hivatkozhatunk a vektortömb egy szeletére (slice), vagy akár a vektortömb egyetlen elemére, a következőképpen:

```
a_little_endian_byte(5 downto 3);
a_little_endian_byte(6);
```

- **Logikai operátorok:** a hagyományos logikai operátorok (and, or, not, xor, xnor, és nand) a std_logic, illetve az std_logic_vector adattípusokon vannak értelmezve. A bit-szintű logikai műveleteket viszont a std_logic_vector adat típuson belül értelmezzük. A logikai operátorok elsőbbségi, vagy kiértékelési sorrendje (precedenciája) a következő:
  - not: negálás (Inverter)
  - and : logikai ÉS
  - or : logikai VAGY
  - nand : negált logikai ÉS (NEM-ÉS)
  - nor: negált logikai VAGY (NEM-VAGY)
  - xor: kizáró VAGY (Antivalencia)
  - xnor : negált kizáró VAGY (Ekvivalencia)

Természetesen átzárójelezéssel a logikai operátorok kiértékelésének sorrendje itt is megváltoztatható, pl:

```
(A and B) or (not A and not B);
```

2.2.5. Szabványos IEEE std_logic_1164 csomag és operátorai

Mind a logikai operátorok, mind a relációs operátorok, és néhány aritmetikai operátor is a Xilinx XST eszközének segítségével automatikusan szintetizálhatók a HDL leírásokban történő felhasználásuk során. Most ezeknek az operátoroknak megfelelő VHDL konstrukciókat vizsgáljuk meg. Az IEEE std_logic_1164 csomag által támogatott operátorok és

a rendelkezésre álló adattípusaik összefoglaló gyűjteménye látható az *2.4. táblázaton* (VHDL-93 szabvány szerint):

2.4. táblázat: az 'IEEE std_logic_1164' csomag VHDL operátorai és adattípusai

VHDL operátor	jelentés	operandus VHDL adat típusa	eredmény VHDL adat típusa
a ** b	hatványozás	interger (egész)	integer (egész)
a + b	összeadás		
a - b	kivonás		
a * b	szorzás		
a / b	osztás		
a & b	konkatenáció (összefűzés)	1D tömb, illetve tömbelemek	1D tömb
a = b	egyenlő	tetszőleges	boolean
a /= b	nem egyenlő		
a < b	kisebb, mint	skalár, 1D tömb	boolean
a <= b	kisebb egyenlő, mint		
a > b	nagyobb, mint		
a >= b	nagyobb egyenlő, mint		
not a	negálás	boolean, std_logic, std_logic_vector	boolean, std_logic, std_logic_vector
a and b	ÉS		
a or b	VAGY		
a nand b	NEM-ÉS		
a nor b	NEM-VAGY		
a xor b	Kizáró VAGY, Antivalencia		
a xnor b	Ekvivalencia		
abs	abszolútérték	integer, fizikai	integer fizikai
mod	maradékos osztás (modulo)		
rem	maradék (remainder)		
sll	balra léptetés (logikai)	1D tömb, illetve tömbelemek	1D tömb, illetve tömbelemek
srl	jobbra léptetés (logikai)		
sla	balra léptetés (arit)		
sra	jobbra léptetés (arit)		
rol	forgatás (rotate) balra		
ror	forgatás (rotate) jobbra		

A VHDL szabvány szerint definiált fenti relációs operátorok az operandusokat összehasonlítják és az eredményük `boolean` (igaz, hamis) típusú lesz. Az `integer` és `natural` beépített VHDL adattípusokon értelmezett aritmetikai operátorok eredménye `integer` lesz. Amennyiben az adott fejlesztési feladatban tetszőlegesen konfigurálható módon szeretnénk megadni az adatok bitszélességét, formátumát (előjeles 2's komplemens `signed`, vagy előjel nélküli

unsigned típusokkal), akkor a IEEE_std_logic_1164 csomag helyett általában az IEEE numeric_std csomag a használatos. Az integer és natural adattípusokat általában az összetett elemi típusok, a tömb határok megadásánál, illetve konstans értékek definiálásánál is alkalmazzuk.

Összefűzés (concatenation)

Az összefűzés '&' operátora alkalmas arra, hogy egy 1D tömb elemeiből, vagy 1D résztömbökből nagyobb tömböket hozzunk létre. A következő példák néhány lehetőséget mutatnak be:

```
library ieee;
use ieee.std_logic_1164.all;

signal a_in_1bit : std_logic;
signal a_in_4bit : std_logic_vector(3 downto 0);
signal b_in_8bit, c_in_8bit, d_in_8bit : std_logic_vector (7 downto 0);

--két nibble (4-bites adat) összefűzése
b_in_8bit <= a_in_4bit  & a_in_4bit;
--2 felső MSB bit '0'-al történő kitöltése, padding, majd összefüzése
c_in_8bit <= "00" & a_in_1bit  & a_in_1bit  & a_in_4bit;
-- két 8 bites adattípus alsó 4 LSB bitjének összefűzése
d_in_8bit <= b_in_8bit(3 downto 0) & c_in_8bit(3 downto 0);
```

Léptető (shift), forgató (rotate) operátorok

A léptetés (shift), illetve forgatás (rotate) logikai és aritmetikai operátorai alkalmasak arra, hogy egy 1D tömb elemeit, részeit tetszőleges módon változtassuk meg adott irányú és hosszúságú léptetéssel, illetve forgatással. A következő példa néhány lehetőséget mutat be használatukra. Tekintsük az IEEE numeric_std csomagot, és adattípusokat:

```
library ieee;
use ieee.numeric_std.all;
--…
signal a_in_8bit : unsigned(7 downto 0);

-- a_in_8bit forgatása az alsó 3 bitjével jobbra
a_in_8bit  ror 3;
-- a_in_8bit logikai léptetése 2-pozícióval balra
a_in_8bit  sll 2;
-- a_in_8bit aritmetikai léptetése 1-pozícióval jobbra
a_in_8bit  sra 1;
```

Ugyanezek az operátorok az IEEE szabvány szerinti std_logic_1164 könyvtári csomagon értelmezett std_logic_vector típusok segítségével is megadhatóak:

```
library ieee;
use ieee.std_logic_1164.all;
--…
signal a_in_8bit : std_logic_vector(7 downto 0);
signal          rotate_right,shift_left_logic,shift_right_arith          :
std_logic_vector(7 downto 0);

-- a_in_8bit forgatása az alsó 3 bitjével jobbra
rotate_right <= a_in_8bit(2 downto 0) & a_in_8bit(7 downto 3); --ror
-- a_in_8bit logikai léptetése 2-bittel balra, úgy hogy közben az alsó 2
-- LSB bitpozícióba ’0’-ákat léptetünk be
shift_left_logic <=  a_in_8bit(7 downto 2) & "00";  --sll
-- a_in_8bit aritmetikai léptetése 1-pozícióval jobbra, úgy hogy közben
-- a felső bitpozícióba az a_in_8bit típus MSB bitjét léptetjük be
shift_right_arith <= a_in_8bit(7) & a (7 downto 1) ;--sra
```

2.2.6. Szabványos IEEE numeric_standard csomag és operátorai

Ez a később megjelent IEEE `numeric_standard` csomag a korábban tárgyalt okokból a std_logic típusnak a kiterjesztését jelentette: új adattpusok (pl. `signed`, `unsigned`) bevezetésével, és az operátorok felüldefiniálása (overloading) segítségével. Tulajdonképpen a `signed`, és `unsigned` adattípusok az std_logic típusú elemek tömbjeként határozhatók meg (bináris reprezentáció). Ez van röviden összefoglalva az *2.5. táblázatban* (VHDL-93 szabvány szerint):

2.5. táblázat: az IEEE numeric_std csomag szintetizálható VHDL operátorai és adattípusai

VHDL operátor	jelentés	operandus VHDL adattípusa	eredmény VHDL adattípusa
a + b	összeadás	unsigned, natural signed, integer	unsigned, signed
a - b	kivonás		
a * b	szorzás		
a = b	egyenlő	unsigned, natural signed, integer	unsigned, signed
a /= b	nem egyenlő		
a < b	kisebb, mint		
a <= b	kisebb egyenlő, mint		
a > b	nagyobb, mint		
a >= b	nagyobb egyenlő, mint		

A forráskódba az IEEE csomagok használatához a következő sorokat kell beilleszteni:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

A fenti aritmetikai műveletek közül a szorzás a legösszetettebb művelet, amelynek szintézise erősen függ a kiválasztott szintézis szoftvertől (pl. XST), illetve az eszköz-specifikus FPGA technológiától. Például a Xilinx Spartan-3E sorozat előjeles, 18x18-bites szorzó blokkokat tartalmaz, míg más eszközöknél (pl. Virtex-5, -6 sorozat) nagyobb, 25x18-bites szorzó-összeadó makrocellák is helyet kaptak. Mint látható, az ieee.numeric_std típusban sincs az osztás / művelete definiálva: ehhez a műveleti osztályhoz egy Xilinx CoreGenerator nevű program segítségével generálhatunk VHDL leírást, amelyhez a fordító a logikai cellák mellett, makró cellákat is lefoglal, amelynek azonban így nagy lesz az erőforrás szükséglete. Ha a feladatban lehetséges, éppen ezért a 2-hatványával való osztás (egyben szorzás) esetén érdemes a léptető (shift) operátorokkal helyettesíteni az osztást (szorzást).

Explicit típus konverzió

Mivel a VHDL erősen típusos nyelv, ezért a konverziós függvények segítségével a fenti IEEE `std_logic_1164`, illetve `IEEE numeric_std` adat típusokon explicit típus konverziót (type casting) lehet végrehajtani azért, hogy a típusokat egyeztessük egy kifejezésben, vagy hozzárendelésben.

2.6. táblázat: Konverzió std_logic_1164 illetve numeric_std csomagokban lévő adattípusok között

VHDL adattípus (x)	Eredmény VHDL adattípusa	konverziós VHDL függvény (type cast)
unsigned, signed	std_logic_vector	std_logic_vector(x)
signed, std_logic_vector	unsigned	unsigned(x)
unsigned, std_logic_vector	signed	signed(x)
unsigned, signed	integer	to_ingeger(x)
natural	unsigned	to_unsigned(x,size)
integer	signed	to_signed(x, size)

A következő néhány példa a fenti adattípusokon történő konverziók használatát, tipikus hibákat, illetve azok kiküszöbölését szemléltetik, megfelelő szabványos csomagok használatával:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
…
signal sx1, sx2, sx3, sx4, sx5, sx6: std_logic_vector(3 downto 0);
-- sx, mint signed std_logic_vector típus
signal ux1, ux2, ux3, ux4, ux5, ux6: unsigned(3 downto 0);
-- ux, mint unsigned numeric típus

--…
-- vizsgálat #1
ux1 <= sx1;       -- hibás típusegyeztetés!
ux2 <= 5;         -- hibás típusegyeztetés!
sx2 <= ux3;       -- hibás típusegyeztetés!
sx3 <= 5;         -- hibás típusegyeztetés!

-- a fentiek javítása explicit korrekció megadásával
uxl <= unsigned(sx1);      --ok, típusmódosítás
ux2 <= to_unsigned( 5 , 4 );     -- ok, konverziós fgv. használata
sx2 <= std_logic_vector(ux3);    -- ok, típusmódosítás
sx3 <= std_logic_vector(to_unsigned( 5 , 4 ) ) ;  -- ok, típusmódosító ---
és konverziós függvény egymásba ágyazása és kombinált használata


--vizsgálat #2
-- következő hozzárendelések helyesek, a numeric_std csomagra
ux4 <= ux2 + ux1;  --ok, mindkét operandus és eredmény is unsigned
ux5 <= ux2 + 1;    --ok , ux2 operandus unsigned, illetve 1 natural


--vizsgálat #3
sx5 <= sx2 + sxl;  --hiba! „+” operátor nem definiált az std_logic_vector
-- adattípusra
sx6 <= sx2 + 1;    --hiba! „+” operátora definiálatlan az
--std_logic_vector adattípusra
```

```
-- a fentiek javítása explicit korrekció megadásával
-- két lépésben adottak:
- először konverziós függvény használata numeric_std csomagra az összeadás elvégzéséhez
-- másodszor a std_logic_vector típuskonverzió használata visszaalakításhoz, és a hozzárendelés bal oldalával történő megfeleltetéshez
sx5 <= std_logic_vector(unsigned(sx2) + unsigned(sx1));    -- ok
sx6 <= std_logic_vector(unsigned(sx2) + 1) ;          -- ok
```

Megjegyzés: A fenti példák csupán néhány tipikus problémra és megoldásukra világítanak rá, természetesen ezeknél sokkal bővebb leírási módra is lehetőség van a VHDL nyelvben. A típusokról, típuskonverziókról további részleteket a megfelelő irodalmi hivatkozásokon keresztül találhatunk [*VHDL87*], [*VHDL93*], [*ASHENDEN*].

2.2.7. Nem szabványos IEEE könyvtári csomagok

Az utóbbi évtizedben számos, nem-IEEE szabvány szerinti kiegészítés, könyvtári csomag is napvilágot látott, amelyekre már hivatkozni lehet a VHDL leírásokban. Ilyenek például az `std_logic_arith`, `std_logic_unsigned`, illetve `std_logic_signed` csomagok. Ez utóbbi kettő az IEEE `std_logic` kibővítését jelentik, amelyek segítségével az `std_logic_vector` adattípusokon az aritmetikai operátorok végrehajtása is engedélyezett, és nem szükséges explicit típuskonverzió sem. Azonban ezek használata nem javasolt, mivel nem a szabványos IEEE csomagok részét képezik, így jelen jegyzetben sem használjuk fel őket, csupán említést teszünk róluk. További részleteket az irodalmi hivatkozásokban olvashatunk [*VHDL93*], [*ASHENDEN*].

2.2.8. Konstansok, változók, jelek, és generic használata

Konstansok

A VHDL kód sok helyen tartalmazhat konstans kifejezéseket és akár tömb határolókat. Egy jó tervezési koncepció lehet – hasonlóan más programnyelvekhez – ha a sokszor használt konstans kifejezéseket szimbolikus névvel definiáljuk, és ott adunk hozzá értéket is. A definiált konstansok az előfeldolgozás során értékelődnek ki és helyettesítődnek, ezáltal nem igényelnek fizikai erőforrásokat. A konstansok deklarációját az entitáshoz tartozó architektúra leírás deklarációs részében kell megadni, még az első használat előtt, amelynek szintaxisa a következő**:**

```
constant konstans_neve : adattípus : = kezdőérték;
```

Példaként a következő konstans deklarációk adottak, amelyeket tömbök határainak, vagy logikai vektor kifejezések bitszélességének megadásákor lehet felhasználni:

```
constant ADAT_BIT: integer : = 8;
```

```
constant ADAT_TARTOMANY: integer : = 2** ADAT_BIT - 1;

-- még néhány példa a használatukra: egesz, valós és fizikai típus
-- deklarációkra
constant byteok_szama : integer := 4;
constant bitek_szama : integer := 8* number_of_bytes;
constant e : real := 2.718281828;
constant jelterjedesi_kesleltetes : time := 3 ns;
constant size_limit, count_limit : integer := 255;
```

Feladat 2/a – konstansok használata

Definiáljunk konstansokat egy 4-bites összeadó (neve `osszeado_4_bit`) áramkörben a bitszélesség megadásához, amelyben a numberic_std csomagot használja az '+' operátorához. A bemeneti operandusok (a_in, b_in) legyenek 4-bit szószélességű std_logic_vector típusúak, a kimeneti eredmény legyen 4-bit szélességű std_logic_vector típusú, és az átvitel (c_out – carry out) 1-bites std_logic típusú. Mivel az '+' összeadás operátora a numeric_std csomagon értelmezett unsigned adattípusra érvényes, ezért a belső jelek, és a külső portok jelei között explicit típus konverzió is szükséges. Az ADAT_BIT nevű szimbolikus konstans szolgál arra, hogy az architektúra belső jeleit 4+1-bites szószélességen definiáljuk (előjel kiterjesztés végett a +1 bit). Egy fontos szabály, hogy két pl. N-bites szám összege helyesen egy N+1 bites eredményként tárolható csak el.

```
-- 4-bites összeadó
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;  -- '+' operátorához

entity osszeado_4_bit is
   port(
      a_in, b_in: in std_logic_vector(3 downto 0);
      c_out: out std_logic;
      sum_out: out std_logic_vector(3 downto 0)
   );
end osszeado_4_bit;

architecture behav of osszeado_4_bit is
   constant ADAT_BIT: integer := 4;   --4-bites konstans kifejezés
   --belső jelek
   signal a_sig, b_sig, sum_sig: unsigned(ADAT_BIT downto 0);
begin
```

```
   a_sig <= unsigned('0' & a_in);  --kiterjesztes elojel nelkul
   b_sig <= unsigned('0' & b_in);
   sum_sig <= a_sig + b_sig;     --osszeadas unsigned jeleken
   -- eredmeny visszalakitasa (ADAT_BIT) szélességűre
   sum_out <= std_logic_vector(sum_sig(ADAT_BIT-1 downto 0));
   --eredmeny MSB bitje lesz a generalt carry out, atvitel
   c_out <= sum_sig(ADAT_BIT);
end behav;
```

Változók (variables)

A VHDL kód sok helyen tartalmazhat változó (variable) objektumokat, amelyeket első használatuk előtt mindenképpen deklarálni kell. A változó azonban a kezdeti értékadás után (szemben a jellel) bármikor megváltoztathatja értékét. A változókat folyamatban vagy alprogramban adatok ideiglenes tárolására használják A változók deklarációját az entitáshoz tartozó architektúra leírás szekvenciális folyamatának (process) deklarációs részében (esetleg a nem tárgyalt eljárásban – procedure) lehet megadni, amelynek szintaxisa a következő**:**

```
variable valtozo_neve : adattípus : = kezdőérték;
```

Ha a változónak kezdeti értéket adunk, azonnal kiértékelődik. Ha kezdeti értéket nem adunk meg, akkor a változó az adott típus legkisebb értéke lesz (növekvő sorrendbe rendezett tartományban), illetve legnagyobb eleme lesz (csökkenő sorrendű tartományban). Példaként a következő néhány változó deklarációk adottak:

```
variable index : natural := 0;
variable osszeg, atlag : real := 0.0;
variable start, befejezes : time := 0 ns;
```

Változók esetén is – hasonlóan a konstans kifejezésekhez, és a generichez – az értékadás operátora ':='. A változók a folyamat egy állapotát reprezentálják.

Következő kódrészlet a változók használatára mutat be egy rövid példát:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity valtozo is
end entity valtozo;

architecture behav of valtozo is
  constant incr : integers := 1;
```

```
begin
process is
  -- ide kerülhetnek a lokális változó deklarációk
  variable sum : integer := 0;
  begin
    sum := sum + incr;
  end process;
end architecture behav;
```

Jelek (signals):

A jelek működésének megértéséhez a következő fontos alapfogalmakat kell tisztázni: esemény, meghajtó illetve lebonyolítás, vagy tranzakció.

Amikor egy jel értéke megváltozik, akkor egy ún. **esemény (event)** történik a jelen. Amikor egy értéknek egy jelhez egy bizonyos késleltetéssel (pl. after klauzula) való hozzárendelése időzítésre került, azt **a jel meghajtóján (driver)** elkezdődött **lebonyolításnak (transaction)** nevezik. Az is lebonyolítás, amely nem változtatja meg a jel értékét, ezért nem okoz eseményt a jelen. Új lebonyolítás nem kezdődik el a jelen, amíg az előző nem fejeződött be.

Az információ átadás szempontjából a legfőbb eltérés a jelek és a változók között, hogy a változók nem vihetnek át információt azon VHDL blokkon kívülre, amelyen belül deklarációjuk történt. Éppen ezért jelen jegyzetben kerüljük a változók, és a megosztott változók használatát. Ezekről részletesen más könyvekben, segédletekben [*ASHENDEN*], [*KERESZTES*] lehet olvasni. A jeleket az entitáshoz tartozó architektúra leírások deklarációs részén kell bevezetni, ezáltal a jelek nemcsak az egyidejű hozzárendelésekben, hanem a sorrendi hozzárendelésekben (process-en belül) is használhatóak. Másik fontos tulajdonság, hogy a jel nem azonnal értékelődik ki, hanem csak bizonyos (delta) késleltetéssel később vehet fel új értéket, míg a változó azonnal megkapja az értékét (sőt mint láttuk értéket is tárol egyben). Jelek esetén a hozzárendelést a '<=' szimbólummal adhatjuk meg, a jel nem tárol értéket, csak késleltet(het)i azt. A jel és a változó feldolgozásban további különbség, hogy a szekvenciális végrehajtású VHDL kódrészeken a feldolgozás sorról-sorra halad, ezért nevezik sorrendi VHDL-nek, míg az egyidejű/konkurens VHDL kódrészeken a sorokat csak akkor dolgozzák fel, ha egy esemény megjelent a jelek érzékenységi listáján. A jelekhez új értéket egyedi késleletetéses módon is adhatunk (pl. after kulcsszóval):

```
signal y : std_logic := '0';
. . .
-- egyedi késleltetés nélkül
y <= not a;
. . .
-- egyedi késleltetéssel
y <= not a after 5 ns;
```

Ebben a VHDL leírásban az `y` a hozzárendelés végrehajtásától számított `5 ns` múlva veszi fel új értékét (`not a`), az after klauzula használatával. A késleltetés egyrészt tekinthető **szintézis** jelleggel a véges jelterjedési idő értékeként, azaz amikor a kimenet a bemenet változásának hatására `5 ns` késleltetéssel veszi fel új értékét. Másrészt vizsgálhatjuk **szimulációs** értelemben is, amikor a PC-n futó szimulátor belső, ciklus-pontos órájához képest felvett időzítést adjuk meg. A jelek és változók összehasonlításáról a későbbi *2.3.2. alfejezetben* még részletesen olvashatunk, a folyamatok (process-ek) végrehajtása során.

Generikusok

Az előző Feladat 2/a akkor lenne tetszőlegesen konfigurálható, amennyiben az entitások portlistájában lévő bitszélességeket is konstansként lehetne definiálni. Erre az ’általánosításra’ a VHDL-ben a generikusok, azaz „generic” kulcsszó alkalmazásával van lehetőség, amely egyrészt entitások portlistájában szereplő port-oknak, másrészt az architektúra leírásokban példányosítandó komponenseknek is adhat át értéket. A generikusokat az entitás deklarációs részén lehet módosítani, vagy akár egy külső állományban (pl. egy felhasználó által készített egyedi csomagban) lehet megadni, amelyet meghívunk az adott entitásban. Ezáltal sokkal jobban modularizálható, skálázható VHDL leírásokat kaphatunk. Fontos megjegyezni azonban, hogy a ’generic’ új értéket az architektúra leírás törzsén belül már nem kaphat. A generic deklarációja a következő:

```
entity entitas_nev is
generic (
 generic_nev : adat_tipus : = alap_ertek(ek);
 generic_nev : adat_tipus : = alap_ertek(ek);
 . . .
 generic_nev : adat_tipus : = alap_ertek(ek)
)
port (
 port_nev : mod adat_tipus;
 . . .
);
end entitas_nev;
```

Feladat 2/b – Generikusok használata

Az előző Feladat 2/a (ADAT_BIT = 4-bites összeadó áramkör) módosítása a ’generic’ kulcsszó használatával a következő (a generic-el kibővített forrás neve legyen `osszeado_4_bit_gen`):

```
-- 4-bites összeadó generic-el
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;  -- ’+’ operátorához
```

```
entity osszeado_4_bit_gen is
   generic (ADAT_BIT : integer := 4);  -- generic megadása 4 adat-bitre
   port(
      a_in, b_in: in std_logic_vector(ADAT_BIT-1 downto 0);
      c_out: out std_logic;
      sum_out: out std_logic_vector(ADAT_BIT-1 downto 0)
   );
end osszeado_4_bit_gen;

architecture behav of osszeado_4_bit_gen is
   --belső jelek
   signal a_sig, b_sig, sum_sig: unsigned(ADAT_BIT downto 0);
begin
   a_sig <= unsigned('0' & a_in);   --kiterjesztes elojel nelkul
   b_sig <= unsigned('0' & b_in);
   sum_sig <= a_sig + b_sig;     --osszeadas unsigned jeleken
   -- eredmeny visszalakitasa (ADAT_BIT) szélességűre
   sum_out <= std_logic_vector(sum_sig(ADAT_BIT-1 downto 0));
   --eredmeny MSB bitje lesz a „generalt” carry out, atvitel
   c_out <= sum_sig(ADAT_BIT);
end behav;
```

Ha a fenti példában szereplő `ADAT_BIT` szélességű összeadót eggyel magasabb hierarchia szinten lévő entitásban komponensként kívánjuk példányosítani, akkor a kívánt 'generic' értéket (itt bitszélességet) a példányosításnál meg kell adnunk. Ha a példányosításnál nem adunk meg, vagy elfelejtünk megadni generic értéket, akkor a bitszélességhez azt a kezdő értéket rendeli a fordító, amely az alacsonyabb hierarchia szinten lévő entitásban lett deklarálva. A következő kódrészlet a korábbi `osszeado_4_bit_gen` összeadó áramkör néhány különböző tetszőlegesen definiált bitszélesség szerinti példányosítását szemlélteti (ahol ADAT_BIT=8-, 16-, valamint 4-bites alapértéket beállítva):

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
. . .
--bemeneti jelek
signal a_4, b_4, osszeg_4: unsigned(3 downto 0);
signal a_8, b_8, osszeg_8: unsigned(7 downto 0);
signal a_16, b_16, osszeg_l6: unsigned(l5 downto 0);
signal carry_4, carry_8, carry_16: std_logic;
```

```
-- osszeado peldanyosítasa 8-bitesként
osszeado_8: work.osszeado_4_bit_gen(behav)
generic map (ADAT_BIT => 8)
port map(a_in=>a_8, b_in=>b_8, c_out=>carray_8, sum_out=>osszeg_8)) ;

-- osszeado peldanyosítasa 16-bitesként
osszeado_16: work.osszeado_4_bit_gen(behav)
generic map (ADAT_BIT => 16)
port map(a_in=>a_l6, b_in=>b_16, c_out=>carry_16 , sum_out=>osszeg_l6)) ;

-- osszeado peldanyosítasa alapértéken, 4-bitesként
osszeado_4: work.osszeado_4_bit_gen(behav)
-- itt nem adjuk meg a generic map-el a bitszelesseget!! -> alapertek
port map(a_in=>a_4, b=>b_in_4, c_out=>carry_4, sum_out=>osszeg_4)) ;
```

A fenti esetekben a példányosítás során a **generic map** kulcsszót kell használnunk, ha a genericnek egy új, tetszőleges kezdőértéket kívánunk megadni.

2.2.9. Generáló struktúrák – Generate utasítás

A ’generate’ egy olyan speciális iteratív utasítás, amely a digitális áramkörök egyes alrendszereiben lévő komponensek szabványos, iteratív összeállítását biztosítja. A „generate” utasítás egy konkurrens utasításokat konkurrens blokká szervező „for” ciklusnak felel meg. Tipikus használatuk például a memóriák, regiszterek tömbjeinek felépítésében van, mivel sok, de hasonló elemből álló, nagy-méretű reguláris tömböt kell megadni. A generate utasításnak több lehetéges változata van:

- **For – generate:** ismétlődő komponens generálás,
- **If – generate** : feltételes komponens generálás,
- **Case – generate**: feltételes komponenes generálás.

A generáló struktúrákról részletesen a [*KERESZTES*], [*ASHENDEN*] referenciákban olvashatunk.

2.3. VHDL – konkurens és szekvenciális hozzárendelési utasítások

Ebben a fejezetben a VHDL nyelvben használatos legfontosabb kifejezések, nyelvi konstrukciók, szekvenciális illetve konkurens (egyidejű) utasítások, valamint vezérlésátadó szerkezetek kerülnek ismertetésre. Minden esetben egy-egy példán keresztül szemléltetjük az utasítások használatát.

2.3.1. Egyidejű (konkurens) hozzárendelési utasítások

Éppen ebben, a konkurens, azaz ténylegesen párhuzamosított utasítás végrehajtásban rejlik az újrakonfigurálható FPGA áramkörök és a HDL nyelvek nagy előnye. Szemben a hagyományos nyelvi leírásokkal, amelyek a sorrendi, azaz egymás utáni lépések sorozatában hajtják végre az utasításokat (pl. ha egy egymagos számítógép architektúrán egy-szálú programfuttatást tekintünk), addig az FPGA-kon ténylegesen egyidejűleg, azaz nagyfokú párhuzamosítással, és utasítás feldolgozással történhet a műveletek végrehajtása. Mindezt ráadásul órajelciklus pontossággal követhetjük nyomon szimuláció során. Ez az ún. „masszívan párhuzamosított” feldolgozás, amely az FPGA-k architekturális felépítésének köszönhető: itt gondoljunk arra, hogy nagyszámú programozható logikai, és dedikált makrocellás erőforrások reguláris 2D-tömbjeiből épülnek fel. Az egyidejű, konkurens utasításoknak a VHDL szintézis szempontjából vizsgálva több lehetséges típusa van:

- **Ténylegesen egyidejű hozzárendelési utasítás (feltétel és kiválasztás nélküli)**
- **Egyidejű feltételes hozzárendelés (when-else szerkezet)**
- **Egyidejű kiválasztó hozzárendelés (with-select szerkezet)**

Egyidejű jelhozzárendelés

Ha nem adunk meg feltételeket és kiválasztó jeleket, akkor alapvetően egy entitás architektúrális részében lévő hozzárendelő (<=) utasításokat konkurens, azaz ténylegesen egyidejű jel hozzárendelési utasításoknak nevezzük: a hozzárendelés jobb oldalán lévő kifejezés értékét a bal oldalon álló objektum (pl. signal) fogja felvenni. A konkurens jelhozzárendelési utasításokat az architektúra `Begin…End` közötti részén, de a szekvenciális utasítás blokkokon (pl. `process()` lásd később) kívüli hatókörön kell deklarálni. A jelhozzárendelési utasításokra láthatunk egy rövid példát:

```
library IEEE;
use IEEE.std_logic_1164.all

entity pelda is
--portlista ...
end pelda;
```

```
architecture behav of pelda is
 signal nibble : std_logic_vector(0 to 3);
 signal concatenation : std_logic_vector (7 downto 0);
 signal switch_sig : std_logic;
 signal sw : std_logic_vector(7 downto 0);
 signal low_sig : std_logic;

begin
--ezek az egyidejű jelhozzárendelési utasítások ténylegesen egyidőben,
-- párhuzamosan hajtódnak végre.
 nibble     <= "1000";
 concatenation <= '1' & sw(1) & sw(0) & "10101";
 switch_sig    <=    sw(7);
 low_sig    <= '0';
-- . . .
end behav;
```

Ha ugyanazon jelhez többszörös, egyidejű hozzárendelési utasítással adunk értéket, akkor mindenegyes konkurens utasítás kiértékelődik, és mint párhuzamos ágak a kimeneti jelhez egyszerre fognak kapcsolódni (tie). Azonban a kimeneti értékek ily módon történő huzalozott összekötése a legtöbb technológiai leírás szerint nem megengedett művelet, illetve explicit módon kell kezelni (a kimeneti jelnek több forrása, ún. drivere lehet), különben a szintézis során hibát kapunk. Ilyen esetekben alkalmazhatók az ún. rezolúciós, feloldó függvények, amelyekről részletesen a [*BME*], [*KERESZTES*], [*ASHENDEN*] referenciákban olvashatunk.

A konkurens utasításoknak további két fő csoportját különböztetjük meg: feltételes-, illetve kiválasztáson alapuló jelhozzárendelési utasításokat. A működésüket tekintve a hagyományos szekvenciális ’if-else’ és ’case’ szerkezetekhez hasonlítanak, azzal a lényeges eltéréssel, hogy ezek a konkurens utasítások (végrehajtási ágakat) a szintézis során a programozható összeköttetés hálózat (routing network) segítségével párhuzamosítva képeződnek le az FPGA áramkörre.

Egyidejű feltételes jelhozzárendelés (when-else)

A feltételes jelhozzárendelés (**when-else szerkezet**) egyszerűsített VHDL szintaxisa a következő:

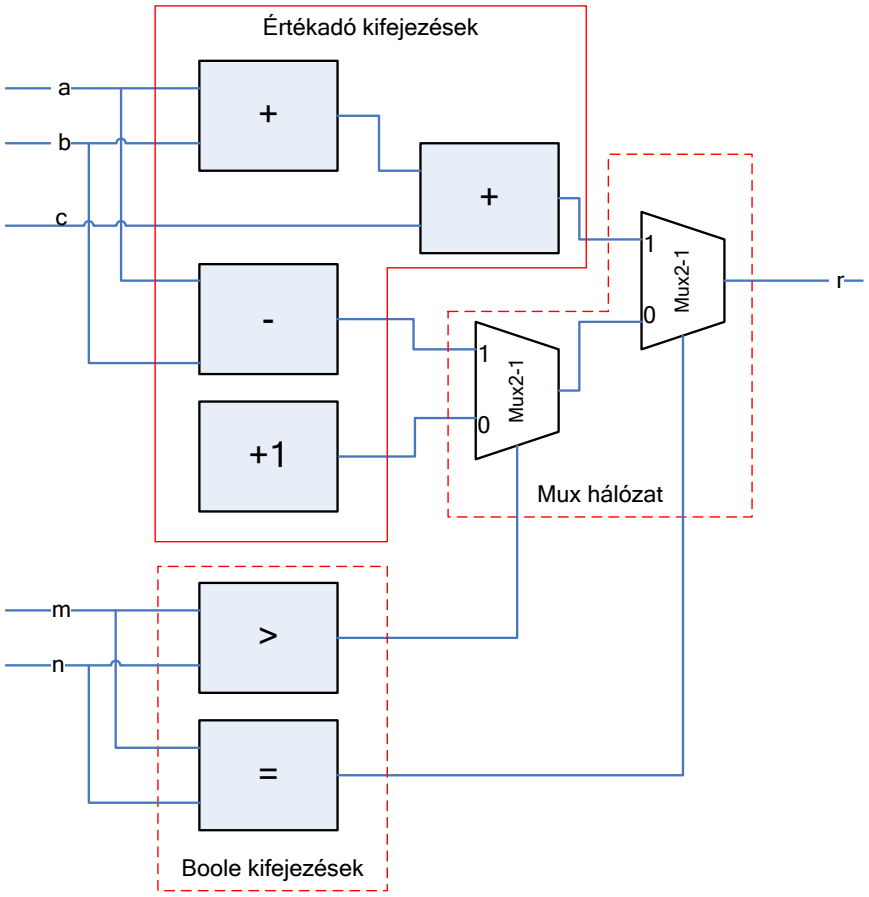
```
jel_neve <= ertekado_kifejezes_1 when boolean_kifejezes_1 else
            ertekado_kifejezes_2 when boolean_kifejezes_2 else
            . . .
            ertekado_kifejezes_n;
```

Minden egyes when-else feltételes ágban szereplő boolean kifejezés egymást követően értékelődik ki (tekinthetnénk prioritásos megvalósításúnak is), mindaddig, amíg valamelyik feltétel igazzá nem válik. Ezekből a when-else szerkezetekből felépülő jelhozzárendeléseket (<=) akár egyidejűleg el tudjuk végezni.

Nézzünk erre egy konkrét példát is. A következő egyidejű feltételes jelhozzárendelés egy többszintű, útvonalválasztó (logikai) hálózatot fog szintetizálni, amelynek elvi megvalósítása a *2.6. ábrán* látható:

```
rout <= a + b + c when m = n else
        a - b     when m > n else
        c + 1;
```

Az útvonalválasztást 2-1 Multiplexerek sorozata végzi, amelynek működését a következő módon lehet definiálni: a multiplexerek lehetséges 2^n bemenete közül az n darab kiválasztó (selector) jel segítségével adjuk meg, hogy pontosan melyik bemenetet kössük (route) össze a kimenettel. Az egyidejű feltételes hozzárendelés elvi kapcsolási rajza pedig a *2.6. ábrán* adott. Ha a kiértékelés sorrendjében az első boolean kifejezés (`m=n`) értéke igaz, akkor az eredményként `a+b+c` kerül a kimenetre, ha nem, akkor a következőt értékeljük ki, azaz `a-b` kerülhet a kimenetre. Ha az `m > n` (és korábban az `m = n`) értéke is hamis, akkor az utolsó ág, `c+1` kerül az `rout` kimenetre.



2.6. ábra: Az egyidejű feltételes hozzárendelési utasításnak megfelelő elvi áramköri modell

Megjegyzés: mivel a Boole kifejezések és értékadó utasítások konkurens módon értékelődnek ki, ahogy a kiválasztáshoz használt Boole kifejezések számát növeljük, úgy növekszik a multiplexer-szintek száma is. Így egy komplexebb when-else szerkezetből hosszabb multiplexer lánc szintetizálódik, amely viszont megnövelheti a teljes áramkör jelterjedési időszükségletét.

Feladat 1

Nézzünk egy konkrét példát when-else egyidejű hozzárendelési utasításra. Tervezzünk egy prioritásos kódoló (priority encoder) áramkört, amelynek a 4-bites bemenete az igényeket, kéréseket definiáló bitmintázatot adja meg, legyen a neve `req(3:0)`. A legnagyobb prioritással az MSB bit, azaz a `req(3)` bír. A prioritásos kódoló `pcode_out(2:0)` kimenete pedig egy olyan bináris kód, amely mindig a beérkezett kérések közül a legnagyobb prioritású kérésnek felel meg. Az áramköri entitás neve legyen „`priencoder_4_conc`”, míg a hozzá tartozó architektúra neve legyen „`Behavioral`” (azaz viselkedési, hiszen az áramkörnek a viselkedését a VHDL leírásban egyidejű feltételes utasítással adjuk meg).

Ennek a prioritásos kódolónak a működése adott a következő igazságtáblázattal:

bemenetek				prioritásos kódolt kimenet
req(3)	**req(2)**	**req(1)**	**req(0)**	**pcode_out(2:0)**
1	-	-	-	100
0	**1**	-	-	011
0	0	**1**	-	010
0	0	0	**1**	001
0	0	0	0	000

A fenti igazságtáblának megfelelő 4-bemenetű prioritásos kódoló áramkör VHDL leírása a következő:

```
-- Feladat. prioritásos kódoló áramkör when-else egyidejű
-- feltételes hozzárendeléssel
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity priencoder_4_conc is
Port (    req : in  STD_LOGIC_VECTOR (3 downto 0);
    pcode_out : out  STD_LOGIC_VECTOR (2 downto 0));
end priencoder_4_conc;

architecture Behavioral of priencoder_4_conc is
```



```
begin
pcode_out  <=  "100" when  (req(3)='1') else
       "011" when  (req(2)='1') else
       "010" when  (req(1)='1') else
       "001" when  (req(0)='1') else
       "000";
end Behavioral;
```

A fenti kódban először az `req(3)` bemeneten keresztül érkezett kérést vizsgáljuk meg, amely ha ’1’-es volt a `pcode_out(2:0) = "100"` -ra állítódik be. Egyébként pedig, mindig az egyre alacsonyabb prioritású igényeket vizsgáljuk sorban egymás után (`req(2), req(1),` végül `req(0)`).

Feladat 2

A következő példa egy n→2^n bináris dekódoló áramkör egyidejű feltételes utasításokkal történő viselkedési leírását adja meg VHDL-ben. Tekintsünk egy 2→4-es dekóder áramkört, amely `en_in` magas-aktív engedélyező bemenettel is rendelkezik, illetve ún. „one-hot”, azaz 1-es súlyú bináris dekódolást használ: a `decode_out` kimeneten mindig az adott 2^n bitpozícióban ’1’-es állítódik be a bemeneti kombinációnak megfelelően. A 2→4-es bináris dekódoló áramkör (neve legyen `decoder_2_4_conc`) működését leíró igazságtáblázat a következő:

bemenetek			dekódolt kimenet
en_in	**a_in(1)**	**a_in(0)**	**decode_out(3:0)**
0	-	-	0000
1	0	0	000**1**
1	0	1	00**1**0
1	1	0	0**1**00
1	1	1	**1**000

Az igazságtáblázat szerint a 2→4-es dekódoló áramkörnek következő viselkedési leírás adható meg, amelyben az egyidejű feltételes hozzárendelő utasítás(oka)t használunk:

```
-- Feladat. 2->4 dekódoló áramkör when-else egyidejű
-- feltételes hozzárendeléssel
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity decoder_2_4_conc is
   port(
      a_in: in std_logic_vector(1 downto 0);
      en_in: in std_logic;
      decode_out: out std_logic_vector(3 downto 0)
   );
end decoder_2_4_conc;

architecture behav of decoder_2_4_conc is
begin
  --one-hot, 1-es bináris súlyú dekódolt kimenetek
    decode_out <= "0000" when (en_in='0') else
             "0001" when (a_in="00") else
             "0010" when (a_in="01") else
             "0100" when (a_in="10") else
             "1000";   -- egyébként ha a_in="11"
end behav;
```

A fenti VHDL kódban először azt teszteljük, hogy en_in engedélyező jel értéke ’0’-e. Ha a feltétel hamis (azaz `en_in` = ’1’), akkor soron következő when-else feltételes utasítás ágak hajtódnak végre előállítva a megfelelő one-hot dekódolt kimeneti értéket.

Egyidejű kiválasztás alapú jelhozzárendelés (with – select)

A másik fontos konkurens jel hozzárendelési módszert kiválasztó hozzárendelési utasításnak vagy **with-select** szerkezetnek is nevezzük, melynek szintaxisa a következő:

```
with kifejezes select
jel_neve <= kifejezes_erteke_1 when valasztas_1,
            kifejezes_erteke_2 when valasztas_2,
            kifejezes_erteke_3 when valasztas_3 | valasztas_4,
            . . .
            kifejezes_erteke_n when others;
```

A fenti **with-select** szerkezettel a `kifejezes` jel értékétől függően rendelünk hozzá a `jel_neve` kifejezéshez valamilyen értéket. A `valasztas_i` jel a `kifejezes` jel egy valódi értéknek, vagy a valódi értékek egy halmazának kell állnia: több lehetőséget is felsorolhatunk ’|’ operátorral elválasztva. Az **`others`** foglalt szót a with-select utasítások végén akkor használhatjuk, amikor minden más lehetőséget, amit nem soroltunk fel, adjuk meg.

Fontos megjegyzések: a **with-select** konkurens kiválasztó jelhozzárendelés esetén a `valasztas_i` lehetőségeknek egymásra nézve egyrészt:

- kölcsönösen kizárólagosnak kell lenniük (mutual exclusive, = mutex tulajdonság), azaz egyazon választási lehetőség sem szerepelhet egyszerre több különböző sorban

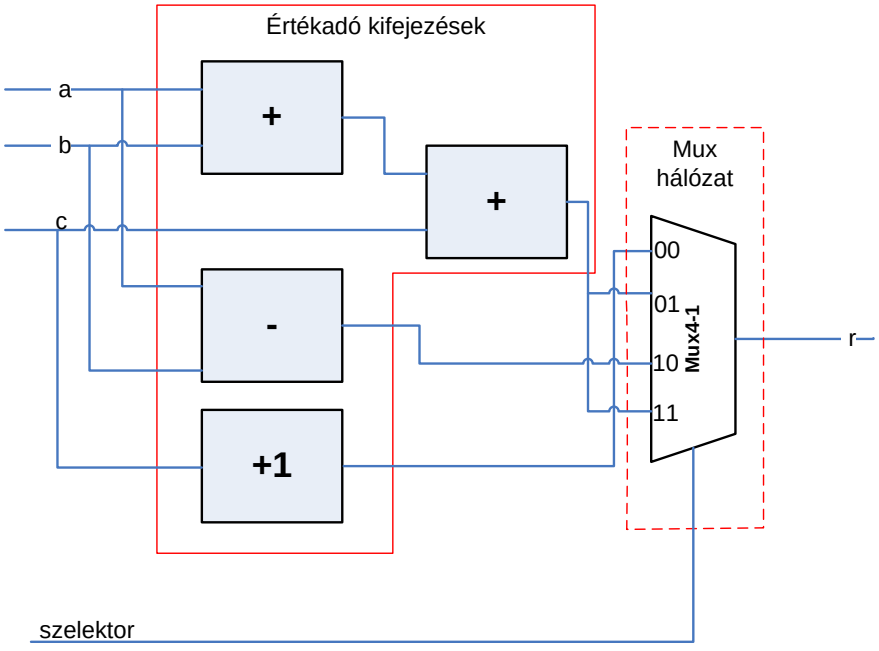
- másrészt minden lehetséges `valasztas_i` lehetőséget fel kell sorolni (all inclusive tulajdonság), nem maradhat ki lehetőség, mert akkor azokat le kell fedni (others).

Vizsgáljuk meg az előző, egyidejű feltételes hozzárendelésnél bemutatott példát, de most az egyidejű kiválasztó hozzárendelési utasításra, kisebb módosítással (a+b+c kiválasztását tekintve):

```
signal szelektor : std_logic_vector(1 downto 0);
…
with szelektor select
rout <=   c + 1    when "00",
    a - b    when "10",
    a + b + c   when others;
```

Összevetve a korábbi áramkörrel az első szintű multiplexer ’1’-es állapotánál kaptunk `a – b` kimenetet, amelyet a második szintű multiplexer ’0’-s állapotán keresztül a `rout` kimenetre továbbítottunk (azaz szelektor ’10’). Hasonló módon az `c+1` bemenetet előállítva, az első szinten lévő multiplexer ’1’-es állapotán, míg a második szintű multiplexer ’0’-s állapotán keresztül jut az `rout` kimenetre. Végül az `a+b+c` előállított bemeneti kifejezésnél azt **`others`** kulcsszót használjuk minden más eset ún. lefedésére.

Ennek az egyidejű kiválasztó jelhozzárendelésnek az áramköri modellje a következő *2.7. ábrán* látható.



2.7. ábra: Egyidejű kiválasztó jel hozzárendelési utasításnak megfelelő elvi áramköri modell

Feladat 3

Tekintsük az egyidejű feltételes hozzárendelésnél korábban megismert 4-bites prioritásos kódoló áramkört (Feladat 1). Ennek az áramkörnek, most az egyidejű kiválasztó jelhozzárendelő utasításokból felépített VHDL viselkedési leírását adjuk meg. Az entitás neve legyen: `priencoder_4_with`.

```
-- Feladat 3. prioritásos kódoló áramkör with-select egyidejű
-- kiválasztó hozzárendeléssel
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity priencoder_4_with is
Port (    req : in  STD_LOGIC_VECTOR (3 downto 0);
    pcode_out : out  STD_LOGIC_VECTOR (2 downto 0));
end priencoder_4_with;

architecture behav of priencoder_4_with is
begin
   with req select
 --összes lehetséges kiválasztási esetet meg kell adni
      pcode_out <= "100" when "1000"|"1001"|"1010"|"1011"|
                           "1100"|"1101"|"1110"|"1111",
                "011" when "0100"|"0101"|"0110"|"0111",
                "010" when "0010"|"0011",
                "001" when "0001",
                "000" when others;   -- req="0000" egyébként
end behav;
```

A fenti kód a lehetséges kiválasztási esetekhez tartozó összes kombinációt megadja a `req(3:0)` jelet illetően. A '|' szimbólum szolgál arra, hogy az több lehetséges kombinációt egyetlen esethez tudjuk rendelni, továbbá az others kulcsszóval minden más esetet le tudunk „fedni".

Feladat 4

Hasonló módon, tekintsük az egyidejű feltételes hozzárendelésnél korábban megismert 2→4-es bináris dekódoló áramkört (lásd *Feladat 2*). Ennek az áramkörnek, most az egyidejű kiválasztó jelhozzárendelő utasításokból felépített viselkedési leírását adjuk meg VHDL nyelven. Az entitás neve legyen: `decoder_2_4_with`.

```
-- Feladat 4. 2->4 dekódoló áramkör with-select egyidejű
-- kiválasztó hozzárendeléssel
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decoder_2_4_with is
   port(
      a_in: in std_logic_vector(1 downto 0);
      en_in: in std_logic;
      decode_out: out std_logic_vector(3 downto 0)
   );
end decoder_2_4_with;

architecture behav of decoder_2_4_with is
    signal sel: std_logic_vector(2 downto 0);
begin
   sel <= en_in & a_in;
   with s select
      decode_out <= "0000" when "000"|"001"|"010"|"011",
                "0001" when "100",
                "0010" when "101",
                "0100" when "110",
                "1000" when others;   -- sel="111" egyebkent
end behav;
```

A fenti VHDL kódban az `en_in` és `a_in(1:0)` jeleket egy 3-bites `sel(2:0)` néven definiált jellé fűzzük össze, éppen azért, hogy a korábbi igazságtáblázatnak megfelelő kiválasztó jelhozzárendeléseket is meg tudjuk adni. A fenti kód a lehetséges kiválasztási esetekhez tartozó összes kombinációt megadja a `sel` jelet illetően. A '|' szimbólum szolgál arra, hogy több lehetséges kombinációt egyetlen esethez tudjuk rendelni, továbbá az ***others*** kulcsszóval minden más esetet le tudunk „fedni".

További feladatok: az előző VHDL nyelven leírt Feladatokhoz 1.)–4.) hozzon létre egy-egy tesztpadot. Ágyazza be a tesztpadba a megfelelő példányosított entitásokat, majd adjon rájuk gerjesztést, és vizsgálja a kimenetek változásait a Xilinx ISim szimulátor segítségével. A szimulációval kapcsolatban bővebb leírást a későbbi *2.4. fejezetben* találhat.

2.3.2. Szekvenciális hozzárendelési utasítások

A VHDL – a hagyományos programozási, modellezési szemléletmódot követve – támogatja a szekvenciális utasítás végrehajtást is, amely, amelyet más magas-szintű nyelvek használatakor már megszokhattunk. A VHDL nyelvben több lehetséges szekvenciális nyelvi konstrukció is biztosított, amelyek közül a legfontosabb szintetizálható utasítások a következők:

- **Process()** – folyamat, amelyben szereplő utasítások bizonyos jelek változására aktivizálódnak (érzékenységi lista).
  - **If – else** utasítás, elágazás, vagy branch (process-en belül adahtó csak meg).
  - **Case** utasítás (process-en belül hívható adható csak meg).

A következő szekvenciális végrehajtású VHDL nyelvi konstrukciók a process() folyamaton belül használhatóak, jelen jegyzetben részletes bemutatásukra nem térünk ki.

- **Loop:** iteratív, ciklikus végrehajtást támogató nyelvi konstrukció
  - **For-loop:** hagyományos 'for' ciklus
  - **While-loop**: hagyományos elől-tesztelő 'while' ciklus
  - **Loop:** tesztelés nélküli ciklusvégrehajtás
- **Next / Exit**: a ciklusokból (loop) való kilépés feltételeit adják meg
- **Wait / Assert:** várakoztató utasítások, illetve bizonyos feltételtől függő beállítások (assertion)
- **Null:** segítségével explicit módon jelezhető, hogy nem szükséges utasítást végrehajtani pl. egy elágazás ágában, vagy egy ciklusban

Folyamat (process)

Folyamat szintaxisa

A folyamat működése eltér az egyidejű, konkurens utasítás végrehajtásától. A folyamat használatának, VHDL leírásba ágyazásának célja, hogy kompatibilis utasításokat biztosítson egy hagyományos szekvenciális végrehajtáshoz [*ASHENDEN*], [*KERESZTES*]. A folyamat egy olyan kód-sorozatnak tekinthető, amely mindig végrehajtásra kerül, ha az érzékenységi listáját szereplő jelek valamelyikén változás áll be. Az **érzékenységi lista** (`sensitivity list`) azoknak a jeleknek a gyűjteménye, amelyekre egy folyamat érzékeny: például kombinációs logikai hálózatok esetén a bemeneteket, míg sorrendi hálózatok esetén – megvalósítandó modelltől függeően – általában a bemeneteket, és a visszacsatolt állapotokat is meg kell adni. Az érzékenyégi listában szereplő jeleket zárójelben, vesszővel elválasztva kell felsorolni. A process() `begin…end` közötti részén, azaz a folyamat törzsében a szekvenciális utasításokat adhatjuk meg. A '[ ]' zárójelben megadott részek opcionálisak, azaz nem szükséges megadni őket, viszont a folyamat neve egyedi kell, hogy legyen, amely így segíthet a szintézis során felmerülő esetleges hibák egyértelmű azonosításában. A `szekvenciális_utasítás_i` egy <= jellel kifejezett értékadó utasítást jelent. A `process` egyszerűsített szintaxisa a következő:

```
[process_neve] : process(erzekenysegi_lista) [is]
begin
 szekvenciális_utasítás_1;
 szekvenciális_utasítás_2;
 . . .
 szekvenciális_utasítás_n;
end process;
```

Folyamat szemantikája

A ***folyamat***ok önmagukban tehát sorrendi, de egymáshoz képest – kívülről nézve – egyidejű programrészek. Ha több folyamat van egy VHDL entitás architektúrális részén, akkor azok ténylegesen egyidejűleg fognak végrehajtódni. Ha például az egyidejű/konkurens hozzárendelési utasítások között soros/szekvenciális végrehajtású blokkokat is szeretnénk definiálni, akkor a `process`()-t, mint soros VHDL nyelvi konstrukciót kell használnunk.

Egy folyamat akkor aktivizálódik, ha az ún. **érzékenységi listáján** – szereplő jelek valamelyikén értékváltozás jelenik meg. Fontos azonban tudni, hogy egy folyamat működése bármikor felfüggeszthető (wait utasításokkal), akár valamilyen feltétel bekövetkezéséig, vagy akár végtelen hosszú ideig. Ugyanakkor egy folyamat nem hívható meg kétszer, mint pl. egy függvény (function), vagy alprogram (procedure). Minden folyamat a VHDL szimuláció elejétől indul és végig „él”, de legfeljebb csak a végrehajtása függeszthető fel bizonyos időre, azaz nem szüntethető meg. A `process()` kizárólagosan vagy **várakozási/tétlen (waiting/passive)**, vagy **végrehajtási/aktív (executing/active)** állapotban lehet.

Bár a szintaxisa nagyon hasonlít a normál konkurens értékadó utasításhoz, azonban ha egy process()-en belül definiált ugyanazon jelhez többszörös szekvenciális hozzárendelési utasítást adunk meg, akkor mindig az utolsó értékadó utasításnak lesz csak hatása:

```
Signal a, b, c_int, q : std_logic;
A1 : process(a ,b)
begin
 c_int <= a and b; --1
 c_int <= a or b;  --2
 q <= not c_int;   --3
end process;
```

amely egyenértékű a következő szekvenciális hozzárendeléssel (a, b, c_int inputok es q output itt signal-kent vannak definiálva, míg a c nincs a folyamat érzékenységi listájában sem megadva):

```
Signal a, b, c_int, q : std_logic;
A2: process(a ,b)
begin
 c_int <= a or b; --1
 q <= not c_int;  --2
end process;
```

A signal-ként definiált belső jel (`c_int`) esetén a `'q<=not _c_int'` jelhozzárendelés végrehajtásakor az `c_int` még a régi értékét tartja. Ezért fel kell venni őt a folyamat érzékenységi listájába, így egy delta késleltetés után a `process()` újra tevékennyé válik. A folyamat egy olyan kód sorozat, amely mindig végrehajtásra kerül, ha az érzékenységi listáját szereplő jelek valamelyikén változás jelentkezik. A példából is látható a változó használatának előnye, hiszen a jel esetében figyelni kell a delta késleltetésre és a folyamatnak kétszer kell lefutnia. Tehát helyesen a következő lenne a sorrendi VHDL kódrészlet:

```
Signal a, b, c_int, q : std_logic;
A2: process(a ,b, c_int)  --c_int hozzaadva!
begin
 c_int <= a or b; --1
 q <= not c_int;  --2
end process;
```

Megjegyzések

1.) A VHDL szabvány megengedi azt is, hogy az érzékenységi listából mellőzzük a jelek felsorolását. Viszont, ha egy jel kimarad egy szintetizálandó VHDL tervben az érzékenységi listából, amelynek vizsgálatára a folyamat blokkjában szükség lehet, akkor a VHDL szimuláció és a szintetizált hardver akár eltérően is viselkedhet.

2.) A belső jel használata rejtett hibához vezethet a szintézisben (pl. Xilinx XST eszközt használva), mivel a folyamat érzékenységi listáját figyelmen kívül hagyhatja sok szintézis eszköz. Ezt kiküszöbölendő a későbbi fejezetekben a regisztereknél ismertetett módon lehet a jeleket tárolásra használni (jel aktuális, illetve következő állapotai közötti hozzárendelések megadásával).

Jelek és változók szemantikai összehasonlítása folyamatok használatával (diszkrét események időbeli modellje)

A VHDL időzítési modellje azon alapul, hogy a modellezett rendszer a bemeneteire adott gerjesztésekre (stimulus) meghatározott módon válaszol, majd a gerjesztés további változásaira várakozik (wait). A szimuláció során az egyes **események** időpontjai mindig szimulációs időben mértek, azaz függetlenek a fizikai időtől. A szimulátor diszkrét időegységekben lépkedve dolgozza fel a szimulációs időpontokra ütemezett eseményeket, amelyeket egyrészt a gerjesztések, másrészt a modellezett rendszernek a gerjesztésekre adott válasza határoz meg.

`Esemény` alatt egy jel értékének megváltozását értjük. Egy adott szimulációs időpontban az események feldolgozása egy vagy több ún. `szimulációs ciklusban` történik. A szimulációs ciklusnak két üteme van.

- Az 1. ütemben a jeleknek az értéke változik meg, amelyek változása az adott szimulációs időpontra van előírva.
- A 2. ütemben lefutnak azok a folyamatok, amelyeket az 1. ütemben megváltozott jelek tesznek tevékennyé.

A folyamatok újabb jelek értékváltozásait írhatják elő, így előfordulhat, hogy az adott szimulációs időpontban további szimulációs ciklusokat is végre kell hajtani. Egy szimulációs

időpontban több szimulációs ciklus egymás utáni végrehajtásakor gondoskodni kell arról, hogy a ciklusok ne zérus idő alatt fussanak le, mert különben a jelváltozások között nem biztosítható az ok-okozati viszony (kauzalitás). Ezért minden szimulációs ciklushoz egy elméleti **delta késleltetést** rendel a VHDL szimulátor, ami a szimulációs időt nem befolyásolja.

Egy adott szimulációs időpontban a szimulációs ciklusok mindaddig fognak ismétlődni, amíg a folyamatokon végig nem gyűrűzik a kezdeti jelváltozások hatása. Ha az adott szimulációs időpontra már nincs több jelváltozás előírva, akkor a szimulátor a következő szimulációs időpontra ütemezett eseményeket kezdi el feldolgozni.

A folyamatok használatában a jelek és változók közötti legfontosabb különbségeket szemlélteti a következő VHDL kódrészlet a folyamat használatára:

```
-- process jelekkel: s1 és s2
-- deklaráció
signal s1, s2 : integer := 0;
. . .
Proc_sig: process
begin
  wait for 10 ns;
  s1<=s1+1;
  s2<=s1+2;
end process;
--
-- process változókkal: v1 és v2 változók
proc_variable: process
  --deklaráció
  variable v1, v2: integer := 0;
begin
  wait for 10 ns;
  v1:=v1+1;
  v2:=v1+2;
end process;
```

Az alábbi *2.7. táblázat* a fenti VHDL kódnak megfelelő viselkedését mutatja be a jeleknek, illetve változóknak. Az egyes oszlopokban az aktuális szimulációs időt, az `s1, s2` szignálokat, valamint `v1, v2` néven definiált változók kiértékelésének eredményét láthatjuk. A két független process egymással párhuzamosan, konkurens módon fut le. Megfigyelhető, hogy a szimulációs idő 10 ns elteltével, a `wait for 10 ns` utasításnak megfelelően a változók rögtön felveszik az új értéküket sorban egymás után, míg ez az értékváltozás a jelek esetén mindig csak egy ún. szimulációs **delta** időkésleltetéssel később következik be:

2.7. táblázat: a folyamaton belüli jelek (s1, s2) és változók (v1, v2) összehasonlítása az értékadások időzítése szempontjából

Szimulációs idő	s1	s2	v1	v2
0 ns	0	0	0	0
10 ns	0	0	1	2
10 ns+**delta**	1	1	1	2
20 ns	1	1	2	3
20 ns+**delta**	2	2	2	3

A ***delta idő***t a VHDL-ben a sorrendi események sorba állítására használjuk. A két sorrendi esemény közötti időt *delta késleltetés*nek nevezik. Egy delta időnek nincs valósidőbeli egyenértéke, hanem miközben telik (végrehajtás történik/aktív folyamat), a szimulációs idő nem halad.

If-else utasítás

Az If-else egy hagyományos szekvenciális feltételes utasítás, amelynek egyszerűsített szintaxisa a következő:

```
if boolean_kifejezes_1 then
 szekvencialis_utasítás(ok);
elsif boolean_kifejezes_2 then
 szekvencialis_utasítás(ok);
elsif boolean_kifejezes_3 then
 szekvencialis_utasítás(ok);
. . .
else
 szekvencialis_utasítás(ok);
end if;
```

A `boolean` kifejezések egymás után, szekvenciálisan értékelődnek ki mindaddig, amíg egy feltételes ágban lévő kifejezés igaz nem lesz, vagy az `else` ágat el nem érjük.

Korábbi `when-else` egyidejű hozzárendelési példánknál maradva, amelyet így írtunk fel, hogy:

```
rout <= a + b + c when m = n else
        a - b     when m > n else
        c + 1;
```

ezt a hagyományos `if-else` szekvenciális hozzárendelési utasításokkal a következő módon lehet megadni:


```
process (a, b, c, m, n)    -- bemeneti jelekre érzékeny folyamat
begin
 if m = n then
    rout <= a + b + c after 2 ns;  --egyedi késleltetés
 elsif m > 0 then
    rout <= a - b after 10 ns;
 else
    rout <= c + 1 after 5 ns;
 end if ;
end;
```

Megjegyzés: Az előző kódrészlet az egyidejű feltételes jelhozzárendeléshez hasonló multiplexer hálózatot fog kialakítani a szintézis során. A jelek hozzárendeléseihez egyedi késleltetéseket adtunk meg, after (clause) kulcsszó használata után megadott fizikai-időbeli típus konkrét megadásával.

Feladat 5

Tekintsük az egyidejű feltételes hozzárendelésnél korábban megismert 4-bites prioritásos kódoló áramkört (Feladat 1), de most ennek az áramkörnek, az if-else szerkezettel kifejezett szekvenciális jelhozzárendelő utasításokból álló VHDL leírását adjuk meg. Az entitás neve legyen: `priencoder_4_if`. Mint ismeretes az if-else szerkezetet a VHDL leírás folyamatának `begin`…`end` közötti törzsébe kell beágyazni.

```
-- Feladat 5. prioritásos kódoló áramkör if-else szekvenciális
-- hozzárendeléssel megadva
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity priencoder_4_if is
Port (    req : in  STD_LOGIC_VECTOR (3 downto 0);
    pcode_out : out  STD_LOGIC_VECTOR (2 downto 0));
end priencoder_4_if;

architecture behav of priencoder_4_if is
begin
   if_process: process(req)
   -- bemeneti req jelre érzékeny folyamat
   begin
      if (req(4)='1') then
         pcode_out <= "100";
      elsif (req(3)='1') then
         pcode_out <= "011";
      elsif (req(2)='1') then
         pcode_out <= "010";
      elsif (req(1)='1') then
         pcode_out <= "001";
      else
         pcode_out <= "000";
      end if;
   end process;
end behav;
```

Feladat 6

Hasonló módon, tekintsük az egyidejű feltételes hozzárendelésnél korábban megismert 2→4 bináris dekódoló áramkört (Feladat 2). Ennek az áramkörnek, most az szekvenciális if-else feltételes hozzárendelő utasításokból felépített viselkedési leírását adjuk meg VHDL nyelven. Az entitás neve legyen: `decoder_2_4_if`.

```
-- Feladat 6. 2->4 dekódoló áramkör if-else feltételes szekvenciális
-- hozzárendeléssel megadva
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decoder_2_4_if is
   port(
      a_in: in std_logic_vector(1 downto 0);
      en_in: in std_logic;
      decode_out: out std_logic_vector(3 downto 0)
   );
end decoder_2_4_if;

architecture behav of decoder_2_4_if is
begin
   process(en_in, a_in)
   -- bemeneti en_in, a_in jelekre érzékeny folyamat
   begin
      if (en_in='0') then
         decode_out <= "0000";
      elsif (a_in ="00") then
         decode_out <= "0001";
      elsif (a_in ="01") then
         decode_out <= "0010";
      elsif (a_in ="10") then
         decode_out <= "0100";
      else              -- a_in ="11"
         decode_out <= "1000";
      end if;
   end process;
end behav;
```

További feladatok: a VHDL nyelven leírt Feladat 5.)-6.) –okhoz hozzon létre egy-egy tesztpadot. Ágyazza be a tesztpadba a megfelelő példányosított entitásokat, majd adjon rájuk

gerjesztést, és vizsgálja a kimenetek változásait a Xilinx ISim szimulátor segítségével. A szimulációval kapcsolatban bővebb leírást a következő *2.4. fejezetben* találhat.

Várakoztató – (wait) utasítások

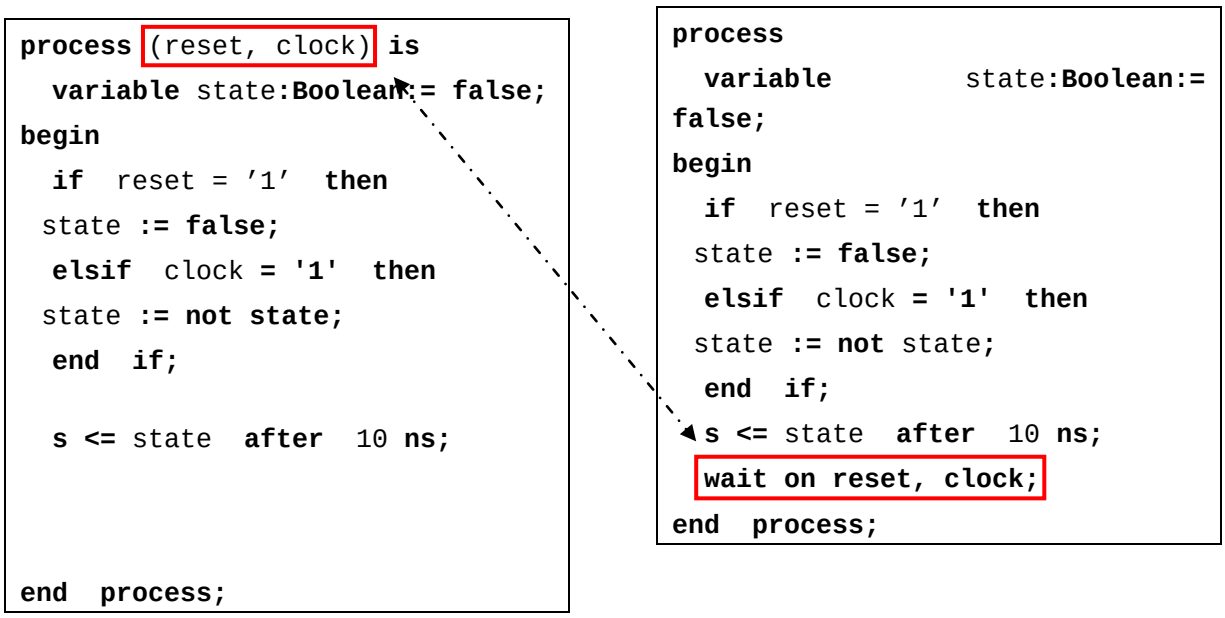
A speciális szekvenciális várakoztató utasításoknak a következő négy lehetséges formája van:

- **Wait**: utasítássorozat végrehajtása leáll, végtelen ideig várakozunk (pl szimulációs process végén használjuk).
- **Wait on** [szenzitív_lista]: jel változására várakozunk.
- **Wait for** [idő_kifejezés]: meghatározott időtartamig várakozunk.
- **Wait until** [feltétel]: addíg várakozunk, amíg a feltételben lévő kifejezés értéke igaz nem lesz.

A következő néhány szemléltető példa a **wait** utasítás lehetséges megadási módjainak használatát szemlélteti:

```
wait; --ha nem használunk az utasítés után semmiféle opcionális kifeje-
zést, -- akkor azt jelzi, hogy végtelen ideig várakoztatunk

signal clk: std_logic;
. . .
wait until (clk);
--addig várakozunk amíg a clk értéke ’1’ nem lesz

constant T : time : = 20 ns;
. . .
wait for T/2;
-- T/2 periódus ideig várakozunk


signal a, b : std_logic;
. . .
wait on a,b;
-- a és b értékének változására várakozunk (pl. process()szenzitív lista
-- helyettesítése
```

Érdemes megjegyezni, hogy a szenzitív listát megadhatjuk egy folyamat deklarációs részében a process() kulcsszó utáni zárójeles részen, ami teljesen ekvivalens azzal, mintha a folyamat törzsében az utolsó utasítás egy wait on utasítás lenne:



'Nem kívánt memória' hatásának kiküszöbölése

A `process()` alkalmazása esetén bizonyos 'rejtett hiba' adódhat a nem megfelelő érzékenységi lista beállításából, azaz ún. „nem kívánt memória" (unintended memory) jöhet létre akár egy kombinációs logikai hálózatban is. Ez nem a VHDL nyelv, hanem sokkal inkább a tervezés során használt szintézis eszköz (esetünkben Xilinx XST) működéséből adódik. A VHDL szabvány szerint egy jel-vezeték (signal) mindaddig megtartja korábbi értékét, ameddig nem rendelünk hozzá egy új értéket, például process()-ben lévő if-else szekvenciális utasítás segítségével. A Xilinx szintézis folyamat során így egy belső állapot alakulhat ki (zárt visszacsatoló hurokban), amely nem kívánt memória elemet hoz létre (hasonlóan egy direkt módon definiált tárolóhoz, pl: D-flipflop-hoz, vagy latches). Azért, hogy ezt kiküszöböljük, a `process()`-en belüli helytelen jelhozzárendelésekből adódó 'nem kívánt memória' elem szintézisénél az I.) vagy II.) módszert kell követni.

Az I.) módszer esetén a következő **szükséges feltételeket kell** biztosítani:

- a.) Szenzitív listába minden szükséges jelet soroljunk fel,
- b.) Egy `if-else` szerkezetnek mindig adjuk meg az `else` ágát is,
- c.) A `process()`-en belüli minden `if-else` ágban, és mindenegyes jelhez rendeljünk hozzá új értéket.

Amennyiben a három a.)-c.) szükséges feltételek egyszerre teljesülnek, úgy egyben az elégséges feltétel is teljesül, ezáltal elkerülhetők a nem szándékosan tervezett tárolóelemek XST szintézisének problémái, akár kombinációs hálózatok VHDL leírása esetén is.

A következő kódrészlet egy hibás megadásra mutat példát:

```
process (a_in) -- hiba: szenzitív listából hiányzó b_in jel megadása
begin
 if ( a_in > b_in ) then
    nagyobb_mint <= '1';
    -- hiba az 'egyenlo'-hez nincs ebben az ágban hozzárendelve
    --új érték
 elsif ( a_in = b_in ) then
    egyenlo <= '1';
    -- hiba az 'nagyobb_mint'-hez nincs ebben az
    --ágban hozzárendelve új érték
 --hiba, az else ág is hiányzik
 end if ;
end process;
```

Bár a fenti szintaxis helyes (a VHDL nyelvi szintaxis ellenőrzésnek is megfelel), azonban megsérti a korábban felsorolt három szükséges feltételt (I.): mivel pl. a `nagyobb_mint` megőrzi korábbi értékét, amikor `a > b` kifejezés hamis, és ezáltal a VHDL nyelv egy szint-vezérelt regiszter (latch) tárolót fog szintetizálni (=infer). A fenti VHDL forráskód helyes megadási módja a három szabályt a.) –c.) is figyelembe véve a következő lenne:

```
process (a_in,b_in)
begin
 if ( a > b ) then
    nagyobb_mint <= '1';
    egyenlo <= '0';
 elsif ( a = b ) then
    nagyobb_mint <= '0';
    egyenlo <= '1';
 else
    nagyobb_mint <= '0';
    egyenlo <= '0';
 end if ;
end process;
```

II.) módszer szerint a fenti leírással ekvivalens módon lehet a következő kódrészlet is, amelyben a jelek alapértelmezett (default) értékeit a `process()`-en belül, de az `if-else` ágakon kívül definiáljuk, megelőzve a nem kívánt memória elem kialakulását szintézis során.

```
process (a_in,b_in)
begin
  --alapértelmezett (default) értékek
  nagyobb_mint <= '0';
  egyenlo <= '0';

  if ( a > b ) then
     nagyobb_mint <= '1';
  elsif ( a = b ) then
     egyenlo <= '1';
  end if ;
end process;
```

A fenti kódrészlet azért is lehet helyes, mivel kezdetben a két jelnek '0' kezdőérték van definiálva, amelyeknél később a soros végrehajtás esetén előfordulhat, hogy nem rendelnénk hozzájuk új értéket (I. módszerben az else ágat elégíti ki), illetve az értékadáshoz képest az egyes if-else ágakban szereplő jelek felülíródhatnak új értékekkel.

Case utasítás

Az Case (eset) szintén hagyományos szekvenciális feltételes utasítás, amelynek egyszerűsített szintaxisa a következő:

```
case kifejezes is
  when valasztas_1 =>
     szekvencialis_utasitasok;
  when valasztas_2 =>
     szekvencialis_utasitasok;
  ...
  when others =>
     szekvencialis_utasitasok;
end case;
```

A Case utasításban a `kifejezes` hasznalható arra, hogy a megfelelő szekvenciális utasítások ágát hajtsuk végre, a `valasztas_i` értékétől függően. A `valasztas_i` lehetőségek egymásra nézve kölcsönösen kizártak, azaz csak egyszer sorolhatóak fel, másrészt pedig az összes lehetséges használni kívánt esetet fel kell sorolni. Az **others** kulcsszóval az összes nem használt eset „lefedhető".

Korábbi with-select egyidejű kiválasztó hozzárendelési példánknál maradva, amelyet így adtunk meg:

```
with szelektor select
rout <=   c + 1    when "00",
    a - b    when "10",
    a + b + c  when others;
```

amelyet a mostani hagyományos case szekvenciális hozzárendelési utasítást alkalmazva a következő módon írhatunk fel:

```
process (szelektor, a, b, c)     -- bemeneti jelekre érzékeny folyamat
begin
 case szelektor is
    when "00" =>
       rout <= c + 1;
    when "10" =>
       rout <= a - b;
    when others =>
       rout <= a + b + c;
 end case ;
end;
```

Megjegyzés: A fenti kódrészlet az egyidejű kiválasztó jelhozzárendeléshez hasonló multiplexer hálózatot fog kialakítani a szintézis során.

Feladat 7

Tekintsük az egyidejű feltételes hozzárendelésnél korábban megismert 4-bites prioritásos kódoló áramkört (Feladat 1), azonban most ennek az áramkörnek, az case szerkezettel kifejezett szekvenciális jelhozzárendelő utasításokból álló VHDL leírását adjuk meg. Az entitás neve legyen: `priencoder_4_case`. Mint ismeretes az case szerkezetet a VHDL leírás folyamatának `begin…end` közötti törzsébe lehet beágyazni.

```
-- Feladat 7. prioritásos kódoló áramkör case szekvenciális
-- hozzárendeléssel megadva
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity priencoder_4_case is
port (    req : in  STD_LOGIC_VECTOR (3 downto 0);
    pcode_out : out  STD_LOGIC_VECTOR (2 downto 0));
end priencoder_4_case;

architecture behav of priencoder_4_case is
begin
   case_process: process(req)
   -- bemeneti req jelre érzékeny folyamat
   begin
      case req is
         when "1000"|"1001"|"1010"|"1011"|
              "1100"|"1101"|"1110"|"1111" =>
            pcode_out <= "100";
         when "0100"|"0101"|"0110"|"0111" =>
            pcode_out <= "011";
         when "0010"|"0011" =>
            pcode_out <= "010";
         when "0001" =>
            pcode_out <= "001";
         when others =>
            pcode_out <= "000";
      end case;
   end process;
end behav;
```

Feladat 8

Hasonló módon, tekintsük az egyidejű feltételes hozzárendelésnél korábban megismert 2→4 bináris dekódoló áramkört (Feladat 2). Ennek az áramkörnek, most az szekvenciális case feltételes hozzárendelő utasításokból összeállított viselkedési leírását adjuk meg VHDL nyelven. Az entitás neve legyen: `decoder_2_4_case`.

```
-- Feladat 8. 2->4 dekódoló áramkör case feltételes szekvenciális
-- hozzárendeléssel megadva
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decoder_2_4_case is
   port(
      a_in: in std_logic_vector(1 downto 0);
      en_in: in std_logic;
      decode_out: out std_logic_vector(3 downto 0)
   );
end decoder_2_4_case;

architecture behav of decoder_2_4_case is
begin
   sel <= en_in & a_in;
   process(sel)
   begin
      case sel is
         when "000"|"001"|"010"|"011" =>
            decode_out <= "0001";
         when "100" =>
            decode_out <= "0001";
         when "101" =>
            decode_out <= "0010";
         when "110" =>
            decode_out <= "0100";
         when others =>
            decode_out <= "1000";
      end case;
   end process;
end behav;
```

Feladat 9

Vizsgáljuk meg a szekvenciális és az egyidejű hozzárendelési utasítások közötti kapcsolatot egy maximum-kiválasztási példán keresztül, ahol az `a_in, b_in` és `c_in` bemeneteket rendezzük és a legnagyobbat `maximum` néven a kimenetekre helyezzük.

Relációs operátorokat ('>') használva egyidejű hozzárendelés során a következő lehetséges VHDL kódrészletet adhatjuk meg:

```
maximum <=   a_in when ((a_in > b_in) and (a_in > c_in)) else
    c_in when (a_in > b_in) else
    b_in when (b_in > c_in) else
    c_in;
```

A fenti kód nem tartalmaz egymásba ágyazott VHDL nyelvi konstrukciókat, és kevésbé jól írja le a kiválasztást. Ugyanezt a maximum-kiválasztást, amennyiben fel kívánjuk írni a megismert szekvenciális if-else szerkezettel (egy folyamaton belül) akkor a következő szemléletesebb, de hosszabb leírást kapjuk:

```
If_process: process (a_in , b_in, c_in)
begin
 if a_in > b_in then
    if a_in > c_in then

       maximum <= a_in;
    else
       maximum <= c_in;
    end if;
 else
    if b_in > c_in then
       maximum <= b_in;
    else
       maximum <= c_in;
    end if;
 end if;
end;
```

További gyakorló feladatok

1.) A VHDL nyelven leírt fenti 8.)-9.) Feladatokhoz hozzon létre egy-egy tesztpadot. Ágyazza be a tesztpadba a megfelelő példányosított entitásokat, majd adjon rájuk gerjesztést, és vizsgálja a kimenetek változásait a Xilinx ISim szimulátor segítségével. A szimulációval kapcsolatban bővebb leírást a *2.4. fejezetben* találhat.

2.) A *2.4. fejezetben* ismertetett 1–12.) tervezési lépéseket felhasználva és a korábbi Feladat-1 példát átgondolva tervezzen meg VHDL nyelven olyan áramkör(öke)t, amely

az 4-bites egyenlőség összehasonlításán kívül az egyenlőtlenségeket (A kisebb, mint B; illetve A nagyobb, mint B) is képes jelezni. A 3-kimenetű logikai hálózat (comparator_4_bit.vhd) tervezése során használja fel az egyenlőség, egyenlőtlenségek relációs operátorait, és működését definiálja a következő esetekre:

- egyidejű hozzárendelő utasítások (when-else, with-select), illetve
- szekvenciális hozzárendelő utasítások (if-else, case).

a.) Tervezzen egy tesztpadot, comparator_4_bit_tb.vhd néven. Szimulálja le a viselkedését a Xilinx ISim segítségével.

b.) Az FPGA-s eszközön az elkészült terv verifikációjához használjon programozható kapcsolókat, ahol sw(3:0)=a_in(3:0), illetve sw(7:4)=b_in(3:0), míg a kimeneti jelek vizsgálatához a LED-eket, Led(2:0) = (gt_out:lt_out:eq_out) a három különböző összehasonlítás eredményéhez rendelve.

c.) Szintetizálja, implementálja, majd pedig generálja le a konfigurációs bitfájl, végül pedig programozza fel a kiválasztott FPGA-ra. A letöltéshez használja a Digilent Adept Suite nevű programját. Ellenőrizze a szintetizált terv helyes működését a kártyán.

2.4.Xilinx ISE környezet és az ISim szimulátor használata

A fejezetben a Xilinx ISE WebPack 12.2 Project Navigator (továbbiakban ISE) [XILINX] integrált fejlesztő környezet bemutatását és használatát egy példán keresztül szemléltetjük.

Tekintsük a korábbi 4-bites összehasonlító áramkör felépítését (Feladat 1, *2.2. fejezet*), amelynek VHDL forráskódjait az ISE környezetben fejlesztjük, szintetizáljuk és implementáljuk egy kiválasztott Xilinx Spartan-3E FPGA áramkörre, majd pedig itt generáljuk a futtatató tárgykódú konfigurációs bitfájlt is. A tervezés során a Xilinx ISim beépített integrált szimulátorát használjuk az elkészült, és lefordított VHDL forrásaink működésének idődiagramokon keresztüli vizsgálatára. Megjegyeznénk, hogy alternatív szimulátorként használhatnánk a szintén ingyenesen elérhető ModelSim PE Student Edition változatot is, amely mégtöbb lehetőséget és beállítást biztosít a szimulációs vizsgálatok során. [MODELSIM]. (Korábban ezt Xilinx ModelSim Starter XE-III néven adták ki.)

2.4.1. Xilinx fejlesztő környezet, mint használt keretrendszer rövid bemutatása

A Xilinx ISE Project Navigator egy olyan integrált fejlesztő környezet, amely egy keretrendszerként integrálja a fejlesztés egyes fázisait segítő programrészeket (parancsokat), vagy *belső* ISE modulokat (amelyek Tools menüből érhetőek el, vagy a felső ikonsoron keresztül). Ezek a Xilinx modulok a következők:

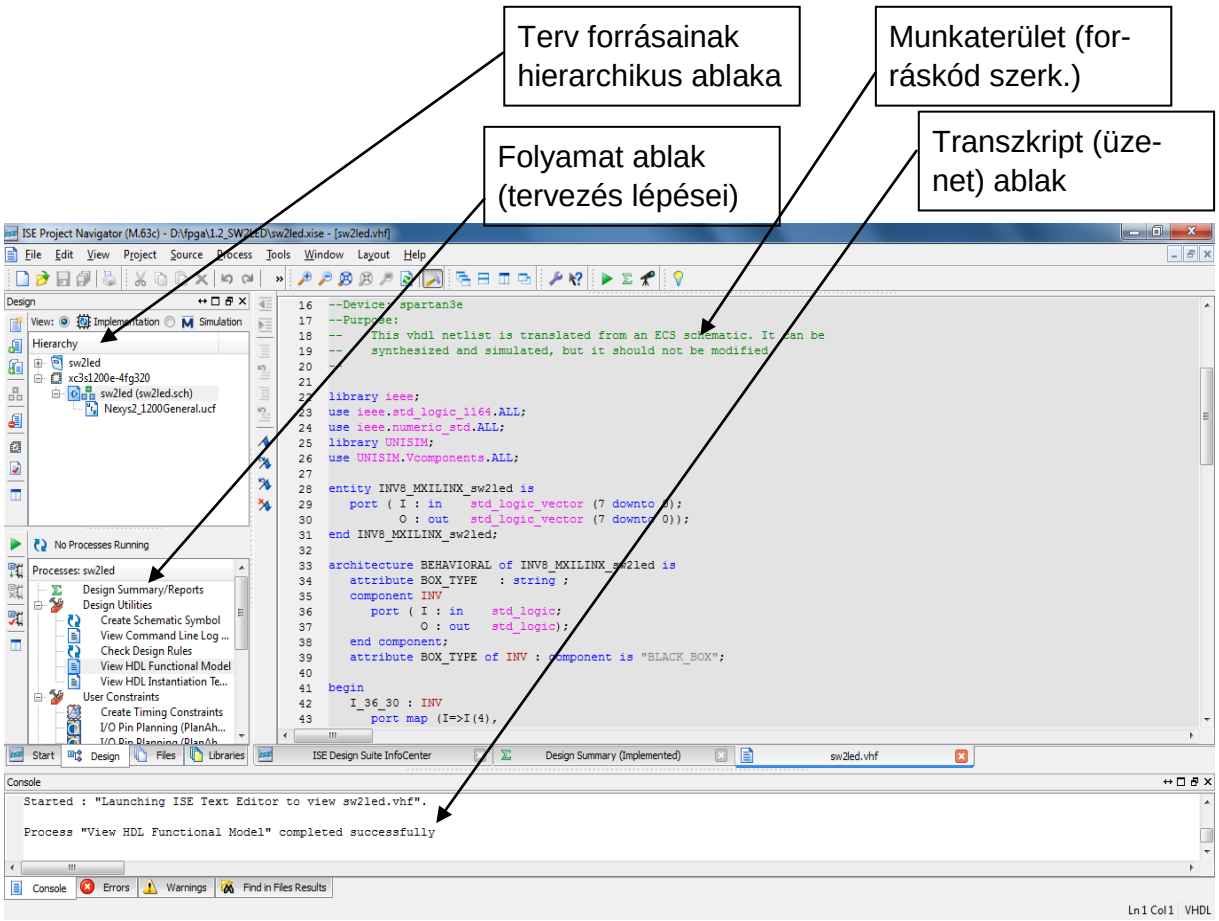
- CoreGenerator – Xilinx IP magok paraméterezése és generálása a terveinkhez. Xilinx logikai cellákat, makrócellákat (primitíveket) is használ, amelyek FPGA család specifikusak is lehetnek.
- PlanAhead – Tervezés menedzselése, I/O portok együttes kezelése együtt (alapvető verzió-követési és moduláris tervezési funkciókkal).
- Shematic Editor – Kapcsolási-rajz szerkesztő.
- Timing Analyzer – Időzítési analizátor modul.
- XPower Analyzer – Teljesítmény analizátor modul.
- Constraint Editor – Kényszerfeltételek megadására szolgáló modul (grafikus nézetet biztosít a lábak bekötésére, amely erősen FPGA specifikus).
- FPGA Editor – FPGA fizikai szintű logikai és makró celláinak esetleges elhelyezésére és összekötésére.
- Impact – Bitfolyam letöltő (programozó) modul, stb. (Jelen jegyzetben **nem** ezt fogjuk használni, mivel speciális JTAG-USB programozó kell hozzá**. Helyette a Digilent Adept Suite programját használjuk a bitfájlok letöltésére – roll-on USB kábelen kereszül! [DIGILENT]).**

Az ISE keretrendszerhez, további *külső* (külön megvásárolható, és telepíthető) modulként a következő programrészek kapcsolódhatnak, amelyet a fejlesztések során használhatunk:

- XPS Xilinx Platform Studio: Integrált keretrendszer beágyazott rendszerek fejlesztésére, amely intágrálja a következő HW-SW moduláris fejlesztő környezeteket (bár ezeket a legújabb verziókban külön-külön is lehet telepíteni és használni).
  - EDK (Embedded Development Kit) – Beágyazott (hardver) rendszerfejlesztő környezet [*XILINX_EDK*]
  - SDK (Software Development Kit) – Beágyazott (szoftver) alkalmazás fejlesztő környezet, amely Eclipse alapú
- ChipScope – Logikai analizátor, az FPGA működése közbeni belső jelek vizsgálatára (debug funkciók)
- System Generator for DSP: MatLab Simulink programba épülő Xilinx DSP feldolgozást támogató eszközök gyűjteménye (speciális jelfeldolgozó, és video feldolgozó blokkokkal kiegészítve)

A Xilinx ISE környezetben található mind a belső, mind pedig a külső modulok általában fejlett GUI támogatással rendelkeznek, amelyek sok esetben lépésről-lépésre haladva, mintegy „varázsló”-szerűen segítik, gyorsítják a tervezés menetét. A GUI alapú tervezés mellett, természetesen – és általában a professzionálisabb, gyorsabb alkalmazást ez jelenti – parancssoros módon (scriptek segítségével) is fejleszthetünk.

Az Xilinx ISE (12.2) elindítása után a következő főablak jelenik meg (*2.8. ábra*):



2.8. ábra: Xilinx ISE Project Navigátor fontosabb ablakai

- Forrás ablak (Source window): hierarchikusan jeleníti meg a projektben létrehozott, vagy hozzáadott forrásfájlokat
- Folyamat ablak (Process window): az aktuálisan kiválasztott forrás fájl(ok) tervezési lépéseit, fordítás folyamatait jeleníti meg
- Üzenet ablak (Transcript window): a fordítási lépések üzeneteit, státuszát, figyelmeztetéseit és hibáit jeleníti meg (kivéve a fordítási riportokat)
- Munkaterület (Workplace): egyszerre több forrás fájl (akár keverten HDL, séma, vagy állapot diagram) is meg lehet nyitva, amelyeket aktuálisan szerkesztünk. A fordítási lépések végén a különböző összefoglaló riportok is ebben jelennek meg .html formában.

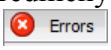
2.4.2. Feladat megvalósítása Xilinx ISE segítségével

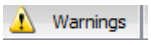
Fontos megjegyzések, amiket a tervezés során mindenképpen érdemes figyelembe venni:

- **Figyeljünk arra, hogy az elérési út NE tartalmazzon ékezetet és white-space karaktereket!**
- **A projekt neve, és a VHDL forrás neve NE kezdődjön számmal, bár ugyan a névben állhat szám az elsőtől különböző pozícióban.**
- **Lehetőség szerint a projekt neve és a forrás(ok) neve legyen eltérő, valamint névnek NE válasszunk foglalt VHDL azonosítót az esetleges későbbi hibaüzenetekben szereplő könnyebb azonosítás végett.**
- **Az entitások neveit tekintve a könnyebb értelmezhetőség és követhetőség végett érdemes lehet konvenciókat felállítani és követni: jegyzetünkben ezért a „`_top`” jelenti a legfelsőbb hierarchia szinten definiált entitást, míg a „`_tb`” végződés a test-bench rövidítését.**

Projekt létrehozása, HDL leírások megadása (részletes):

1. Hozzunk létre egy új projektet a Xilinx ISE Project Navigátorában (**File → New Project,** vagy az ikonsoron **New Project ikon)**. Válasszuk ki és adjuk meg a projekt könyvtár helyét és nevét (ügyelve arra, hogy számmal nem kezdődhet): „`<meghajtó>:\fpga\`”, illetve neve „`feladat_01`” (lásd korábbi példa). Top-level source type, azaz a legfelsőbb szintű architektúra forrásának típusa legyen „HDL”, mivel kezdetben a hagyományos VHDL leíró nyelvet kívánjuk használni a tervezés legfelsőbb szintjén (design entry). Itt lehet akár blokkdiagram szintű (schematic), generált EDIF (szabványos elektronikai tervezési formátum) szintű, vagy a már korábbi köztes Xilinx fordítási lépésként kapott NGC/NGO szintű terveket is beállítani.
2. Következő „Set Device Properties” ablakban a legördülő listákból válasszuk ki a kategóriát, FPGA családot, eszköz típusát, tokozását és sebességi fokát: rendre All, Spartan3E, XCS1200E, FG320, és -4. Ezek az alkalmazott FPGA-s eszköz típusának pontos paramétereit definiálják, a Digilent Nexys-2 kártya adatlapjában (Reference Manual [*NEXYS2*]) pontosan meg vannak adva, illetve a legtöbb paraméter az FPGA-s chip tokozásán is leolvasható.

3. A szintézis eszköz legyen az alapértelmezett XST – Xilinx Synthesis Tool, a szimulátor a Xilinx ISim, preferált nyelv a VHDL, míg a szabvány a VHDL-93 a nyelvi konstrukciók támogatásához.
4. Tervösszesítő ablakban (Design Summary) az 1-2 lépésben beállított paraméterek kerülnek listázásra, utólagos ellenőrzésként. Befejezés.
5. **Project → New Source**, vagy baloldali eszköztáron **New Source ikon** segítségével adjunk hozzá a projekthez egy új VHDL modult „`egyenloseg_1_bit`” (vhd) néven. A „Define Modul” ablakban a portlistát hagyjuk üresen, mivel ezeket a forráskódban kívánjuk megadni. Ezután a forráskód szerkesztő ablakba a korábbi példa forráskódját (Feladat_1/b, amely az entitáshoz rendelt két különböző architektúra leírást is tartalmazta) írjuk be, vagy másoljuk be. Természetesen itt a tervhez már meglévő HDL fájl(ok) forráskódjai, illetve másolataik is hozzáadhatók külső fájl(ok)ból. Jelen esetben a top-level (későbbiekben többször használjuk ezt a kifejezést is), azaz a legfelsőbb hierarchia szinten lévő modul az „`egyenloseg_1_bit.vhd`” lesz.
6. A VHDL forrás szerkesztése után, a források ablakban az „`egyenloseg_1_bit.vhd`” -t kijelölve a *Check Syntax* segítségével lehet a nyelvi szintaxis ellenőrzést elvégezni (ezt a későbbiekben más szerkesztett, vagy hozzáadott forrásra is érdemes elvégezni). Amennyiben a *Check Syntax* opció rejtett, a folyamat ablakban, [+] XST Synthesize mellett kattintva megjeleníthető a tényleges hierarchia. A szintaxis ellenőrzés során a transcript ablak fogja az ellenőrzés eredményét mutatni (riportolni az esetleges hibákat, és figyelmeztetéseket stb.). Hiba 

esetén azonnal leáll a futtatás, míg figyelmeztetés 

esetén továbbléphetünk a fordítás következő lépésére. Mindaddig, amíg hiba van valahol a forrásállományban, el kell végezni a korrekciót, majd pedig újból a Szintaxis ellenőrzést (a warning megengedhető, feltéve, hogy a későbbi fordítási lépés során nem pont az okozza a hibát). Ha a szintaxis ellenőrzés során minden rendben volt egy jelenik meg.
   a. Próbaképpen állítsunk be egy direkt hibát a forrásállományban: a **begin** és **end** közötti részen az EQ kifejezést írjuk át A-ra. Ekkor ugyan a nyelvi szintaxis helyes lenne, azonban szintetizálás során mégis hibát kapunk, mivel az A bemeneti irányultságú (mód) objektumhoz rendeltük a jobboldali logikai kifejezést, holott a hozzárendelés bal oldalán csak OUT irányultságú objektum állhatna, tehát a portok irányultsága ellentétes (Hiba: „Object A of mode IN can not be updated”).

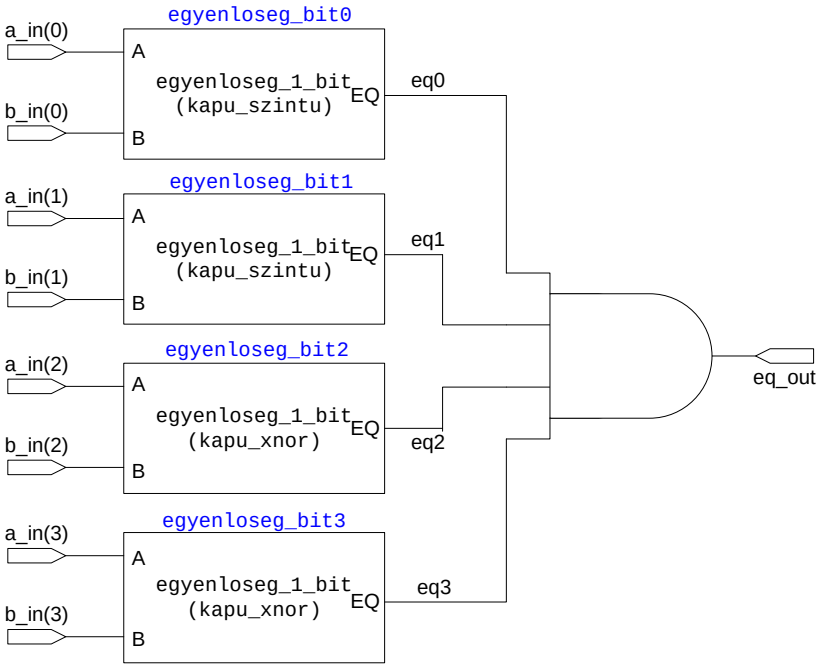
A későbbi 7.) pontban ismertetendő tesztpad létrehozásához majd pedig teszteléséhez elsőként a következő *2.4.3.*, illetve *2.4.4.* pontokban leírt lépéseket kell elvégezni.

2.4.3. Strukturális modell szerinti áramkör felépítése példányosítással

A mai digitális rendszereket legtöbbször kisebb alrendszerekből, logikai részhálózatokból kell összeállítani, amely lehetővé teszi, hogy a komplex rendszereket egyszerűbb egyedi-, vagy előre-defniált komponensekből, modulokból építsük fel (moduláris tervezés „bottom-up” metódika szerint). A VHDL nyelv a strukturális leírások megadását a komponensek beillesztésével, ún. példányosításával biztosítja.

Feladat 1:

Tekintsük a korábbi példánkat (***2.2.2. fejezet, Feladat 1/a. – 1/b.*** **alapján)** melyben 1-bites egyenlőség komparátor áramkört terveztünk. Ezeknek, mint alacsonyabb hierarchia szinten lévő komponensnek a felhasználásával építsünk fel egy 4-bites egyenlőség összehasonlító áramkört. A 4-bites egyenlőség komparátor felépítése a *2.9. ábrán* látható:



2.9. ábra: A 4-bites egyenlőség komparátor (egyenloseg_4bit_top) blokkszintű felépítése

A fenti *2.9. ábrának* megfelelően legyen `egyenloseg_4_bit_top` a neve legmagasabb szintű entitásnak (top-level), amelyben négyszer van példányosítva az `egyenlőség_1_bit` entitás, valamint definiálva vannak a közöttük húzódó belső összeköttetések is.

```
-- Feladat 02
entity egyenloseg_4_bit_top is
   port(
      a_in, b_in: in std_logic_vector(3 downto 0);
      eq_out: out std_logic
   );
end egyenloseg_4_bit_top;

architecture arch of egyenloseg_4_bit_top is
   -- komponens deklarációk VHDL 93 szabványban nem szükségesek,
   -- de érdemes megadni őket a VHDL-87-el való kompatibilitás miatt
```

```
   component kapu_szintu
      port(
         A, B: in std_logic;
         EQ: out std_logic
      );
   end component;

   component kapu_xnor
      port(
         A, B: in std_logic;
         EQ: out std_logic
      );
   end component;

   -- belső jel deklarációk
   -- bitenkenti egyenloseg vizsgalatok kimenetei
   signal eq0, eq1, eq2, eq3 : std_logic;
begin
   -- példányosítások architektúra #1-4
   egyenloseg_bit0: entity work.egyenloseg_1_bit(kapu_szintu)
      port map(A=>a_in(0), B=>b_in(0), EQ=>eq0);
   egyenloseg_bit1: entity work.egyenloseg_1_bit(kapu_szintu)
      port map(A=>a_in(1), B=>b_in(1), EQ=>eq1);
   egyenloseg_bit2: entity work.egyenloseg_1_bit(kapu_xnor)
      port map(A=>a_in(2), B=>b_in(2), EQ=>eq2);
   egyenloseg_bit3: entity work.egyenloseg_1_bit(kapu_xnor)
      port map(A=>a_in(3), B=>b_in(3), EQ=>eq3);
   --példányosított entitások egyedi kimeneteinek ÉS kapcsolata: eredmeny
   eq_out <= eq0 and eq1 and eq2 and eq3;

end arch;
```

A példányosításnak két lépése van: első lépés a komponens specifikáció, azaz hogy milyen entitást, milyen architektúra leírással és milyen (munka – work) könyvtárból szeretnénk példányosítani. A munkakönyvtárban a lefordított entitások és a hozzájuk tartozó architektúrák vannak eltárolva. A második a lépés a port hozzárendelés (port map), amely az aktuális, azaz a top entitásban lévő portokhoz hozzárendeli a formális jeleket, azaz a példányosított komponensek (entitások) jeleit. A fenti példa során a példányosításnál a szabványos VHDL-93 formátumot használtuk [*VHDL93*].

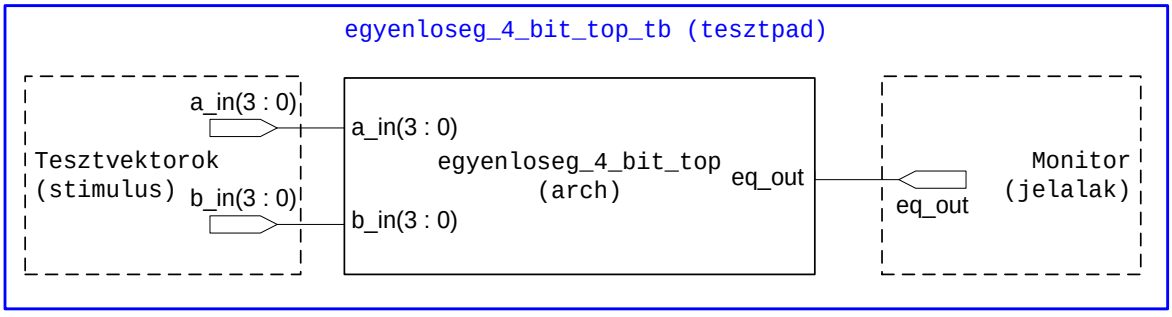
A példánkban (Feladat 1.) az `egyenloseg_bit0`, illetve `egyenloseg_bit1` néven definiált példányokhoz az `egyenloseg_1_bit` nevű entitás `kapu_szintu` architektúra leírása, míg az `egyenloseg_bit2`, és az `egyenloseg_bit3` entitásokhoz a `kapu_xnor` architektúrák lettek hozzárendelve. Az entitás példányok neveinek egyedieknek

kell lenniük. A mostani példánál a 'work' a munka könyvtár nevére utal, ha másként nem állítottuk be ez az alapértelmezett (amely a projektünk alatt található). A munka könyvtár neve és elérhetősége természetesen a könyvtár alapú tervezésnek köszönhetően változhat: konkrét entitás és architektúra munkakönyvtárának nevére és elérési útjára '.' –al elválasztva hivatkozhatunk. A zárójelben megadott architektúra név (`kapu_szintu`, ill. `kapu_xnor`) opcionális: ha nem adjuk meg akkor az utoljára lefordított architektúra kerül példányosításra.

A fenti példa egyben azt is szemlélteti, hogy a blokk diagram (schematic) és a VHDL szintű nyelvi leírások között szoros kapcsolat van. A mai integrált fejlesztő környezetek (mint pl. Xilinx ISE, Altera Quartus stb.) minden esetben tartalmaznak kapcsolási-rajz szerkesztő programrészt is (schematic editor [*XILINK_SCHEMATIC*]), amelyben grafikus felületen keresztül lehet összeállítani a terveket. Ezekből a kapcsolási rajzokból ezután HDL struktúrákat, leírásokat lehet generáltatni, és integrálni, példányosítani, majd pedig szintetizálni az áramköröket a kiválasztott FPGA-ra.

2.4.4. Tervek teszteléséhez tesztpad (test-bench) összeállítása

Manapság sok nagynevű gyártó kínál szimulátor programokat az elektronikus tervek automatizált ellenőrzésének biztosításához, de számos ingyenes szoftver is elérhető ezen az alkalmazási területen. A jegyzet készítése során a Xilinx beépített ISim szimulátorát használjuk, amely az ISE telepítésével válik elérhetővé. Esetünkben inkább a gyors kezelhetőségen, és egyszerűségen, mintsem a professzionális felhasználáson és teljeskörű paraméterezhetőségen volt a hangsúly. Az utóbbiakra számos szimulátor programot találhatunk: az egyik legnagyobb név a Mentor Graphics – ModelSim szimulátora, amelyet professzionális ipari környezetekben használnak (ennek létezik többféle ingyenes oktatói-tanulói változata is – régebbi ISE verziókhoz ModelSim XE-III Starter néven [*XILINX*], amely a Xilinx ISE 12.3-as változatáig együttműködve regisztráció után ingyenesen licenszelhető. Az új ModelSim Student PE változat már a ModelSim oldaláról érhető csak el [*MODELSIM*]).



2.10. ábra: a 4-bites egyenlőség összehasonlító áramkör testpadjának felépítése: stimulus, legfelsőbb szintű terv és a jelalak monitorozó összekapcsolása

A HDL leírásokból összeállított tervek megfelelő működését – akár a tervezés minden egyes hierarchia szintjén – szükséges ellenőrizni, és szimulálni. Vizsgálatok kimutatták, hogy a legtöbb tervezési feladatban – komplexitástól függően – nem az egyes entitások HDL leírása (20-30%), hanem sokkal inkább azok megfelelő tesztelése, egy jó tesztpad/tesztágy összeállítása (~70-80%) jelenti a legnehezebb és egyben legidőigényesebb feladatot, amely a kívánt

kimenet szempontjából egyátalán nem tekinthető közömbösnek. A módszer során a vizsgálandó entitást, vagy a top-modul szintjéig példányosított entitások csoportját egy ún. tesztágyba (testbench) helyezzük el, amelyhez tesztvektorokat, mint gerjesztéseket (stimulus) adhatunk, hogy szimuláljuk a működését. A szimuláció során a kimeneti jeleket monitorozzuk, amelyben grafikusan (általában hullámforma alapján) vagy akár szövegesen (lista) ábrázolva vizsgálhatjuk meg az entitások viselkedését az egyes jeleik változását, illetve beállítását az esetleges hibák felderítése végett. A korábbi példánknál – 4-bites egyenlőség komparátor – maradva mutatjuk be egy tesztpad összeállítását, és működését, valamint a Xilinx ISim szimulátor program használatának bevezetését.

A *2.10. ábra* szerinti tesztpad (test-bench) VHDL leírása a következő kódrészlet szerint adott:

```
-- Feladat 02 - testbench
library ieee;
use ieee.std_logic_1164.all;

entity egyenloseg_4_bit_top_tb is
end egyenloseg_4_bit_top_tb;

architecture tb_arch of egyenloseg_4_bit_top_tb is
   signal test_in0, test_in1: std_logic_vector(3 downto 0);
   signal test_out: std_logic;
begin
   -- a top-level entitás példányosítása uut néven
   uut: entity work.egyenloseg_4_bit_top(arch)
      port map(a_in=>test_in0, b_in=>test_in1, eq_out =>test_out);
   -- testvektorok generálása
   -- Stimulus process
   stim_proc : process
   begin
      -- test vektor 1
      test_in0 <= "0000";
      test_in1 <= "0000";
      wait for 100 ns;
      -- test vektor 2
      test_in0 <= "0000";
      test_in1 <= "0001";
      wait for 100 ns;
      -- test vector 3
      test_in0 <= "0000";
      test_in1 <= "0010";
```

```
      wait for 100 ns;
      -- test vector 4
      test_in0 <= "0000";
      test_in1 <= "0011";
      wait for 100 ns;
      --…
      -- test vector 254
      test_in0 <= "1100";
      test_in1 <= "1111";
      wait for 100 ns;
      -- test vector 255
      test_in0 <= "1110";
      test_in1 <= "1111";
      wait for 100 ns;
      -- test vector 256
      test_in0 <= "1111";
      test_in1 <= "1111";
      wait for 100 ns;

      -- tesztvektor generálás vége

      assert false
         report "Szimulacio befejezodott"
      severity failure;

   end process;
end tb_arch;
```

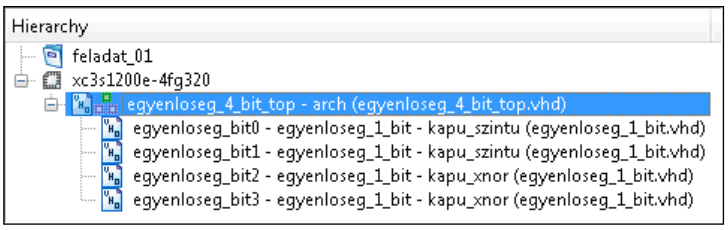
A fenti VHDL kódban a legfelsőbb hierarchia szinten lévő 4-bites összehasonlító áramkör van példányosítva 'uut' (unit under test) néven. A leírás tartalmaz egy `process()`-t, vagy más néven folyamatot is, amelyben szereplő utasítások a hagyományos magas-szintű nyelvekhez hasonlóan egymás után, azaz szekvenciálisan hajtódnak végre (lásd későbbi fejezet). Minden tesztesethez egyedi tesztvektorokat kell definiálni, amelyekhez egy-egy 100 ns-os szimulációs késleltetés tartozik (`wait for 100 ns` hozzárendelés). Ez az utasítás azt adja meg, hogy az előtte lévő hozzárendelések szerint mennyi ideig kell a tesztvektorok aktuális értékit tartani, miután új értéket kaphatnak.

Megjegyzés: Ha a fenti HDL leírásban a 4-bites egyenlőség összehasonlító áramkör helyes működésének vizsgálatához az össze lehetséges tesztvektor variációt meg kívánjuk adni, akkor a lehetséges 256 tesztesetet fel kellene sorolni, amely nagyban megnöveli a tesztpad

forráskódjának méretét, kezelhetőségét és értelmezését. A későbbiek során az ilyen tesztesetekre egy jobb megoldást biztosít, ha n-bites számlálókat (counter) használunk a tesztvektorok generálásához (pl. 1 db 4-bites felfelé számlálót `test_in0`-ra, illetve 1 db 4-bites lefelé számlálót `test_in1`-re), azért, hogy az összes lehetséges tesztvektort előállítsuk.

Az ISE implementációs és modellező feladat további lépései (folytatás)

7. Ezután az 5. lépéshez hasonló módon szerkesszük meg, vagy adjunk hozzá a projekthez egy új VHDL forrást („`egyenloseg_4_bit_top`” leírását) amit a *2.4.2.-ben* ismertettünk (a *2.10. ábrán* látható). Ebben az esetben az ISE forrás ablaka a következő hierarchia szinteket fogja ábrázolni:



Így. az `egyenloseg_4_bit_top.vhd` került a legmagasabb hierarchiaszintre (top modul), amelyben az `egyenloseg_1_bit.vhd` van példányosítva két architektúra leírással: `kapu_szintu`, illetve `kapu_xnor`. Ezután ismételjük meg a 6. lépésben ismertetett szintaxis ellenőrzést a top modulra.

Tesztpad (test-bench) létrehozása:

8. A következő lépésben hozzuk létre a tesztpadot (test bench), amelynek HDL leírását a 2.5.4. során ismertettük. **Project → New Source**, vagy baloldali eszköztáron **New Source ikon** segítségével adjunk hozzá a projekthez egy új „VHDL Test-Bench”-et, melynek neve legyen: „`egyenloseg_4_bit_top_tb`”. A következő ablak rákérdez arra, hogy melyik entitáshoz kívánjuk rendelni a tesztpadot: itt válasszuk ki a legfelsőbb hierarchia szinten lévő „`egyenloseg_4_bit_top`” entitást, hiszen ennek a tesztjét szeretnénk elvégezni. Majd a megjelenő tervösszesítő ablakon nyomjunk „Finish” gombot. Ekkor az Xilinx ISE a top-entitás alapján legenerál egy tesztpad-ot, amelyben a következő VHDL kódrészletnek megfelelően:

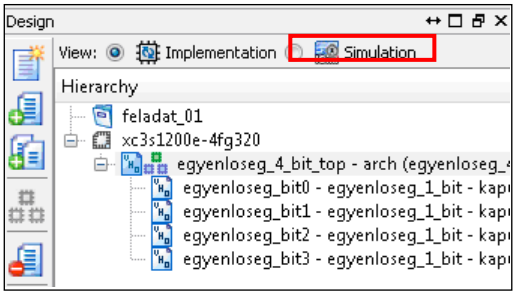
```
-- No clocks detected in port list. Replace <clock> below with
-- appropriate port name
constant <clock>_period : time := 10 ns;
.
.
.
```

```
-- Clock process definitions
   <clock>_process :process
   begin
    <clock> <= '0';
    wait for <clock>_period/2;
    <clock> <= '1';
    wait for <clock>_period/2;
   end process;
.
.
.
wait for <clock>_period*10;
```

9. Mivel órajel periódust, mint egy fizikai típussal deklarált mennyiséget egyelőre nem kívánunk használni, és a tesztpad generálásakor az ISE a top-entitásban sem talált órajelet, ezért ezeket a `<clock>_period` tartalmazó sorokat nyugodtan kommentezzük ki, pontosabban a generált tesztpad forráskódját cseréljük ki a *Feladat 1* (*2.4.3. fejezet*) szerint megadott HDL leírással (`egyenloseg_4_bit_top_tb`). Ezután mentsük el a tesztpad állományt (File → Save) és ellenőrizzük a nyelvi szintaxisát (Process ablak → [+] Synthesize XST → Check Synthax)

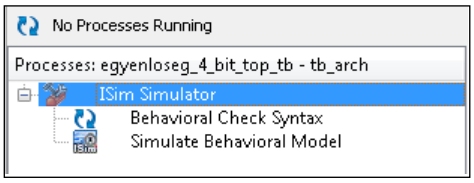
2.4.5. Viselkedési RTL szimuláció Xilinx ISim segítségével

10. a.) Vizsgáljuk meg a 4-bites egyenlőség összehasonlító áramkör működését a Xilinx ISE integrált szimulátorának segítségével (ehhez természetesen használhattuk volna akár a professzionális Mentor ModelSim [*MODELSIM*] szimulátor XE-III verziójú ingyenes változatát is, valamint más külső gyártók ingyenes, vagy fizetős termékeit). A VHDL entitások RTL szintű szimulációjának végrehajtásához válasszuk ki a Design ablak → View nézetében az 'Implementation' (terv megvalósítás) helyett most a 'Simulation' (szimulációt) opciót. Ekkor a megjelenő ISim ikon szimbolizálja az ISE beépített, integrált ISim szimulátorának kiválasztását. A tesztelni kívánt entitás legyen: `egyenloseg_4_bit_top`):

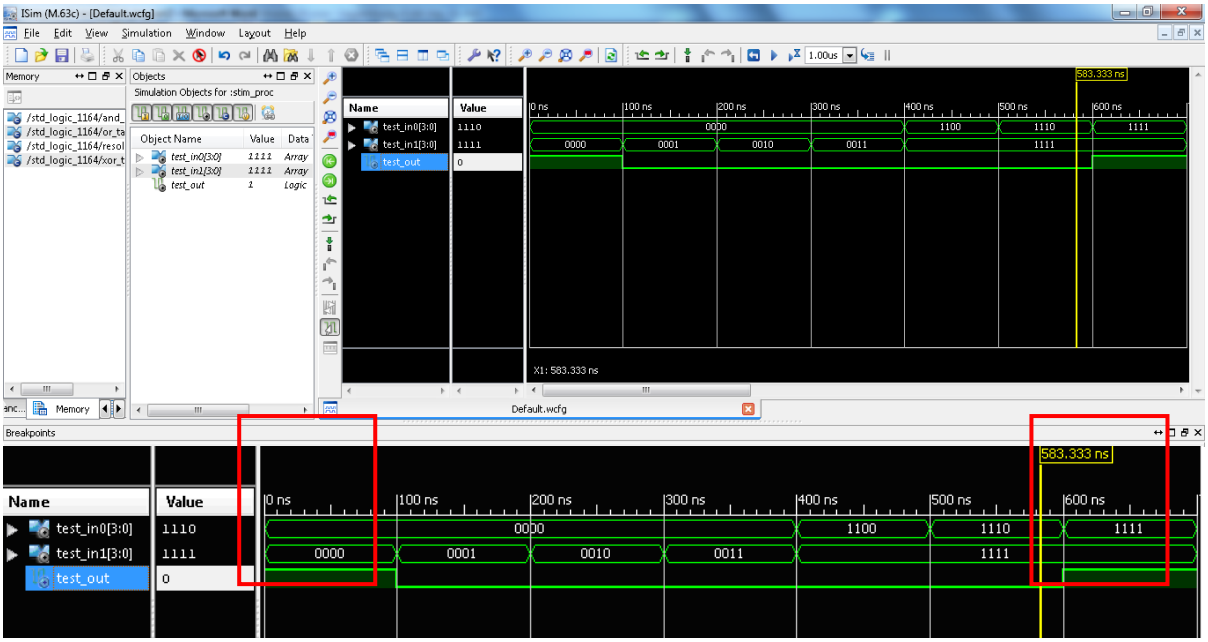


b.) Ezután a szimulátorhoz kapcsolódó Process ablakban válasszuk ki az ISim szimulátort.

Ha szükséges, a Behavioral Check Syntax-menüponttal indítsuk el a tesztágy forráskódjának nyelvi ellenőrzését is. A 'Simulate Behavioral Model' indítja el végül az ISim szimulátorát.



c.) A Xilinx ISim elindítása után a hullámforma jelalakokat ábrázoló grafikus ablak fogad bennünket (test_in0, illetve test_in1 4-4 bites tesztvektorok, tárolják az összehasonlító áramkör 4-bites operandusainak tesztérték-kombinációit, míg a test_out a kimenet):



2.11. ábra: Xilinx ISim szimulátor használata az elkészült VHDL tesztpad leírásának viselkedési RTL szimulációjára (bekeretezve az azonos logikai bemenetekre adott válaszok)

Szimuláció kiértékelése: ahogy elvártuk, ha a 4-bites `test_in0` értéke, bináris bitmintázata teljesen azonos a `test_in1` bitmintázatával, akkor a `test_out` kimeneten egy '1'-es jelenik meg (pirossal bekeretezve), jelezve a két bitminta azonosságát. Minden más esetben a `test_out` kimenetre '0'-t kell, hogy kapjunk. Továbbá látható, hogy a 4-bites bemeneti (a_in, b_in) minták jelváltásai 100 ns-os időegységenként történnek, ahogyan ez a tesztpad VHDL leírásában adott volt. A Xilinx ISim szimulátor használatának részletes leírása túlmutat jelen jegyzet ismertetésén, használata a [*ISIM*] referenciában adott. Azonban megjegyeznénk, hogy minden tervezési feladat során nagyon fontos lépés a HDL leírások viselkedési szimulációja, tehát mielőtt a kiválasztott FPGA-s eszközre a konfigurációs fájl előállítjuk, és letöltenénk, mindenképpen meg kell bizonyosodnunk az áramköri terv helyes működéséről.

2.4.6. Kényszerfeltételek megadása, szintézis és implementáció:

11. Amennyiben a 4-bites egyenlőség összehasonlító áramkört FPGA-ra szeretnénk szintetizálni, és már a 10. lépésben ismertetett helyes működéséről megbizonyosodtunk, következhet a letölthető konfigurációs file (bitstream) generálása. Ezt a lépést következő 3 fázisban lehet elvégezni:

- Tervezői kényszerfeltételek megadása (user constraints),
- Szintézis (synthesis),
- Implementáció (implementation).

a.) A tervezői kényszerfeltételek rögzítik azokat a megkötéseket az áramkörök szintéziséhez és implementálásához, amelyekkel egyrészt speciális jelekhez (pl. órajelhez), illetve a be-, és kimeneti jelekhez (pl. adat-, cím-, vezérlési információ) konkrét fizikai lábakat rendelhetünk. Ilyenek lehetnek órajelek esetén, például a periódusidő, vagy kitöltési tényező, illetve az általános ki-, bemeneti jelek esetén olyan fizikai paraméterek, mint például I/O láb lokalizáció, I/O szabvány, meghajtó képesség, maximális jelváltozási sebesség (slew rate), vagy akár a le-, illetve felhúzó ellenállások beállításai. A láb-hozzárendeléseket és elnevezéseket mindig a legfelsőbb hierarchiaszinten lévő (jelen példánál maradva `egyenloseg_4_bit_top`) entitás I/O port listájában szereplő lábakhoz és elnevezéseikhez kell rögzíteni, különben az implementációs során hibaüzenetet kapunk és leáll a fordítás. Az is általánosan elmondható, hogy az FPGA-s fejlesztő kártyákon megtalálható perifériák IO jelei fizikailag is mindig a kiválasztott FPGA eszköz lábaihoz csatlakoznak, tehát ezek paramétereinek beállításánál mindig nagyon körültekintően kell eljárni.

Szerencsés esetben – ez igaz a legtöbb mai FPGA-s fejlesztőkártyák gyártóira – a lábak hozzárendelései, paraméterei rendelkezésre állnak egy letölthető tervezői fájl, ún. user constraint állományban összegyűjtve. Ez Xilinx FPGA esetén .ucf fájlkiterjesztéssel adott. A jegyzetünkben szereplő Digilent Nexys-2 kártyához is letölthető a gyártó által összeállított kényszerfeltételeket tartalmazó .ucf fájl (Nexys2_1200General.ucf néven) [*NEXYS2*]. Itt jegyeznénk meg, hogyha tervezési, vagy oktatási célokból a kisebb Spartan3E-500K (XC3S500E) FPGA-t tartalmazó Nexys-2 kártyát választjuk, akkor annak az I/O hozzárendelései és beállításai szintén megtalálhatóak a gyártó által kiadott Nexys2_1200General.ucf fájlban (néhány LED lábkiosztása került fizikailag más helyre az FPGA-n). Konkrétan, a láb hozzárendelések közötti egyetlen különbség a led<7:4> esetén van. Kényszerfeltételek megadása a következő lépésekből áll:

- Mivel a gyári .ucf fájlban megadott `Led<0>` kívánjuk felhasználni az `eq_out` kimeneti jel bekötésére: ha ’1’ világít a legjobboldalibb LED(0), ha ’0’, nem világít. Ezért az eredeti .ucf fájlban nevezzük át a `Led<0>`-t a legfelsőbb hierarchia szinten lévő entitás I/O port listájában szereplő névre, azat „`eq_out`”-ra.
- ii.) Mivel a hardveres tesztelés során tetszőleges módon kívánjuk változtatni a bemeneti értékeket, ezért a kártyán lévő programozható kapcsolókat (switch) használjuk fel a bemeneti bitminták megadására. Ezért az alsó 4 kapcsolóhoz rendeljük az `a_in(3)`…`a_in(0)` bemeneteket az `sw(3)`…`sw(0)` helyett, míg a felső 4 kapcsolóhoz a `b_in(3)`…`b_in(0)` bemeneteket, az `sw(7)`…`sw(4)` helyett.

- iii.) Tehát a fent ismertetett módon a gyári Nexys2_1200General.ucf-ben megadott láb hozzárendeléseket a következő kényszerfeltételek szerint kell megváltoztatni:

```
# ucf (user constraint) fájl megjegyzése
NET "eq_out"   LOC = "J14"; # Bank = 1, Pin name = IO_L14N_1/A3/RHCLK7,
Type = RHCLK/DUAL, Sch name = JD10/LD0

NET "a_in<0>" LOC = "G18"; # Bank = 1, Pin name = IP, Type = INPUT, Sch
name = SW0
NET "a_in<1>" LOC = "H18"; # Bank = 1, Pin name = IP/VREF_1, Type = VREF,
Sch name = SW1
NET "a_in<2>" LOC = "K18"; # Bank = 1, Pin name = IP, Type = INPUT, Sch
name = SW2
NET "a_in<3>" LOC = "K17"; # Bank = 1, Pin name = IP, Type = INPUT, Sch
name = SW3
NET "b_in<0>" LOC = "L14"; # Bank = 1, Pin name = IP, Type = INPUT, Sch
name = SW4
NET "b_in<1>" LOC = "L13"; # Bank = 1, Pin name = IP, Type = INPUT, Sch
name = SW5
NET "b_in<2>" LOC = "N17"; # Bank = 1, Pin name = IP, Type = INPUT, Sch
name = SW6
NET "b_in<3>" LOC = "R17"; # Bank = 1, Pin name = IP, Type = INPUT, Sch
name = SW7
```

Ha az eredeti Nexys2_1200General.ucf fájlt módosítjuk a fenti tartalommal, akkor elegendő az ISE projekthez hozzáadni (**Project → Add Source,** vagy Add Source  ikon az eszközsoron). Alternatív megoldásként **Project → New Source**, vagy baloldali eszköztáron **New Source ikon**  segítségével is hozzáadhatunk a projekthez egy új forrást, amelyben a tervezői kényszerfeltételeket (Implementation Constraints File) rögzítjük: adjuk meg fájlnévnek az „egyenloseg_4_bit_top.ucf” . Majd az így létrejött állományba másoljuk be a fenti láb hozzárendeléseket. (Megjegyeznénk, hogy a kényszerfeltételeket tartalmazó .ucf fájlt nem csak szöveges szerkesztőben, hanem a Xilinx ISE grafikus Floorpan I/O Editor-ában is módosítani lehet.)

További megjegyzés: Lehetőség van arra is, hogy a Nexys2_1200General.ucf fájlban minden IO lábhoz tartozó hozzárendelést meghagyjunk (kikommentezni #, és kitörölni sem szükséges). Ha az éppen aktuálisan használt lábak nevét megváltoztatjuk, de a többi nem használt lábhozzárendelést meghagyjuk az .ucf fájlban, amelyhez a top-entitás szintjén nem adtunk meg IO portot, akkor az ISE a fordítás során hibaüzenetet ad, és nem lép tovább az implementálás lépéseire. Ezt a hibát úgy lehet kiküszöbölni, hogy a Process ablakban → [jobb gomb] Implement Design → Properties... és az „Allow Unmatched LOC Constraints” tulajdonságot ki kell „pipálni”. Ekkor nem számít, hogy van-e olyan IO hozzárendelés, amely a top-modul szintjén nincsen deklarálva, a fordítási lépéseket az ISE akkor is végrehajtja.

b.) „Logikai" szintézis során a VHDL entitásokból és architektúra leírásokból kialakított áramkör logikai kapu szintű (pl. LUT, FFs, stb.) megvalósítása történik több lépésben: HDL fordítás → terv hierarchikus analízise → HDL analízis és szintézis. A Process ablakban → XST = Xilinx Szintézis Eszköz segítségével indíthatjuk el az FPGA-ra történő szintetizálás folyamatát.

c.) Implementáció során az előző lépésben szintetizált HDL leírásokból egymást követő lépések sorozatával fizikai leképezést, elhelyezést, majd pedig összeköttetést valósítunk meg a kiválasztott FPGA eszközön (esetünkben Spartan-3E-500K / 1200K) rendelkezésre álló fizikai erőforrásokon. Implementációs tervezési lépések indításához a Process ablakban → 'Implement design' opcióra kell duplán kattintani. Ennek pontos lépéseit, TRANSLATE → MAP → PLACE&ROUTE fázisokat már a korábbi *2.1.5. fejezetben* tárgyaltuk.

A tervezés egyes fázisairól egy-egy összefoglaló táblázat (Design Summary) jelenik meg a következő *2.12. ábra* szerint:

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	3	17,344	1%	
Number of occupied Slices	2	8,672	1%	
Number of Slices containing only related logic	2	2	100%	
Number of Slices containing unrelated logic	0	2	0%	
Total Number of 4 input LUTs	3	17,344	1%	
Number of bonded IOBs	9	250	3%	
Average Fanout of Non-Clock Nets	1.00			

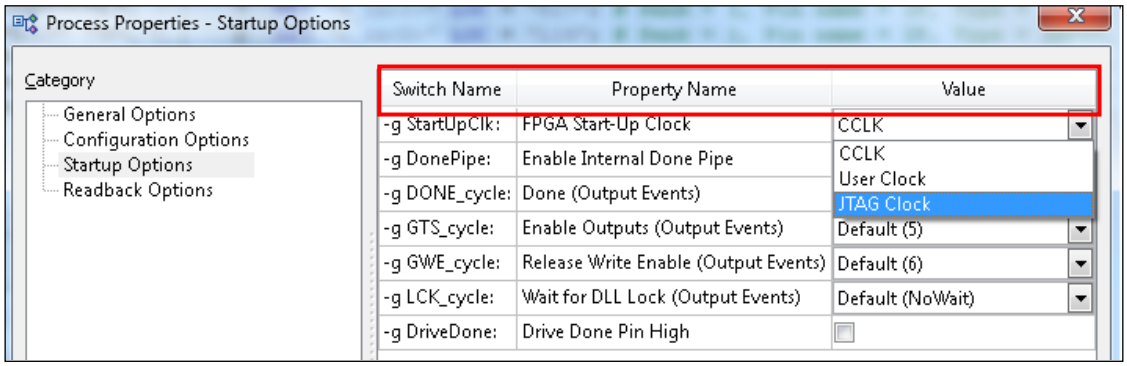
Performance Summary				[-]
Final Timing Score:	0 (Setup: 0, Hold: 0)	**Pinout Data:**	Pinout Report	
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report	

2.12. ábra: A fordítási lépések végén megjelenő összesítő táblázat

A fenti összefoglaló táblázatban a legfontosabb paraméterek a kiválasztott FPGA eszköz erőforrásainak kihasználtságát (utilization), illetve a fizikai összeköttetések (routing results) és az időzítési késleltetéseket (Setup/ Hold time értékek: lehetőleg '0' legyen) összegzik. Ahogyan azt a jelenlegi példánk HDL forráskódjában is megadtuk egy 4-bites `a_in`, 4-bites `b_in` bemenet, valamint egyetlen `eq_out` kimenet lett definiálva, azaz összesen 9 db (4+4+1) fizikailag bekötött (bounded) I/O blokk összesítését vártuk, és egyben kaptuk meg. Ezek a rendelkezésre álló Spartan3E-500 FPGA 250 I/O blokkjainak egy kis részét, mindösszesze 3%-át teszik ki.

d.) Utolsó lépésként **állítsuk elő a konfigurációs bitfájlt** a Process ablak – Generate programming file opciójával. De előtte még az `egyenloseg_4_bit_top` top-level entitást kiválasztva [jobb gomb] → Generate Programming File → Design properties tulajdonságai között, a Startup Options kategória mellett a Startup Clock órajelet állítsuk át „JTAG clock"-ra (az alapérték CCLK volt). Ez azért nagyon fontos beállítás, mivel a JTAG órajel fogja specifikálni azt a jelet, amely segítségével a konfigurációs folyamat végén a kezdeti szekvencia információkat (startup-sequence) betöltjük (Boundary Scan – JTAG). Ellenkező esetben

a generálás végeztével figyelmeztető üzenetet kapnánk a konfiguráció során. (Itt jegyeznénk meg, hogy a legtöbb kapcsolót az ISE parancs soros üzemmódjában is beállíthatjuk a 'switch name' alapján, pl: -g StartUpClk.)



Az így kapott generált bitfájl (`egyenloseg_4_bit_top.bit`) lesz a kiválasztott FPGA eszközre letölthető formátumú konfigurációs állomány. A generált bitfájl neve mindig a legfelsőbb hierarchia szinten lévő entitás nevével fog megegyezni.

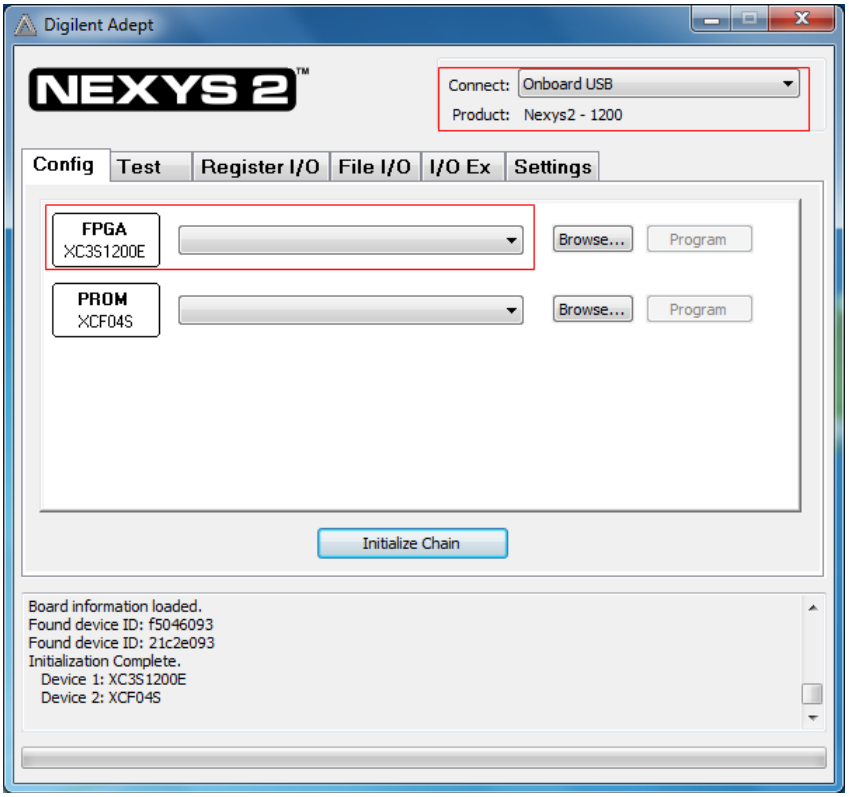
12. A következő lépésben a teszteléshez az FPGA-ra az előállított bit-szintű konfigurációs állományt le kell tölteni. Ehhez a Digilent-es termékekhez biztosított Digilent Adept Suite nevű programot (és nem az ISE keretrendszer Xilinx Impact programját!) használjuk. Az Adept Suite egy olyan alkalmazás, amely az FPGA-kon kívül, a konfigurációs bitfájlok letöltését (.bit, .mcs) támogatja CPLD, PROM / Flash típusú eszközökre, vagy akár adatok átvitelét a szabványos USB porton keresztül, illetve speciális célú regiszterek írását és olvasását. (Megjegyenénk, hogy a Adept SDK segítségével a PC oldali alkalmazás fejlesztésére is lehetőség nyílik).

a.) Adept indítása: **Programok → Digilent → Adept Suite.**

b.) Csatlakoztassuk a Digilent Nexys-2 kártyát az USB-s roll-on szalagkábelen keresztül a PC-n lévő egyik USB 2.0 foglalatba. Első használatkor a Digilent-es kártya USB meghajtó programjának a telepítését meg kell várni. Ekkor a következő *2.13. ábra* szerint egy „Onboard USB – Nexys2-1200", vagy „Nexys2-500" azonosító felirat kell, hogy megjelenjen az Adept Suite alkalmazás ablakában (pirossal jelölve). Másrészt, ha felismerte a kártyát az „Initialize Chain"-re kattintva lehet az FPGA, illetve a PROM eszközöket felismertetni. Tehát nemcsak az FPGA-t, hanem a Nexys2 kártyán lévő PROM memóriát is lehet konfigurálni.

c.) Az FPGA-s eszköz mellett a Browse… gombra kattintva kell kiválasztani a generált bitfájlt (.bit), amelyet a tervezés 1.) lépésben létrehozott projekt könyvtárban <drive>\feladat_01\ alatt kell keresni, `egyenloseg_4_bit_top.bit` néven.

d.) Végezetül a Program gombra kattintva megtörténik a generált bitfájl programozása az FPGA-s eszközre (ezt jelzi a Nexys2-es kártyán a narancssárga Done LED felvillanása). Ezzel a tervezési feladatot befejeztük. ☺



2.13. ábra: Digilent Adept Suite letöltő program grafikus ablaka

Megjegyzés: az FPGA-s kártyán lévő perifériákat az első használat előtt mindenképpen érdemes letesztelni. Erre is lehetőséget ad a Digilent Adept Suite program, amelynek Test fülére kattintva a következő periféria teszteket tudjuk elvégezni:

- RAM
- Flash
- Kapcsolók
- Nyomógombok
- LED-ek

A tesztprogram egy Xilinx EDK-ban elkészített konfigurációs bitfájlt is letölt, amely PWM működtetéssel teszteli a 7-szegmenses kijelzőket, és a VGA kimenetet is.

A továbbiakban a jegyzetben ismertetésre kerülő FPGA alapú tervezési feladatok megvalósításánál is ezeket az 1–12.) részletezett lépéseket célszerű követni. Az ISE használatának további hátteréről egyrészt az ISE Quick Start Tutorialban (ISE Help menü), másrészt az ISE In-Depth Tutorial-ban lehet olvasni.

További gyakorló feladatok

1.) A fejezetben ismertetett 1–12.) lépéseket felhasználva tervezzen VHDL nyelven egy olyan többkapus logikai hálózatot (`multi_gates.vhd` néven), amelyben AND, OR, NAND, NOR, XOR, XNOR kapuk vannak felhasználva. A hálózatnak legyen két bemenete, `a_in`, illetve

`b_in` és annyi kimenete (6) ahány logikai kaput integráltunk: ezek nevei rendre `and_out, or_out, nand_out, nor_out, xor_out,` illetve `xnor_out` legyenek.

a.) Tervezzen egy tesztpadot az előző VHDL leíráshoz, `multi_gates_tb.vhd` néven. Szimulálja le a viselkedését a Xilinx ISim segítségével.

b.) Az FPGA-s eszközön az elkészült terv verifikációjához használjon programozható kapcsolókat, ahol `sw(0)=a_in`, illetve `sw(1)=b_in`, míg a kimeneti jelek vizsgálatához a LED-eket (`Led(5:0)`) a hat különböző kapu kimeneteihez rendelve.

c.) Szintetizálja, implementálja, majd pedig generálja le a konfigurációs bitfájl, végül pedig töltse le a kiválasztott FPGA-ra. A letöltéshez használja a Digilent Adept Suite nevű programját. Ellenőrizze a terv helyes működését a kártyán.

2.) A fejezetben ismertetett 1–12.) lépéseket felhasználva és a korábbi Feladat-1 példánkat kiegészítve tervezzen meg VHDL nyelven egy olyan áramkört, amely az 4-bites egyenlőség összehasonlításán kívül az egyenlőtlenségeket (A kisebb, mint B; illetve A nagyobb, mint B) is képes jelezni. A 3-kimenetű logikai hálózat (`comparator_4_bit.vhd`) tervezése során tisztán logikai kapukat használjon: AND, OR, NAND, NOR, XOR, XNOR (ne használja az egyenlőség, egyenlőtlenségek relációs operátorait). A hálózatnak legyen két 4-bites bemenete, `a_in(3:0),` illetve `b_in(3:0)` és 3-kimenete, amelyek nevei rendre `eq_out, gt_out, lt_out` legyenek (gt=greater than: nagyobb, mint, lt=less than: kisebb, mint).

a.) Tervezzen egy tesztpadot az előző VHDL leíráshoz, `comparator_4_bit_tb.vhd` néven. Szimulálja le a viselkedését a Xilinx ISim segítségével.

b.) Az FPGA-s eszközön az elkészült terv verifikációjához használjon programozható kapcsolókat, ahol `sw(3:0)=a_in(3:0),` illetve `sw(7:4)=b_in(3:0),` míg a kimeneti jelek vizsgálatához a LED-eket, `Led(2:0)` = (`gt_out/lt_out/eq_out`) a három különböző összehasonlítás eredményéhez rendelve.

c.) Szintetizálja, implementálja, majd pedig generálja le a konfigurációs bitfájl, végül pedig programozza fel a kiválasztott FPGA-ra. A letöltéshez használja a Digilent Adept Suite nevű programját. Ellenőrizze a terv helyes működését a kártyán.

3.) A fejezetben ismertetett 1–12.) lépéseket felhasználva és tervezzen meg VHDL nyelven egy 3-8-as dekóder áramkört, amely engedélyező bemenettel is rendelkezik. A 8-kimenetű, „one-hot'" kódolást használó logikai dekódoló áramkör (`decoder_3_8_kapu.vhd`) tervezése során tisztán logikai kapukat használjon: AND, OR. A hálózatnak legyen egy 3-bites bemenete, `a_in(2:0),` és 8-kimenete `decode_out(7:0)` néven. Az áramkör tervezéséhez nyújt segítséget a *2.8. táblázat*:

2.8. táblázat: 3-8 dekóder áramkör, engedélyező bemenettel

bemenetek				**dekódolt kimenet**
en_in	**a_in(2)**	**a_in(1)**	**a_in(0)**	**decode_out(7:0)**
0	-	-	-	0000 0000
1	0	0	0	0000 000**1**
1	0	0	1	0000 00**1**0
1	0	1	0	0000 0**1**00
1	0	1	1	0000 **1**000
1	1	0	0	000**1** 0000
1	1	0	1	00**1**0 0000
1	1	1	0	0**1**00 0000
1	1	1	1	**1**000 0000

a.) Tervezzen egy tesztpadot az előző VHDL leíráshoz, `decoder_3_8_kapu_tb.vhd` néven. Szimulálja le a viselkedését a Xilinx ISim segítségével.

b.) Az FPGA-s eszközön az elkészült terv verifikációjához használjon programozható kapcsolókat, ahol `sw(2:0)=a_in(2:0)`, illetve `sw(3) = en_in`, míg a kimeneti dekódolt jelek vizsgálatához a LED-eket, `Led(7:0) = decode_out(7:0)` rendeljen.

c.) Szintetizálja, implementálja, majd pedig generálja le a konfigurációs bitfájl, végül pedig programozza fel vele a kiválasztott FPGA-t. A letöltéshez használja a Digilent Adept Suite nevű programját. Ellenőrizze a terv helyes működését a kártyán.

2.5. Strukturális és viselkedési áramköri modellek megvalósítása

Ebben a fejezetben a korábban ismertetett szekvenciális és egyidejű hozzárendelési utasításokat használjuk fel ahhoz, hogy az áramkörök VHDL leírásait implementáljuk két lehetséges módon: viselkedésük, illetve strutúrális leírásuk megadásával. Viselkedési modell esetén nem a belső felépítésüket, hanem sokkal inkább a sorrendi működésüket, funkciójukat kívánjuk definiálni. Míg a másik, strukturális esetben a viselkedés helyett a belső felépítésükre, azok összeköttetéseire vagyunk kíváncsiak. Egy harmadik modellezési szempont lehetne az időzítési modell, amely főként azon alapul, hogy a modellezett áramköri rendszer a bemeneteire adott gerjesztésekre milyen módon reagál, illetve a gerjesztések további változásaira hogyan működik. Ez utóbbi szemléletmódot a szimulációs környezet összeállítása során már megismerhettük (*2.4.4.–2.4.5. fejezetek*). Jelen fejezet-részben a szekvenciális és egyidejű hozzárendelések alapjait vizsgálunk meg néhány szemléltető példán keresztül.

A VHDL leíró nyelvre jellemző az a képesség, hogy támogatja egy digitális rendszer tisztán viselkedési, tisztán struktúrális modellezését, valamint ezek kombinációját is. Tehát könnyen megvalósítható olyan komplex áramköri rendszer, amelyet strukturális leírással kisebbekre bonthatunk (top-down stílus), de amelyeket belül, az alacsonyabb szinteken már viselkedési leírással modellezünk.

2.5.1. Strukturális áramköri modellek

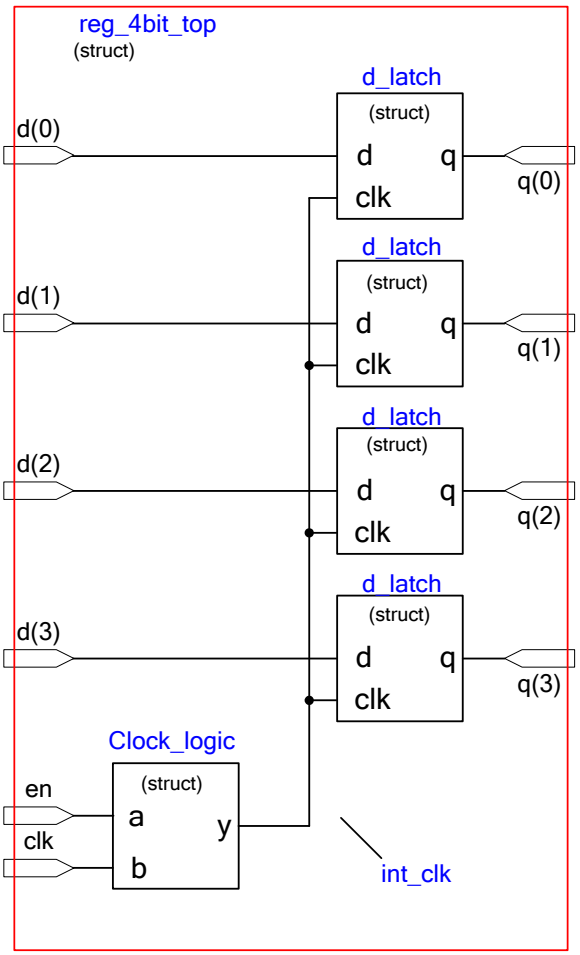
A strukturális modellek esetén a blokkok teljes belső felépítését hierarchikus szinteken keresztül adjuk meg. A tervezési metodika történhet a hagyományos bottom-up vagy top-down stílus szerint. A modell és hierarchikus alrendszerei közötti kapcsolatot – a viselkedési modellhez hasonlóan – itt is speciális irányított változók, az ún. portok teremtik meg (`in`- bejövő, `out` –kimenő, `inout` – kétirányú adatáramlás).

Feladat 1: 4-bites regiszter strukturális áramköri modellje, szimulációval

Tervezzünk meg egy elemi D- tárolókból (FlipFlop) felépített 4-bites párhuzamos betöltésű, és párhuzamos kiolvasású regisztert strukturális modellt alkalmazva, és bottom-up tervezési metodikát használva (*2.14. ábra*). A bemeneteket jelöljük `d(i)`-vel, a kimeneteket `q(i)`-vel). Az órajel engedélyező logika egy ÉS kapu, amely az `en` engedélyező jel hatására juttatja az ún. kapuzott órajelet `clk` az egyes elemi D-tárolók órajel bemeneteire. A top modul neve legyen `reg_4bit_top.vhd`. Használjunk szekvenciális hozzárendelő utasításokat.

A fenti ábra szerinti 4-bites regiszter összeállításához, először a legalacsonyabb hierarchia szinten meg kell tervezni az órajel logikát, illetve az elemi 1-bites D-tárolókat, amelyeket azután a következő egyel magasabb hierarchia szinten példányosítani fogunk.

Az egy bites elemi tároló neve legyen `d_ff`, az architektúra leírás neve pedig legyen `struct`. Ennek VHDL kódja látható a következő példán:



2.14. ábra: 4-bites párhuzamos betöltésű és kiolvasású regiszter strukturális modellje

```
-- D tároló
library IEEE;
use IEEE.std_logic_1164.all;

entity d_ff is
  port ( d, clk : in std_logic;
         q : out std_logic );
end d_ff;

architecture struct of d_ff is
begin
  ff_inst : process (clk, d) is
  begin
    if clk'event and clk = '1' then
      q <= d after 2 ns;   --egyedi késleltetés modellezése: aktuális időponthoz --képest 2ns múlva fogja felvenni a d értékét a q
    end if;

  end process ff_inst;
end architecture struct;
```

A fenti kódrészletben a process törzsének a végére betehetjük a következő sort is:

```
wait on clk, d;
```

és helyettesítjük, hogy a clk, és d bemeneti jeleket a process érzékenységi listájába nem közvetlenül tesszük.

Az órajel engedélyező logika VHDL forráskódja a következő:

```
-- D flipflop
library IEEE;
use IEEE.std_logic_1164.all;

entity clock_logic is
  port ( a, b : in std_logic;
            y : out std_logic );
end clock_logic;

architecture struct of clock_logic is
begin
  clock_logic_inst: process (a, b) is
  begin
    y <= a and b after 5 ns; --egyedi késleltetés modellezése: aktuális
-- időponthoz képest 5 ns múlva fogja felvenni a d értékét a q
    end process clock_logic_inst;
end architecture struct;
```

A következő, eggyel magasabb hierarchia szinten lévő `reg_4bit_top` entitásban a fenti *2.14. ábrának* megfelelően kell példányosítani az egyes entitásokat (4x `d_ff` , 1x `clock_logic`).

```
-- órejel logika
library IEEE;
use IEEE.std_logic_1164.all;

entity reg_4bit_top is
  port ( d : in std_logic_vector(3 downto 0);
         q : out std_logic_vector(3 downto 0);
         clk : in  std_logic;
         en : in  std_logic );
end reg_4bit_top;
```

```
architecture struct of reg_4bit_top is
  signal int_clk : std_logic;  --belső ún. kapuzott óra jel!
begin
  --port map VHDL 93 szabvány szerint, nem kell component
  bit0 : entity work.d_ff(struct)
    port map (d(0), int_clk, q(0));
  bit1 : entity work.d_ff(struct)
    port map (d(1), int_clk, q(1));
  bit2 : entity work.d_ff(struct)
    port map (d(2), int_clk, q(2));
  bit3 : entity work.d_ff(struct)
    port map (d(3), int_clk, q(3));
  gate : entity work. clock_logic (struct)
    port map (en, clk, int_clk);
end architecture struct;
```

A következő lépésben ehhez a magas szintű entitáshoz (`reg_4bit_top`) készítsünk teszt-pad-ot, és ágyazzuk be, amelynek a neve legyen `reg4_test_bench` . Az testbench-nek a VHDL forrását a következőképpen adjuk meg.

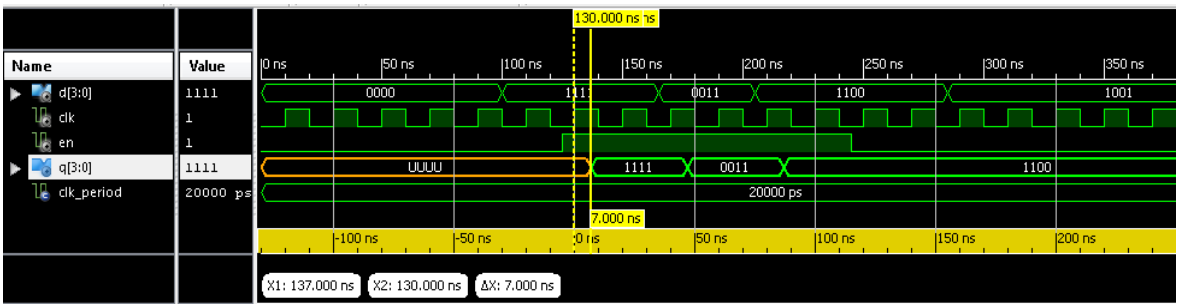
```
-- tesztágy 4-bites regiszterhez
library IEEE;
use IEEE.std_logic_1164.all;

entity reg4_test_bench is
end entity reg4_test_bench;

architecture test of reg4_test_bench is
    -- komponens deklarációk a teszteléshez
    COMPONENT reg_4bit_top
    PORT(
         d : IN  std_logic_vector(3 downto 0);
         q : OUT  std_logic_vector(3 downto 0);
         clk : IN  std_logic;
         en : IN  std_logic
        );
    END COMPONENT;
   --belső jelek bemenetekhez
   signal d : std_logic_vector(3 downto 0) := (others => '0');
   signal clk : std_logic := '0';
```

```
   signal en : std_logic := '0';
   --belső jelek kimenetekhez
   signal q : std_logic_vector(3 downto 0);
   --belső órajel, szimulációhoz
   constant clk_period : time := 20 ns;  -- 50MHz órajel szimulátorhoz,
   -- time: fizikai típus deklarációval megadott mennyiség

begin
  -- Unit Under Test: a tesztelni kívánt top-level entitás példányosítása,
  -- amit előtte a komponens deklarációs részen megadtunk
  uut: reg_4bit_top PORT MAP (
          d => d,
          q => q,
          clk => clk,
          en => en
        );

  -- Clock process definíciók
   clk_process :process
   begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
   end process;

  -- gerjesztések definíciói
  stimulus_proc : process
  begin

    -- kezdetben ún. reset állapot 100ns-ig, jelenleg nem adtunk meg reset
    -- jelet
    wait for 100 ns;

    d <= "1111";
    en <= '0';
    wait for 25 ns;

    en <= '1';
```

```
    wait for clk_period*2;

    d <= "0011";
    wait for clk_period*2;

    d <= "1100";
    wait for clk_period*2;

    en <= '0';
    wait for clk_period*2;

    d <= "1001";
    wait for clk_period*2;
    -- . . . további esetek
    wait;
  end process stimulus_proc;
end architecture test;
```

A megfelelően példányosított és a tesztpadba beágyazott entitásokra gerjesztést kell adni, hogy meg tudjuk vizsgálni a kimenetek `q(3:0)` változásait. Természetesen a fenti kódban a `std_logic_vector(3 downto 0)` adattípust is deklaráltuk úgy, hogy a gerjesztés során először `'1111'`, majd pedig engedélyezve `'0011'`, `'1100'` és végül `'1001'` bitmintázatokat rendeltünk. A szimulációban megjelenik a gerjesztő folyamat mellett egy másik, órajel generáló process() is, amely a peridódus idő feléig ’0’-át, majd a másik felében ’1’-et szolgáltatva váltakozik. Ez a processz szükséges az `clk` órajel szimulációjához, hisszen ehhez rendeltük a VHDL leírásban a `d_ff` tárolását, felfutó élre vezérelve. A `clk` órajel impulzusidejének beállításához a konstans fizikai (’time’) típus deklarációt használunk, amelynek periódus ideje 20 ns. A VHDL leírásszimulációhoz a Xilinx ISim szimulátor használható, mellyel kapcsolatban bővebb leírást a *2.4. fejezetben* találhatunk.

A szimuláció során a következő hullámforma alakot kapjuk (*2.15. ábra*):



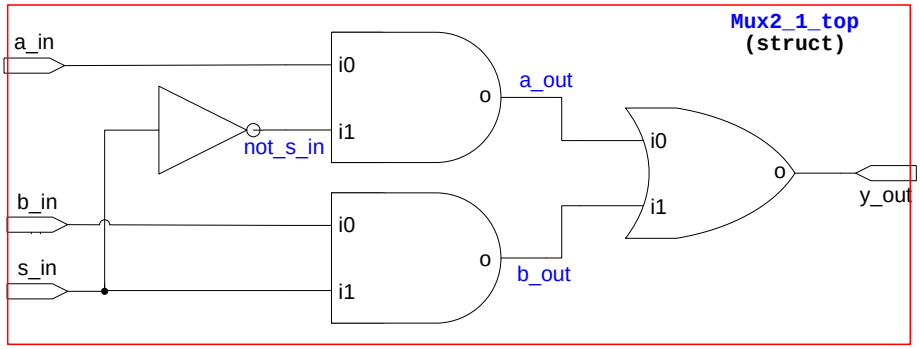
2.15. ábra: `reg_4bit_top` szimulációja során kapott hullámforma

Szimulációs eredmény részletes vizsgálata: látható, hogy a `'0000'` bitmintázat hiába adott a szimuláció 0 ns-os idejétől kezdve, mivel nem érkezik engedélyező jel (`en = '1'`), ezért az

órajelet sem kapuzza be az egyes D-tárolók bemeneteire a 4-bites regiszterben. Az `'1111'` bitmintázat 100 ns –nál érkezik, de az engedélyező még mindig inaktív marad (`en = '0'`). Az engedélyező jel 125 ns-nál vált '1'-be, azaz a következő felfutó órajel hatására (`clk'event and clk = '1'`) a bemeneten `d(3:0)` lévő `'1111'` bitmintázat a 4-bites regiszterbe kerül, és 125+7 ns múlva jelenik meg a tároló `q(3:0)` kimenetein (ezt jelzi a két sárga ún. marker közötti időkülönbség). A 7ns-os időkésleltetés a példányosított `d_ff` -ek beállított 2 ns-os késleltetéséből, illetve az órajel logika (`and`) 5 ns-os késleltetésének összegéből adódik. Ezután a következő érvényes `'0011'` bitmintázat 150 ns időben érkezik, amely 150+7 ns múlva jelenik meg a q kimeneteken. A tesztelés során gerjesztésként beállított utolsó `'1001'` bitmintázat az inaktív engedélyező jel (`en = '0'`) miatt már nem kerül a 4-bites regiszterbe.

Feladat 2. 2-1 multiplexer strukturális modell szerinti egyidejű értékadással felépítve

Tervezzünk meg egy 2-1 MUX áramkört, amely egyidejű (konkurens) hozzárendelési utasításokat és strukturális áramköri VHDL leírást használ. A 2-1 multiplexer áramkör strukturális felépítése a következő *2.16. ábrán* látható. Az áramkör bemeneteit jelölje `a_in, b_in;` kiválasztó jelét `s_in`;illetve kimenetét `y_out`. A legfelsőbb szintű entitás neve legyen `mux2_1_top.`



2.16. ábra: 2-1 Multiplexer áramkör strukturális modellje

Az áramkör strukturális leírásának VHDL forrása a következő:

```
-- 2-to-1 MUX kapu szintű strukturális leírása
library IEEE;
use IEEE.std_logic_1164.all;
entity mux2_1_top is
port(
 a_in : in STD_LOGIC;
 b_in : in STD_LOGIC;
 s_in : in STD_LOGIC;
 y_out : out STD_LOGIC
);
end mux2_1_top;
```

```
architecture struct of mux2_1_top is
  --belső jelek deklarációja
  signal a_out : STD_LOGIC;
  signal b_out : STD_LOGIC;
  signal not_s_in : STD_LOGIC;
begin
  --konkurrens értékadások logikai operátorok felhasználásával
  a_out <= not_s_in and a_in;
  b_out <= s_in and b_in;
  not_s_in <= not(s_in);
  y_out <= b_out or a_out;
end struct;
```

Gyakorlati feladat: tovább feladatként érdemes ehhez a feladathoz is egy tesztpadot létrehozni, amelybe a legfelsőbb szintű entitást be kell ágyazni. A gerjesztésekre adott válaszokat vizsgálja meg az ISim szimulációs program segítségével, mellyel kapcsolatban bővebb leírást a *2.4. fejezetben* találhat.

További feladatként írjon a modellhez egy tesztpad-ot, és vizsgálja meg a működését ISim szimulátor segítségével.

2.5.2. Viselkedési áramköri modellek

A viselkedési modellek esetén nem a blokkok teljes belső felépítésére, sokkal inkább azok funkcióját, működését kívánjuk megadni. A tervezés itt is történhet hierarchikusan, azaz szintenként csakúgy, mint a strukturális eseteben. A viselkedési modell egy rendszer vagy részrendszer funkcionális leírását tartalmazza. A viselkedési modell legfőbb alkotóelemei a folyamatok (process-ek), amelyek között az adatátvitelt speciális jelek (signal-ok) közvetítik.

Feladat 3: 2-1 multiplexer szekvenciális if-else szerkezettel

Tekintsük a következő igazságtáblázatot, amely **egy 2-1 multiplexer** működését definiálja (`mux2_1_top`).

2.9. táblázat: 2-1 multiplexer működését leíró igazságtábla

bemenetek			multiplexált kimenet
s_in	**a_in**	**b_in**	**y_out**
0	1	0	a_in
1	0	1	b_in

Ahogy látjuk, a kiválasztó s_in jel értékétől függően:

- ha `s_in = '0'`, akkor az `a_in` bemenetet választja ki, és teszi `y_out` kimenetre, míg
- ha `s_in = '1'`, akkor pedig a `b_in` bemenet értéke kerül az `y_out` kimenetre.

Ennek az igazságtáblázatnak megfelelő a viselkedési modellt a következő VHDL kódrészlet tartalmazza, amelyben **if-else** szekvenciális szerkezetet használunk:

```
-- 2-to-1 MUX viselkedési leírása szekvenciális if-else szerkezettel

library IEEE;
use IEEE.std_logic_1164.all;
entity mux2_1_top is
port(
 a_in : in STD_LOGIC;
 b_in : in STD_LOGIC;
 s_in : in STD_LOGIC;
 y_out : out STD_LOGIC
);
end mux2_1_top;

architecture behav of mux2_1_top is
begin
  if_process : process (s_in, a_in, b_in) is
  begin
   if s_in = '0' then
      y_out <= a_in;  -- sel = '0'
   else
      y_out <= b_in; -- sel /= '0'
   end if;
  end process;
end behav;
```

További feladatként írjon a modellhez egy tesztpad-ot, és vizsgálja meg a működését ISim szimulátor segítségével.

Feladat 4: 2-1 multiplexer szekvenciális if-else szerkezettel, generic használatával

Tervezzünk a korábbi ismeretek alapján egy olyan **2-1 multiplexer áramkört**, amelynek bemenetei (a_in, és b_in), illetve kimenete (y_out) N=4 bitesek. Ehhez használjunk generic típust. A *2.9. igazságtáblázatnak* megfelelő a viselkedési modellt a következő VHDL kódrészlet tartalmazza, amelyben **if-else** szekvenciális szerkezetet használunk:

```
-- 2-to-1 MUX viselkedési leírása szekvenciális if-else + generic
--szerkezettel

library IEEE;
use IEEE.std_logic_1164.all;

entity mux2_1_top is
generic(N:integer := 4);
port(
 a_in : in STD_LOGIC_VECTOR (N-1 downto 0);
 b_in : in STD_LOGIC_VECTOR (N-1 downto 0);
 s_in : in STD_LOGIC;
 y_out : out STD_LOGIC_VECTOR (N-1 downto 0)
);
end mux2_1_top;

architecture behav of mux2_1_top is
begin
  if_process : process (s_in, a_in, b_in) is
  begin
   if s_in = '0' then
      y_out <= a_in;  -- sel = '0'
   else
      y_out <= b_in; -- sel /= '0'
   end if;
  end process;
end behav;
```

További feladatként írjon a modellhez egy tesztpad-ot, és vizsgálja meg a működését ISim szimulátor segítségével!

Feladat 5: 4-1 multiplexer szekvenciális if-else szerkezettel

Tervezzünk egy **4-1 multiplexer** áramkört, amely viselkedési VHDL leírással adott, és szekvenciális **if-else** utasításokat használ (`mux4_1_if`). A bementeket jelöljük `d_in0,…,d_in3`-al, amelyek std_logic típusúak legyenek. A kimenetet jelöljük `y_out`-al, amely szintén std_logic típusú, a kiválasztó jelet pedig nevezzük el `sel_in(1:0)`- al, amely std_logic_vector típusú legyen. Ekkor a következő viselkedési leírást adhajtuk meg:

```
-- 4-to-1 MUX viselkedési leírása szekvenciális if-else szerkezettel

library ieee;
use ieee.std_logic_1164.all;

entity mux4_1_if is
port (
 sel_in : in std_logic_vector(1 downto 0);
 d_in0, d_in1, d_in2, d_in3 : in std_logic;
 y_out : out std_logic );
end entity mux4_1_if;

architecture behaviour of mux4_1_if is
begin
 mux_select : process (sel_in, d_in0, d_in1, d_in2, d_in3) is
 begin
    if sel_in = ”00” then
       y_out <= d_in0;
    elsif sel_in = ”01” then
       y_out <= d_in1;
    elsif sel_in = ”10” then
       y_out <= d_in2;
    else
       y_out <= d_in3;
    end if;
 end process mux_select;
end architecture behaviour;
```

Feladat 6: 4-1 multiplexer szekvenciális case-when szerkezettel

Tervezzünk egy **4-1 multiplexer** áramkört, amely viselkedési VHDL leírással adott, és szekvenciális **case-when** utasításokat használ (`mux4_1_case`). A bementeket jelöljük `d_in0,…,d_in3`-al, amelyek std_logic típusúak legyenek. A kimenetet jelöljük `y_out` -al, amely szintén std_logic típusú, a kiválasztó jelet pedig nevezzük el `sel_in(1:0)`-al, amely std_logic_vector típusú legyen. Ekkor a következő viselkedési leírást adhajtuk meg:

```
-- 4-to-1 MUX viselkedési leírása szekvenciális case-when szerkezettel


library ieee;
use ieee.std_logic_1164.all;

entity mux4_1_case is
port (
 sel_in : in std_logic_vector(1 downto 0);
 d_in0, d_in1, d_in2, d_in3: in std_logic;
 y_out : out std_logic );
end entity mux4_1_case;

architecture behav of mux4_1_case is
begin

  case_select : process (sel_in, d_in0, d_in1, d_in2, d_in3) is
  begin
 case sel_in is
 when "00" =>
    y_out <= d_in0;
 when "01" =>
    y_out <= d_in1;
 when "10" =>
    y_out <= d_in2;
 when others =>
    y_out <= d_in3;
 end case;
  end process case_select;
end architecture behav;
```

További feladatként írjon a modellhez egy tesztpad-ot, és vizsgálja meg a működését ISim szimulátor segítségével!

Feladat 7: 4-1 multiplexer egyidejű when-else szerkezettel

Tervezzünk egy **4-1 multiplexer** áramkört, amely viselkedési VHDL leírással adott, és jelen esetben konkurens, egyidejű **when-else** hozzárendelő utasításokat használjunk (`mux4_1_when`). A bementeket a korábbi példákhoz hasonlóan jelöljük `d_in0,…,d_in3`-al, amelyek std_logic típusúak legyenek. A kimenetet jelöljük `y_out`-al, amely szintén std_logic típusú, a kiválasztó jelet pedig nevezzük el `sel_in(1:0)`- al, amely std_logic_vector típusú legyen. Ekkor a következő viselkedési leírást adhajtuk meg:

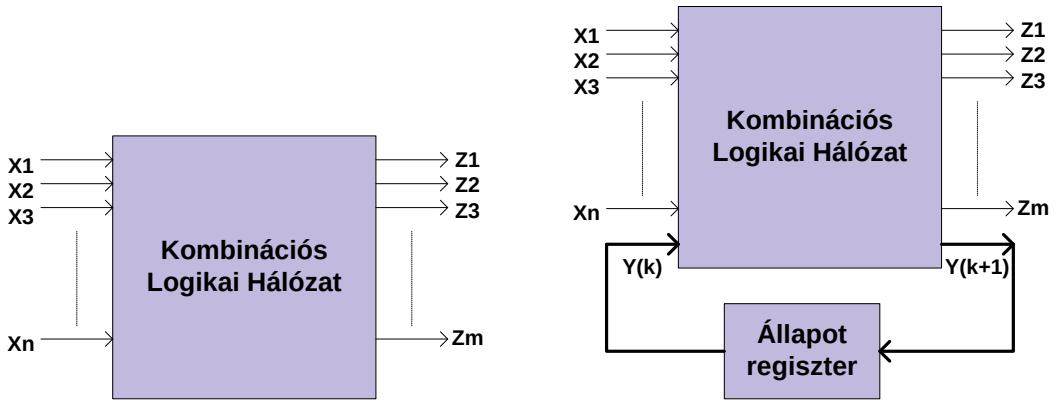
```
-- 4-to-1 MUX viselkedési leírása egyidejű when-else szerkezettel

library ieee;
use ieee.std_logic_1164.all;

entity mux4_1_when is
port (
 sel_in : in std_logic_vector(1 downto 0);
 d_in0, d_in1, d_in2, d_in3: in std_logic;
 y_out : out std_logic );
end entity mux4_1_when;


architecture behav of mux4_1_when is
begin
 -- konkurens értékadások
 y_out <=  d_in0 when sel_in = "00" else
        d_in1 when sel_in = "01" else
        d_in2 when sel_in = "10" else
        d_in3 when sel_in = "11" else
        'X';    --egyébként unknown
end behav;
```

További feladatként írjon a modellhez egy tesztpad-ot, és vizsgálja meg a működését ISim szimulátor segítségével!

2.6. Szekvenciális hálózatok

Rövid áttekintés

A kombinációs logikai hálózatok és a szekvenciális (sorrendi) hálózatok között a legfontosabb különbség, hogy míg egy hagyományos kombinációs hálózatnál a mindenkori bemenetek kombinációja (matematikailag inkább variációja) határozza meg a kimeneteket, addig a szekvenciális hálózatoknál a mindenkori bemeneti kombinációk, valamint a szekunder (másodlagos) állapotkombinációk együttesen határozzák meg az új kimeneti értékeket. Általánosságban azt mondhatjuk, hogy a szekvenciális hálózatok olyan kombinációs logikai hálózatok, amelyek rendelkeznek egy belső visszacsatolással, illetve a visszacsatoló ágban állapot tárolással. Az állapotok szekvenciája ugyanazon bemeneti kombinációk mellett is képes mindig új kimeneti értéket, illetve következő állapot értéket generálni, amelyből a visszacsatolt aktuális állapotok meghatározhatóak. A kombinációs hálózatok, illetve szekvenciális hálózatok blokkdiagram szintű felépítése a következő *2.17. ábrán* adott:



2.17. ábra: kombinációs logikai hálózatok és szekvenciális hálózatok összehasonlítása

Mindkét esetben X jelöli a bemenetek, Z a kimenetek, míg a sorrendi hálózat esetén Y(k+1) a következő állapotot, Y(k) a visszacsatolt aktuális állapotot jelöli. A állapotok következő értékének (next-state) meghatározásától függően a szekvenciális hálózatok között a következő főbb csoportokat lehet képezni:

- **Reguláris szekvenciális áramkörök**: az állapot változások (átmenetek) szabályos kombinációs mintát követnek, mint pl. a számláló vagy léptető regiszter nyomon követhető állapot változásai. A következő állapot logika huzalozott kombinációs hálózatból épül fel.
- **FSM:** Véges állapotú autómata (Finite State Machine). Az állapot átmenetek nem egy szabályos, ismétlődő mintát követnek, hanem ún. véletlen logika (random-logic) elvén valósulnak meg. Az ilyen autómata modellek működése a vezérlés folyam gráfok (CFG) segítségével szemléletes módon ábrázolhatóak (vezérlési út).

- **FSMD**: Véges állapotú autómata modellek + adatúttal (Finite State Machine with Data Path): ez az áramkör általánosan egy szabványos szekvenciális hálózat, illetve egy FSM modell összekapcsolásából áll. Külön adat-, és vezérlési úttal rendelkezik (azaz működése DFG adatfolyam, illetve CFG vezérlési folyam gráfokkal felírva szimbolikus módon is szemléltethető).

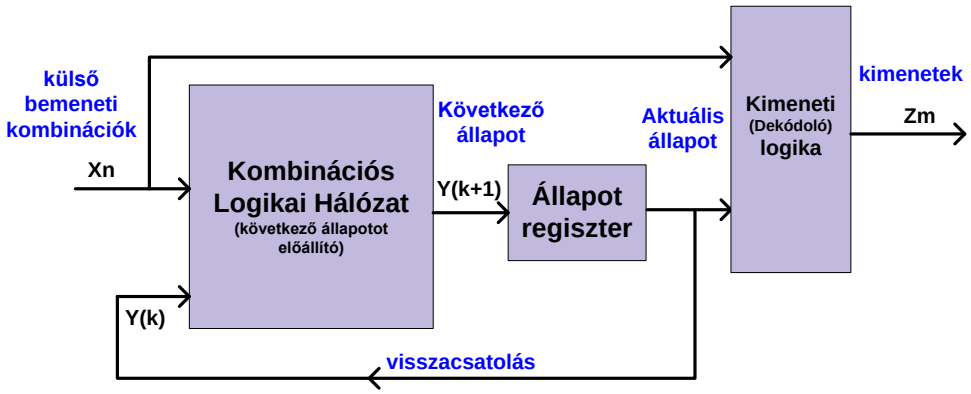
Az állapotregiszterek többféle sorrendi hálózat visszacsatoló ágában lévő alap építőelemként használhatóak. A legtöbb elemi tároló esetén külön kell választani a szinkron (közös órajellel vezérelt), illetve aszinkron (ütemezett aszinkron) működést. A szinkron (esetenként aszinkron) működésű tároló elemek között tipikusan az S-R, J-K, T, D, illetve D-G típusú tárolókat szokás megemlíteni. Ezek közül is, az egyik legfontosabb alapépítő elem a D-tároló, amely a sorrendi hálózatok visszacsatoló ágaiban a legtöbb alkalmazásoi példa esetén megtalálható. D-tároló esetén – de általánosságban igaz a többi tároló esetére is – hogy a definíció szerint, és egyben működésbeli módjuk szerint, a következő lényeges megkülönböztetéseket kell tennünk:

- *Latch* vagy *retesz*: szintvezérelt esetben (logikai ’1’ v. ’0’ szintre vezérelt aszinkron működésű tároló elem),
- *Flip-flop*: élvezérelt esetben (az órajel felfutó, vagy lefutó élére vezérelt tároló elem).

Az áramköri szintézis szempontjából fontos itt emlékeznünk arra is, hogy az Xilinx FPGA eszközök konfigurálható logikai blokkjain (CLB) belül képzett szeleketben (slice) a LUT mellett pontosan D-tárolókat találhatunk, amelyek akár szintvezérelt, akár élvezérelt áramköri elemként is tetszőlegesen konfigurálható (alkalmazástól függően).

Szekvenciális hálózatok felépítése

A következő *2.18. ábrán* egy hagyományos szekvenciális (sorrendi) hálózat felépítése látható.



2.18. ábra: Sorrendi hálózat blokkdiagramja

A sorrendi hálózatok legfőbb építőelemei:

- **Kombinációs logikai hálózat**: hívják még következő állpot-logikának, mivel a soron következő Y(k+1) előállításáért felel, amelyet azután eltárolunk

- **Állapot tároló / regiszter**: ez általában egy szinkron működésű tároló elem (legegyszerűbb esetben a visszacsatoló ágban elhelyezett D-FF, vagy azokból felépített regiszter), amely egyrészt az aktuális állapoto(ka)t tárolja, másrészt visszacsatolja az Kombinációs logikai hálózat bemenetére (Y(k) visszahatása a bemenetre)
- **Kimeneti, vagy dekódoló logika**: olyan logikai építőelem (ált. dekódoló áramkör) amely a kimeneti jeleket (pl. akár vezérlő jeleket) generálhat

További fontos paraméterek egy szinkron szekvenciális hálózatban a következőek lehetnek:

- T(setup): az állapotregiszter elemi tárolóinak az órajel aktív éle előtti időzítési intervalluma
- T(hold): az állapotregiszter elemi tárolóinak az órajel aktív éle utáni időzítési intervalluma
- T(cq): („clock to q") az az időkésleltetés, amíg az órajel hatására az elemi tárolók d bemenetén lévő adat a q kimenetre jut (ún. megszólalási idő). De lehetne akár modellezni a vezetékek jelterjedési idejét is (T(propagation)), valamint
- T(comb): Kombinációs logikai hálózat megszólalási ideje, mialatt a bemeneti Xn kombinációk a logikai hálózat egy-, vagy több- kimenetére (F) nem jutnak.

Ezekből az értékekből egy szinkron sorrendi hálózat maximális működési frekvenciáját lehet meghatározni, előzetes becsléssel. Természetesen a végső terv implementálása során a szintetizáló programfejlesztő környezet (XST) pontosan megadja a „routolás" fázisa után kapott időzítési késleltetéseket. Ezek pontos részleteire jelen jegyzetben nem térünk ki [XILINX]. Ekkor a maximális frekvencia meghatározása a következő *(2.2) képlet* szerint adható meg:

$$f_{\max} = \frac{1}{T} = \frac{1}{T(cq) + T(comb) + T(setup)}. \qquad (2.2)$$

D-tárolók és regiszterek

Az ismertetett szekvenciális hálózatok, vagy akár későbbi a Mealy, illetve Moore FSM automata modellek visszacsatoló ágaiban általánosan a legegyszerűbb elemi tárolót, a „D-tárolót" használják ahhoz, hogy belőlük az állapotok bitszélességéhez igazított méretű „állapotregisztert" tervezzenek. Azonban a VHDL nyelvet újonnan megismerők számára a legnagyobb problémát a tároló elemek, illetve belőlük felépülő komponensek leírása jelenti anélkül, hogy bizonyos esetekben, például a kombinációs hálózatok esetén az ún. „nemkívánt memória hatást", mint rejtett hibát kiküszöböljük (lásd *2.3.2. fejezet*). Ezeknek a tárolóknak, illetve a belőlük felépített regisztereknek a VHDL nyelvi leírásait adjuk meg a következő részekben:

A szinkron, órajellel vezérelt esetben ezt a D-tárolót nevezzük flip-flop-nak, amelynek működtetése történhet felfutó (rising edge), vagy lefutó élre (falling edge) vezérelve. A felfutó élre vezérelt D-FF legtipikusabb működési módjait definiáló igazságtáblázatok, a hozzájuk kapcsolódó VHDL leírások a következők:

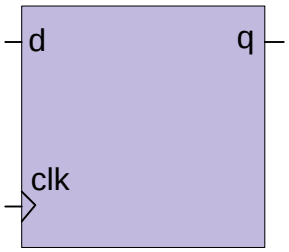
– egyszerű élvezérelt D-tároló, reset és engedélyező jel nélkül
– élvezérelt D-tároló aszinkron reset jellel
– élvezérelt D-tároló aszinkron reset, és engedélyező jellel.

Most ezek felépítését, működését és leírásait nézzük meg részletesen.

a.) Feladat: élvezérelt D-tároló aszinkron reset és engedélyező jel néljül

Ez a hagymányos D-FF elemi tároló elem, engedélyező és reset bemenetek nélkül. A clk felfutó élének ↑ hatására tárolja el a d-bemenetről kapott értéket, amelyet egy clk órajel-ciklussal később a q kimenetére tesz (változatlan értékkel).

2.10. táblázat: Élvezérelt D-tároló (reset és engedélyező jelek nélkül)



clk	q
0	q
1	q
↑	d

Az aszinkron reset és engedélyező jel nélküli D-tároló lehetséges VHDL leírása a következő:

```
-- élvezérelt D-tároló aszinkron reset és engedélyező jel néljül
library ieee;
use ieee.std_logic_1164.all;

entity d_flipflop is
   port(
      clk: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end d_flipflop;

architecture behav of d_ flipflop is
begin
   process(clk)
   begin
      if (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
end behav;
```

Megjegyeznénk, hogy a fenti kódban a következő sor helyett

```
if (clk'event and clk='1') then
```

írhattuk volna a következőt is:

```
if rising_edge(clk) then
```

mivel a VHDL IEEE1164 szabványos csomagja tartalmazza az utasítást. Hasonló módon, ha lefutó élre ↓ vezérelt D-tárolót adunk meg, akkor pedig:

```
if (clk'event and clk='0') then
```

helyett

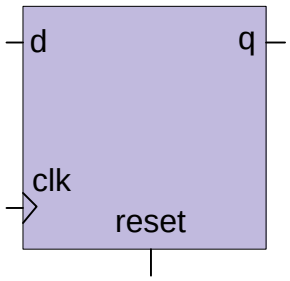
```
if falling_edge(clk) then
```

utasítást adhatjuk meg. Mindekettő foglalt szó a VHDL-ben.

b.) Feladat: élvezérelt D-tároló aszinkron reset jellel, de engedélyező jel néljül

Ez a tároló egy speciális, kibővített működésű D-FF, és aszinkron reset bemenettel rendelkezik, de engedélyező bemenet nélkül. Ha a reset bármikor aktív magas (reset = ’1’), akkor a tároló inicializált q = ’0’ állapotba kerül. A reset jel ’aszinkron’ viselkedésű, mivel független a clk bemenettől. Ha nincs reset (reset = ’0’), akkor pedig az a.) pontban előbb említett hagyományos D-FF működését kapjuk.

2.11. táblázat: Élvezérelt D-tároló, magas aktív aszinkron reset jellel (de engedélyező nélkül)



reset	clk	q
1	-	0
0	0	q
0	1	q
0	↑	d

Az aszinkron resettel rendelkező, de engedélyező jel nélküli D-tároló VHDL leírása a következő:

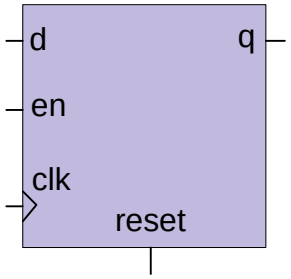
```
--aszinkron resettel rendelkező, de engedélyező jel nélküli D-tároló
library ieee;
use ieee.std_logic_1164.all;

entity d_flipflop_reset is
   port(
      clk, reset: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end d_flipflop_reset;
```

```
architecture behav of d_flipflop_reset is
begin
   process(clk,reset)
   begin
      if (reset='1') then
         q <='0';
      elsif (clk'event and clk='1') then
         q <= d;
      end if;
   end process;

end behav;
```

c.) Feladat: élvezérelt D-tároló aszinkron reset és szinkron engedélyező jellel

Szintén speciális, kibővített működésű D-FF, amely mind aszinkron reset, mind pedig szinkron engedélyező bemenettel is rendelkezik. Ha a reset bármikor aktív magas (reset = ’1’), akkor a tároló inicializált q = ’0’ állapotba kerül. Aszinkron viselkedésű a reset jel, mivel független a clk bemenettől. Ha nincs reset állapotban (reset = ’0’), és engedélyezve van (en = ’1’), valamint felfutó órajelet kap (↑), akkor tárolja el a d-bemenetről érkező értéket (q=d). Minden más esetben a korábbi állapot értéket (q) őrzi meg.

2.12. táblázat: Élvezérelt D-tároló, magas aktív aszinkron reset és szinkron engedélyező jellel



reset	clk	en	q
1	-	-	0
0	0	-	q
0	1	-	q
0	↑	0	q
0	↑	1	d

Az aszinkron resettel és szinkron módú engedélyező jellel is rendelkező D-tároló VHDL leírása pedig a következő:

```
-- élvezérelt D-tároló, magas aktív aszinkron reset és szinkron engedélyező
--jellel
library ieee;
use ieee.std_logic_1164.all;

entity d_flipflop_en is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end d_flipflop_en;
architecture behav of d_flipflop_en is
begin
   -- clk és reset jeltől is együttesen függ - szenzitív lista
   process(clk, reset)
   begin
      if (reset='1') then
         q <='0';
      elsif (clk'event and clk='1') then
         if (en='1') then
            q <= d;
         end if;
      end if;
   end process;
end behav;
```

Mivel az előző áramkör szinkron módú engedélyező jelet is tartalmaz, ezért helyettesíthető, illetve felépíthető lenne egy elemi D-tároló, illetve egy egyszerű következő állapot logikát megvalósító kombinációs hálózat segítségével. Ez a kombinált modul ráadásul egyidejű hozzárendeléssel van megadva a következő VHDL leírásban:

```
-- élvezérelt D-tároló, magas aktív aszinkron reset és szinkron engedélyező
--jellel - konkurens módon

library ieee;
use ieee.std_logic_1164.all;
```

```
entity d_flipflop_en2 is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end d_flipflop_en2;

architecture behav of d_flipflop_en2 is
   signal r_reg, r_next: std_logic;
begin
   -- D FF
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <='0';
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- egyidejű hozzárendelések
   -- next-state logika
   r_next <= d when en ='1' else
             r_reg;
   -- kimeneti logika
   q <= r_reg;
end behav;
```

Regiszterek

Amennyiben a fenti D-FF-ok valamelyikétből n-darabot egymás mellé helyezünk, egy n-bites regisztert, illetve n-bites léptető (shift) regiszter kapunk függően attól, hogy párhuzamos vagy soros módszerrel kapcsoljuk őket össze. Ha azt tételezzük fel, hogy a korábban ismertetett Mealy, illetve Moore modellek n-bites visszacsatolt állapottal rendelkeznek, akkor például egy n-bites párhuzamos ún. *állapot regiszter* képes csak eltárolni a bemenetre visszacsatolandó n darab Y(k) aktuális állapot értékeket.

Feldadat 1: Hagyományos N-bites párhuzamos betöltésű, és kiolvasású regiszter

Az N-bites regiszter egyes elemi D-tárolói tehát ugyanazt a `clk` órajelet, illetve `reset` jelet kell, hogy megkapják a működéshez. A regiszter bitszélességét definiáljuk egy `WIDTH` nevű `integer` típusú `generic` kifejezéssel, amelynek értéke legyen 4.

```
-- Hagyományos N-bites párhuzamos betöltésű, és kiolvasású regiszter

library ieee;
use ieee.std_logic_1164.all;
entity reg_reset is
   generic(WIDTH: integer := 4);
   port(
      clk, reset: in std_logic;
      d: in std_logic_vector(WIDTH-1 downto 0);
      q: out std_logic_vector(WIDTH-1 downto 0)
   );
end reg_reset;

architecture behav of reg_reset is
begin
   process(clk,reset)
   begin
      if (reset='1') then
         q <=(others=>'0');
      elsif (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
end behav;
```

Megjegyzés: Ha a fenti VHDL kódot leszintetiáljuk a Xilinx XST segítsévével (Design → Synthesize XST), akkor a generálás közben az üzenet ablakban a következő sorokat látjuk:

```
Found 4-bit register for signal <q>.
    Summary:
 inferred   4 D-type flip-flop(s)
. . .
=========================================================================
Macro Statistics
# Registers                                          : 1
 4-bit register                                      : 1
=========================================================================
Macro Statistics
# Registers                                          : 4
 Flip-Flops                                          : 4
```

Azaz a szintézis riport is a D-tárolókból felépített 4-bites regiszter generálását bizonyítja.

Feladat 2: Hagyományos N-bites léptető (shift) regiszter, soros betöltéssel és kiolvasással

A hagyományos léptető regiszter, az egyik legegyszerűbb soros betöltésű és kiolvasású regiszter, amelynél nem kontrollálható kivülről a működés, csupán 1-bitpozícióval képes balra, vagy jobbra léptetni a tartalmát. Az N-bites léptető regiszter, amely 1-bitpozíciónyival jobbra lépteti ciklusonként a tartalmát VHDL leírása a következő. Ebben az esetben is használjuk a már ismert generic-et, a bitszélesség megadásához (N := 4):

```
-- Hagyományos N-bites léptető (shift) regiszter
library ieee;
use ieee.std_logic_1164.all;

entity shift_regN is
   generic(N: integer := 4);
   port(
      clk, reset: in std_logic;
      s_in: in std_logic;
      s_out: out std_logic
   );
end shift_regN;
```

```
architecture behav of shift_regN is
   signal r_reg: std_logic_vector(N-1 downto 0);
   signal r_next: std_logic_vector(N-1 downto 0);
begin
   -- szinkron regiszter, aszinkron reset-el, engedélyező nélkül
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   --konkurrens hozzárendelése az 1-bit beléptetéséhez, és kimenet
   --elvételéhez
   -- next-state logika (1 bitpozícióval jobbra léptetés)
   r_next <= s_in & r_reg(N-1 downto 1);
   -- kimenet
   s_out <= r_reg(0);
end behav;
```

Látható a fenti példánál, hogy a konkurens `r_next` hozzárendelésben az `r_reg` legalsó bitjét elhagyjuk, míg a legfelső bitjéhez hozzáfűzzük jobbról az 1-bites `s_in` értékét. Ezáltal a **jobbra** léptetés 1-bitpozícióval megvalósul. Hasonló módon, ha 1-bitpozíciónyival akarunk **balra** léptetni, akkor a következő módosítást kell megtenni a fenti kódban:

```
   -- next-state logika (1 bitpozícióval balra léptetés)
   r_next <= r_reg(N-2 downto 0) & s_in;
```

Fontos megjegyezni, hogy szintézis szempontjából a shift regiszter a Xilinx FPGA-k CLB blokkjaiban lévő 4-bemenetű LUT táblázatában realizálódik. Egyetlen 4-bemenetű LUT felhasználásával maximálisan 16x1 bites shift regiszter implementálható.

Feladat 3: Univerzális N-bites léptető (Barrel-shift) regiszter, párhuzamos/soros betöltéssel és kiolvasással

Az univerzális vagy ún. Barrel-shift regiszter a hagyományos léptető regisztertől annyiban tér el, hogy kivülről is kontrollálható a működése (ctrl jel megadásával), és léptethető a tartalom 1-bitpozícióval balra, jobbra, vagy akár meg is tarthatjuk az állapotát. Az univerzális léptető regiszter akár párhuzamos-soros, akár soros-párhuzamos betöltéssel-kiolvasással is rendelkezhet. A Barrel-shift kifejezést általában arra szokás használni, amikor nemcsak 1-bittel balra, illetve jobbra tudunk léptetni, hanem tetszőleges M bittel, ahol általában N>M. A `ctrl` egy kétbites std_logic_vector típusú vezérlőjel, amellyel szabályozható az entitás portlistáján keresztül a belső működés:

```
-- univerzális N_bites shift regiszter
library ieee;
use ieee.std_logic_1164.all;

entity univ_shift_reg is
   generic(N: integer := 4);
   port(
      clk, reset: in std_logic;
      ctrl: in std_logic_vector(1 downto 0); -- szabályozza a léptetést
      d: in std_logic_vector(N-1 downto 0);
      q: out std_logic_vector(N-1 downto 0)
   );
end univ_shift_reg;
architecture behav of univ_shift_reg is
   signal r_reg: std_logic_vector(N-1 downto 0);
   signal r_next: std_logic_vector(N-1 downto 0);
begin
   -- szinkron regiszter, aszinkron reset-el, engedélyező nélkül
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif rising_edge(clk) then
         r_reg <= r_next;
      end if;
   end process;
```

```
   --konkurrens with-select hozzárendelése az kontrollálható léptetéshez,
   -- next-state logika
   with ctrl select
    r_next <=
      r_reg                          when "00", --helyben marad
      r_reg(N-2 downto 0) & d(0)     when "01", --shift balra 1-bittel;
      d(N-1) & r_reg(N-1 downto 1)   when "10", --shift jobbra 1-bittel;
      d                              when others; -- párhuzamos betöltés
   -- kimenet
   q <= r_reg;
end behav;
```

Szintézis során látható hogy a következő állapot logika (r_next) egy 4-1 mux-ot használ a megfelelő bemenetek kiválasztásához.

További feladatként tervezzen a fenti példa alapján egy N-bites Barrel Shift regiszter, amelyben M pozíciót lehet léptetni, attól függően hogy a kapcsolókon keresztül milyen irányt (jobbra, balra, helyben marad, vagy éppen párhuzamosan betölt) engedélyezünk. Szimulálja le a viselkedését ISim szimulátor segítségével.

További feladatok

1.) Az előző példák mindegyikéhez

- Hagyományos N-bites párhuzamos betöltésű, és kiolvasású regiszter
- Hagyományos N-bites léptető (shift) regiszter, soros betöltéssel és kiolvasással
- Univerzális N-bites léptető (Barrel-shift) regiszter, párhuzamos/soros betöltéssel és kiolvasással
- Modulo-m N-bites bináris számláló

tervezzen külön-külön tesztpadot, és szimulálja le a viselkedésüket a Xilinx ISim szimulátor segítségével!

2.) Szintén az előző példák mindegyikét szintetizálja le, majd fizikailag implementálja FPGA áramkörre. Generáljon letölthető bitfájlokat, és ellenőrizze müködésüket az eszközön.

Segítség: a kényszerfeltételek (.ucf) megadásához használja a Digilent gyártó által adott .ucf állományt a benne lévő paraméterekkel. Ezt felhasználva szerkesszen az egyes lábakhoz kényszerfeltételeket. Használja a nyomó gombokat, kaplcsolókat (pl. a számláló szinkron törléséhez, illetve a léptetés irányához, adatok párhuzamos betöltéséhez stb.)

Regiszter tömbök

A regiszter tömbök (register file, register bank) olyan összetett tároló elemek, amelyek elemi n-bit szélességű regiszterek lineáris tömbjéből (gyűjteményéből) állnak. Definiájunk egy olyan regiszter tömböt, amelynek egy bemeneti adat portja, egy kimeneti adat portja, valamint a szokványos módon egyetlen írási címe, és egy olvasási címe van, ezeket jelöljük rendre: `w_data`, `r_data`, `w_addr`, és `r_addr` jelekkel. Legyen `WIDTH` generic az egyes regiszterek

bitszélessége, míg ADDR generic pedig a regisztertömb címvonalának szélessége. Ezáltal összesen 2^ADDR×WIDTH bit kapacitású regiszter bankot tudunk elérni. Az egyes regiszterek használjanak aszinkron reset, illetve szinkron engedélyező jeleket. A regiszter bank felépítéséhez először egy WIDTH bitszélességű std_logic_vector-okból álló és 2^ADDR nagyságú tömbtípust definiáljunk, reg_file_type néven. Majd ezt a saját 2D-tömbtípust használjuk fel egy array_reg nevű tömb, mint belső jel deklarációjához, amely a regiszterbe történő beíráshoz, illetve kiolvasáshoz szükséges. A regiszter tömb leírását szolgáló lehetséges VHDL leírás a következő:

```
-- regiszter tömb

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity reg_file is
   generic(
      WIDTH: integer:=8;   --bitek száma az egyes regiszterekben
      ADDR : integer:=2      --cím bitek száma
   );
   port(
      clk, reset: in std_logic;
      wr_en: in std_logic;
      w_addr, r_addr: in std_logic_vector (ADDR-1 downto 0);
      w_data: in std_logic_vector (WIDTH-1 downto 0);
      r_data: out std_logic_vector (WIDTH-1 downto 0)
   );
end reg_file;

architecture behav of reg_file is
   type reg_file_type is array (2**ADDR-1 downto 0) of
        std_logic_vector(WIDTH-1 downto 0);
   signal array_reg: reg_file_type;
begin
   process(clk,reset)
   begin
      if (reset='1') then
         array_reg <= (others=>(others=>'0'));
      elsif (clk'event and clk='1') then
         if wr_en='1' then
```

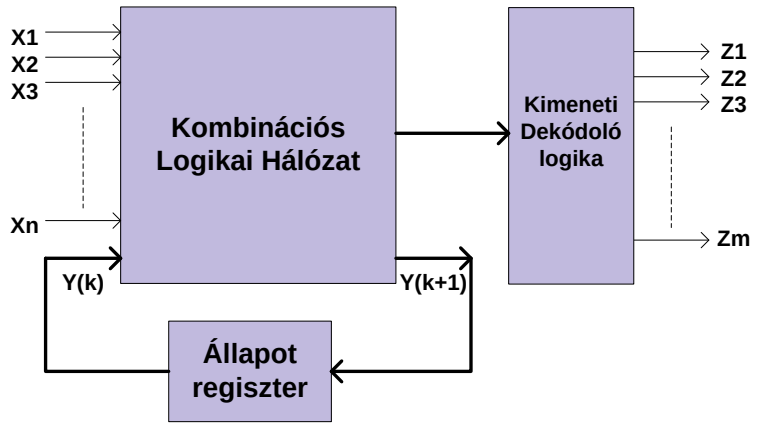
```
            -- ÍRÁSI port
    array_reg(to_integer(unsigned(w_addr))) <= w_data;
         end if;
      end if;
   end process;
    -- OLVASÁSI port
   r_data <= array_reg(to_integer(unsigned(r_addr)));
end behav;
```

Megjegyeznénk, mivel a 2D-tömb indexelése integer típust feltételez, ezért a címek (`w_addr, r_addr`) explicit típus konverziójára van szükség két lépésben: először előjel nélkülivé, majd pedig integerré konvertáljuk a címeket a tömb hozzárendeléshez.

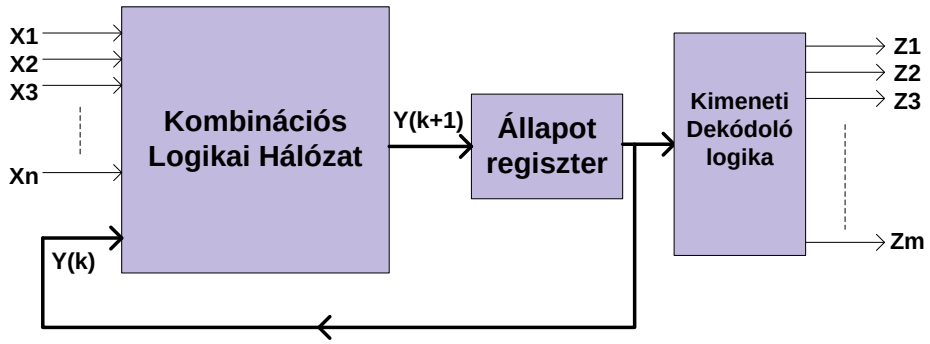
FSM: Véges Állapotú Automata modellek

Szekvenciális hálózatok klasszikus Mealy, Moore modelljei

Az FSM kifejezés (Finite State Machine) – véges állapotú automata modellt jelent, amelynek két fontos reprezentánsa a Mealy, illetve Moore modellek, amelyeket a szinkron sorrendi hálózatok tervezése során előszeretettel alkalmaznak.



a.) Mealy modell



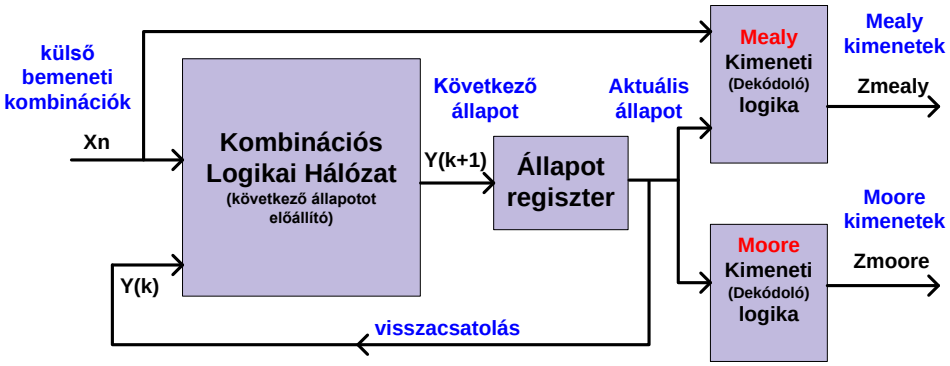
b.) Moore modell

2.19. ábra: Szinkron szekvenciális hálózatok Mealy és Moore féle alap modelljei

A következő összehasonlításban a szekvenciális hálózatok ezen klasszikus alapmodelljeit ismerhetjük meg röviden. A két automata modell közötti lényeges különbség a kimenetek (Z) előállításában van. Míg Mealy esetben a kimenetek közvetlenül függenek az aktuális bemeneti kombinációk, illetve a visszacsatolt aktuális állapotok együttes értékeitől, addig a Moore modell esetén a bementeket csak közvetett módon (és nem közvetlenül) befolyásolják a kimenetek előállítását. Azaz Moore modell esetében a kimeneteket az aktuálisan eltárolt állapot értéktől (present state) kapunk. Ezáltal a kimenetek és az állapotok között fellépő szinkronizáltlanságból adódó probléma megszüntethető. A következő *2.19. ábra* a klasszikus Mealy, illetve Moore-féle szekvenciális áramköri modellek blokkdiagram szintű felépítését mutatják:

Kombinált FSM modell: Mealy, Moore

A fenti *2.19. a.) és b.) ábrákon* a Kombinációs Logikai Hálózat felel a soron következő Y(k+1) állapotok előállításáért, ezért hívják következő állapot (Next State) logikai blokknak is. Számos szakirodalomban [*ASHENDEN*], [*CHU*] élnek azzal a lehetőséggel is, hogy a két FSM sorrendi hálózati modellt egyetlen komplex áramköri modellel adják meg, lásd következő *2.20. ábra*.

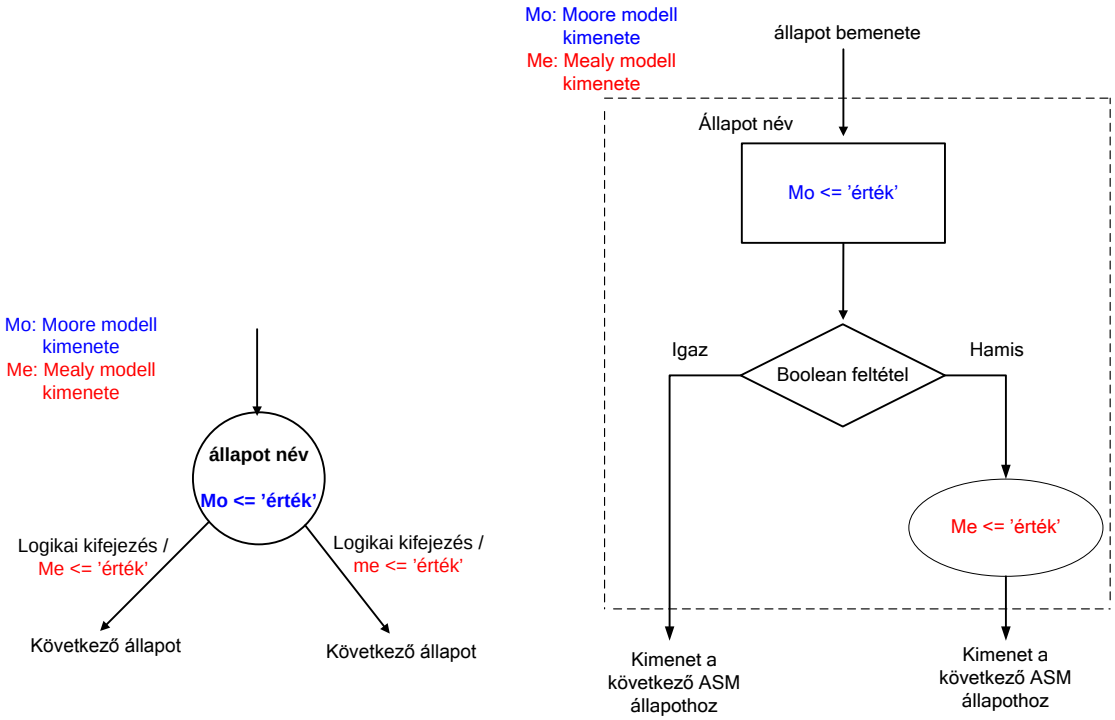


2.20. ábra: Kombinált FSM modell: Mealy, Moore esetre

FSM működésének szemléletes ábrázolási formái Mealy, Moore modellekre

Az FSM automata modelleket szemléletes módon általában különböző grafikus állapot diagramok segítségével, vagy az ún. ASM (Algorithmic State Machine Chart – algoritmikus állapotgép) diagramok megadásával lehet reprezentálni. Mindegyik esetben a bemeneteket, kimeneteket, állapotokat és állapot átmeneteket (transition) is tudjuk ábrázolni. Míg a hagyományos állapot diagram sokkal tömörebb megadási módot tesz lehehetővé, főleg egyszerűbb sorrendi hálózatok működésének leírására; addíg az ASM diagram felépítése összetettebb, és részletesebb leíró jelleget ölt, amely egyben azt is jelenti, hogy sokkal bonyolultabb feladatok leírását is biztosítja (pl. állapot átmenetek feltételei, akciók sorrendje stb. is szemléltethető). De a két diagram természetesen ekvivalens módon azonos viselkedést kell, hogy meghatározzon.

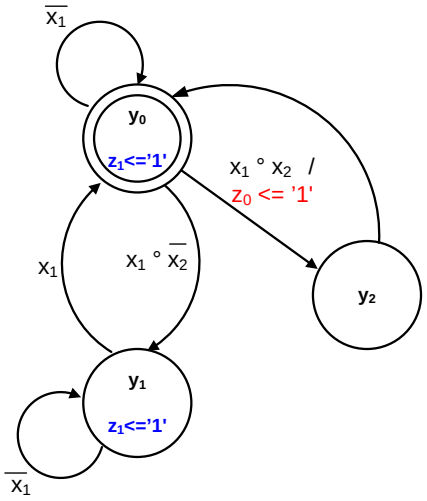
A hagyományos állapotdiagram, illetve ASM diagramm jelölései és paraméterei a következő ábrán adottak (*2.21. ábra*):



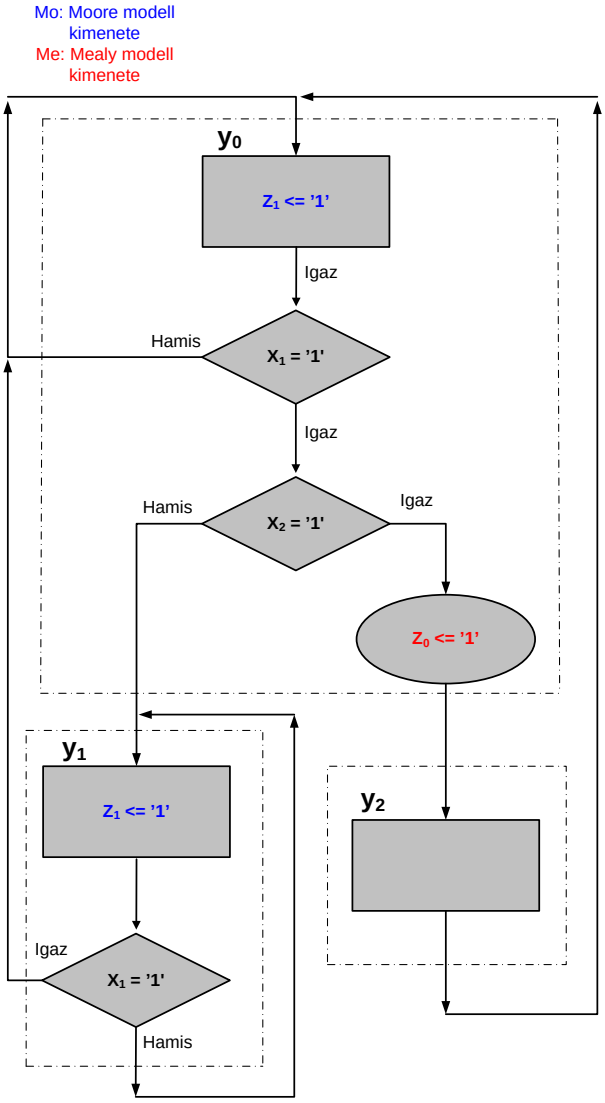
a.) állapotdiagram b.) ASM-diagram

2.21. ábra: Állapotdiagram és ASM-diagram egyetlen állapotának paraméterei, fontosabb jelölései

Tekintsük a következő *2.22. ábra* szerinti példát (hagyományos és ASM-diagram esetére egyaránt):



a.) állapotdiagram



b.) ASM (algoritmikus állapotdiagram)

2.22. ábra: Hagyományos állapotdiagram, illetve ASM diagram lehetséges megadási módja

Az FSM modell egyes állapotait a felsorolt, ún. enumerált VHDL típussal adjuk meg, amely saját `state_type` típust definiálva belső jeleket adhatunk meg (`state_reg`, illetve `state_next`):

```
   type state_type is (y0, y1, y2);   --típus deklaráció
   signal state_reg, state_next: eg_state_type;  --belső jelek
```

A fenti állapotdiagramon, illetve ASM diagramon ábrázolt FSM modell Mealy, illetve Moore modell szerinti származtatását a következő VHDL leírások mutatják. A két eset közötti legfontosabb különbség természetesen a következő állapot (next-state) logika előállításában van: mint ismeretes Mealy modell esetén a bemenetek és az állapotok is együttesen befolyásolják a kimenet előállítását, míg Moore modellnél mindig csak az aktuális állapotokból kapjuk a kimenetetek értékeit.

```
-- Mealy modell VHDL leírása állapot diagram alapján
library ieee;
use ieee.std_logic_1164.all;
entity fsm_mealy is
   port(
      clk, reset: in std_logic;
      x1, x2: in std_logic;
      z0, z1: out std_logic
   );
end fsm_mealy;

architecture arch of fsm_mealy is
   --egyedi felsorolt/enumerált típus deklarációja, és definíciója az
   -- állapotok megadásához: y0 = 0, y1 = 1, y2 = 2
   type state_type is (y0, y1, y2);
   signal state_reg, state_next: state_type;
begin
   -- állapot regiszter
   process(clk, reset)
   begin
      if (reset='1') then
         state_reg <= y0;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;

   -- next-state logika case-when szerkezettel. Leírja, hogy melyik
   -- állapotból, mely lehetséges állapotba juthatunk
   process(state_reg, x1, x2)
```

```
   begin
      case state_reg is
         --y0 állapot
         when y0 =>
            if x1='1' then
               if x2='1' then
                  state_next <= y2;
               else
                  state_next <= y1;
               end if;
            else
               state_next <= y0;
            end if;
         --y1 állapot
         when y1 =>
            if (x1='1') then
               state_next <= y0;
            else
               state_next <= y1;
            end if;
         --y2 állapot
         when y2 =>
            state_next <= y0;
      end case;
   end process;
   -- Mealy kimeneti logika. z
   --itt az allapottol és a bemenetektől direkt módon függ a kimenet!
   process(state_reg, x1, x2)
   begin
      case state_reg is
         when y0 =>
            if (x1='1') and (x2='1') then
              z0 <= '1';
            else
              z0 <= '0';
            end if;
         when y1 | y2 =>
            z0 <= '0';
      end case;
   end process;
end arch;
```

A következő VHDL forrás pedig az állapotdiagram, vagy ASM diagram Moore modell szerint leírását szemlélteti:

```
-- Moore modell VHDL leírása állapot diagram alapján
library ieee;
use ieee.std_logic_1164.all;
entity fsm_moore is
   port(
      clk, reset: in std_logic;
      x1, x2: in std_logic;
      z0, z1: out std_logic
   );
end fsm_moore;

architecture arch of fsm_moore is
   --egyedi felsorolt/enumerált típus deklarációja, és definíciója az
   -- állapotok megadásához: y0 = 0, y1 = 1, y2 = 2
   begin
   type state_type is (y0, y1, y2);
   signal state_reg, state_next: state_type;
begin
   -- állapot regiszter
   process(clk, reset)
   begin
      if (reset='1') then
         state_reg <= y0;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
```

```
   -- next-state logika case-when szerkezettel. Leírja, hogy melyik
   -- állapotból, mely lehetséges állapotba juthatunk
   process(state_reg, x1, x2)
   begin
      case state_reg is
         --y0 állapot
         when y0 =>
            if x1='1' then
               if x2='1' then
                  state_next <= y2;
               else
                  state_next <= y1;
               end if;
            else
               state_next <= y0;
            end if;
         --y1 állapot
         when y1 =>
            if (x1='1') then
               state_next <= y0;
            else
               state_next <= y1;
            end if;
         --y2 állapot
         when y2 =>
            state_next <= y0;
      end case;
   end process;
   -- Moore kimeneti logika
   --Moore eset: csak az aktuális allapottól függ direkt módon a kimenet!
   process(state_reg)
   begin
      case state_reg is
         when y0|y1 =>
            z1 <= '0';
         when y2 =>
            z1 <= '1';
      end case;
   end process;
end arch;
```

2.7. További példaprogramok VHDL-ben

Feladat 1: Hagyományos N-bites bináris számláló és szimulációja

A Spartan3E FPGA a Digilent Nexys-2 fejlesztőkártya 50 MHz-es kristály osszcillátorról kapja az órajelet a B8-as lábon keresztül (lásd Digilent Nexys2 Reference Manual [*DIGILENT*], [*NEXYS2*]). Az `.ucf` fájlban a kényszerfeltételként (constraints) megadott peridódus időt ezért 20ns –ra kell beállítani. A jegyzetben a legtöbb példánál pontosan ennek a külső 50 MHz-es órajelnek a használatát feltételezzük a szintetizálás során. Az FPGA alapú beágyazott rendszerek tervezési feladatai során azonban felmerülhet az igény más órajelek, valamint alacsonyabb órajel frekvenciák beállítására különböző periférák használata esetén. Ez több módon is történhet:

- Xilinx CoreGen generátor segítségével egy DCM-et használva a bemenő órajelből (esetünkben 50MHz) tetszőleges kimenő órajel(eke)t állíthatunk elő, mivel az órajelek szorzására, osztására, fázistolására is lehetőség van.
- Előre definiált VHDL nyelvi templatek segítségével: Xilinx ISE → Edit menü →Language Templates → Device Primitive Instantiation → Spartan-3E → Clock Components → Digital Cloxk (DCM_SP). Ebben az esetben a Spartan3E eszköz primitívet hívunk meg, amelynek forráskódja beilleszthető egy VHDL forrásba (Valójában Xilinx CoreGenerator is egy ilyen nyelvi templatet / primitívet generál).
- Alacsonyabb órajelet egyszerűbben is elő tudunk állítani, a bejövő (esetünkben 50 MHz) órajel folyamatos „leosztásával”, amelyet egy N-bites számláló létrehozásával tudunk legegyszerűbben megvalósítani. Most erre nézünk egy példát.

Ebben a feladatban az N-bites számlálónknak legyen a neve `counterN_bit`, és legyen két bemenete: egy `clk` órajel, illetve egy `clr` törlő `std_logic` bemenet, valamint egy `q` kimenete, amely N-bites `sdt_logic_vector(N-1 downto 0)` típusú, illetve egy `max_tick` kimenete, amely a számláló átfordulása előtt a maximális érték elérését jelzi (std_logic típusú). Használjunk **generic** típust az N értékének megadásához, és legyen `N=4`. Az `N=4` bites számláló `'0000' – '1111'` ig számol, majd átfordul (wrap-around), és újból `'0000'`-tól kezdve folytatja a számláló léptetését. Ekkor az N bites számláló működése a következő VHDL leírással adható meg:


```
-- hagyományos N-bites bináris számláló
library IEEE;
use IEEE.STD_LOGIC_1164.all;
USE ieee.numeric_std.ALL;

entity counterN_bit is
  generic(N : integer := 4);
  port(
   clr : in STD_LOGIC;
   clk : in STD_LOGIC;
   max_tick : out STD_LOGIC;
   q : out STD_LOGIC_VECTOR(N-1 downto 0)
      );
end counterN_bit;

architecture behav of counterN_bit is
    signal count_reg: unsigned(N-1 downto 0);
    signal count_next: unsigned(N-1 downto 0);
begin
  process(clk, clr)
  begin
    if clr = '1' then
    count_reg <= (others => '0');
    elsif clk'event and clk = '1' then
    count_reg <= count_next;
    end if;
  end process;
  --kovetkezo allapot
  count_next <= count_reg + 1;
  --kimeneti logika
  q <= STD_LOGIC_VECTOR(count_reg);
  max_tick <= '1' when count_reg = (2**N-1) else  '0' ;
end behav;
```

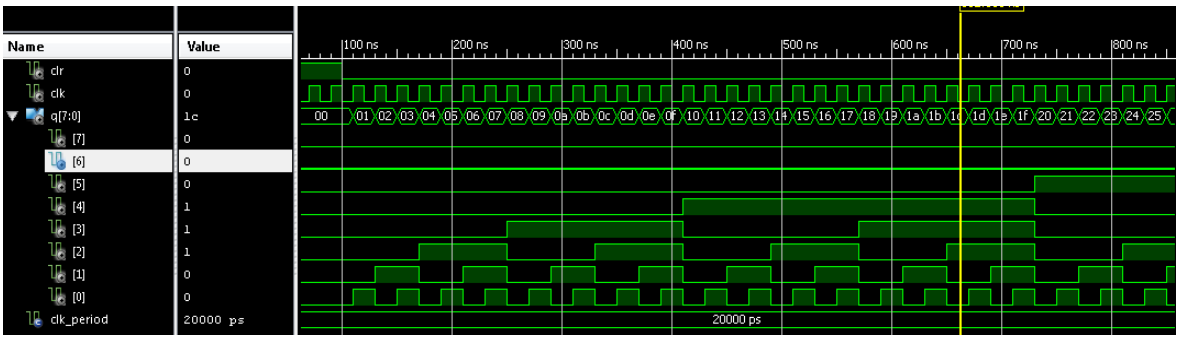
Megjegyzés: nyomatékosítás végett a `_next` kulcsszóval a következő állapot értékét, míg `_reg` kulcsszóval az aktuális állapot értékét jelöltjük, belső jelekként (signal) megadva.

Szimuláció: tervezzünk egy tesztpad-ot (`counterN_bit_tb` néven), amelybe az előző `counterN_bit` entitást, mint legmagasabb szintű modult példányosítjuk, és adjunk a bemeneti jeleire gerjesztést (`clk, rst`), ahhoz, hogy a működését vizsgálhassuk. Teszteljük a VHDL leírás viselkedését a Xilinx ISim szimulátorában. A konstans órajel periódusnak adjuk meg 20 ns időegységet (~ 50 MHz a szimulációs freki is), míg a `stim_process()`-be helyezzük el a gerjesztéseket.

```
-- Tesztágy a CounterN számlálóhoz
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY counterN_bit_tb IS
END counterN_bit_tb;

ARCHITECTURE behavior OF counterN_bit_tb IS

    -- Component deklaráció  Unit Under Test (UUT) példányosításhoz

    COMPONENT counterN_bit
    generic(N : integer :=8);
    port(
         clr : IN  std_logic;
         clk : IN  std_logic;
         max_tick : OUT std_logic;
         q : OUT  std_logic_vector(N-1 downto 0)
        );
    END COMPONENT;


   --bemenetek
   signal clr : std_logic := '0';
   signal clk : std_logic := '0';

 --kimenetek
   signal max_tick : std_logic;
   signal q : std_logic_vector(7 downto 0);

   -- Clock_periodus def: fizikai tipus deklaráció
   constant clk_period : time := 20 ns;

BEGIN

 -- példányosítás the Unit Under Test (UUT)
   uut: counterN_bit
   --felüldefiniálja az alacsonybb hierarchiában lévő N értékét
   generic map(N => 8)
   PORT MAP (
          clr => clr,
          clk => clk,
```

```
          max_tick => max_tick,
          q => q
        );

   -- Clock process definiciók
   clk_process :process
   begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
   end process;

   -- Stimulus process - gerjesztések
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      clr <= '1';
 wait for 100 ns;
 -- insert stimulus here
 clr <= '0';
      wait for clk_period*10;


      wait;
   end process;

END;
```

A szimulációs teszt során kapott hullámforma látható a következő *2.23. ábrán*:



2.23. ábra: CounterN számláló viselkedési szimulációja során kapott hullámalakok

Vizsgálat: Az ISim szimulátorban a `q(7:0)` jelre a jobb egérgombbal kattintva egy legördülő menü jelenik meg, amelyből a `'Radix' -> 'Hexadecimal'` kifejezést kell kiválasztani. Ekkor a `N=8`-bites számláló `q[7:0]` adat kimenetei az alapértelmezett bináris formátum helyett hexadecimális formátumban kerülnek ábrázolásra. A tesztet nem elég az alapértelmezett `1000` ns-ig futtatni, hanem a szimulátor futási idejének kb. `5200` ns-os időegységet kell beállítani (mivel 100 ns reset + 256 * 20 ns míg eljut a számláló végéig: '00'…'FF'). Ekkor a `max_tick` értéke `'1'` lesz, jelezve, hogy elértük a maximális értékét a számlálónak. Az is látható, amikor elérjük az `'FF'` végértéket (8-bit esetén), akkor átfordul a számláló, és ismét `'00'`-tól kezdi egyesével növelni (inkrementálni) az értékeket, ahogy azt a helyes működés szerint is elvártuk.

Implementáció: implementáljuk a fenti tervet, majd töltsük le az Nexys-2 FPGA-ra, felhasználva a korábban tárgyalt lépéseket (lásd *2.4. fejezet*). Ehhez használjuk bemenetként `clear`-nek a `btn(0)` nyomógombot, illetve `clk`-nak a külső 50 MHz-es órajelet (B8 láb), valamint a `q(7:0)` kimeneteket kössük rá a megfelelő `Led(7:0)` lábakra. Ezeket a kényszerfeltételek megfelelő beállításával tudjuk megtenni (lásd korábban .ucf fájl beállításai). Az implementált tervet töltsük fel az FPGA-ra, és vizsgáljuk meg a működését.

Feladat 2: Hagyományos N-bites bináris számláló működtetése osztott órajel használatával, szimulációval, implementációval

Megjegyzés: mivel a külső órajel 50MHz volt, ezért 20 ns-onként változnak a LED(7:0)-ek állapotai. Az előző példánkban tehát folyamatosan nagy frekvenciával vibráló LED kijelzőket, nem pedig egymás utáni lépésekben a számláló növekedésének megfelelő, de lassan felvillanó LED fényeket láthattunk. Felmerül a kérdés, hogyan lehet mérsékelni a kimeneti jelek változásait? Válasz ennek megoldására a következő:

1. egyrészt egy órajel osztót (`clk_div`) kell VHDL-ben realizálni, majd pedig
2. a megfelelően leosztott órajelet (azaz a `clk_div` entitás kimenetét) az N-bites bináris számláló órajel bemenetéhez kell kötni.

Ezután már a LED kijelzőkön keresztül az emberi szemmel is érzékelhető, lassan változó számláló működését láthatjuk.

1.) Órajel leosztására (clock division) kétféle lehetőség is adódik:

- a.) Egyik lehetőség a Xilinx CoreGenerator nevű programjában kiválasztunk és legenerálunk egy DCM_SP blokkot Spartan-3E FPGA-ra (IP Catalog → FPGA Features and Design → Clocking → SPartan3E → Single DCM_SP). Ebben az esetben azonban csak 5 MHz – 311 MHz közötti értéket választhatunk a kimeneti, osztott órajelnek (ami a feladatunk szempontjából nem megfelelő, amennyiben másodpercenkénti jelváltást szeretnénk elérni a LED kimeneteken!!). Ráadásul ennek a DCM_SP blokknak az erőforrás foglalása is magasabb (1 DCM + némi logikai erőforrást is allokál). Ez a megoldás olyan alkalmazások esetén lehet kézenfekvő, amikor többféle, és magasabb kimeneti órajelet szeretnénk előállítani, vagy bufferelt órajel(eket)t, illetve ha egy bejövő külső órajelhez akarjuk szinkronizálni (PLL – Phase Locked Loop) az általunk létrehozott áramköri terveket.
- b.) Előre definiált VHDL nyelvi templatek / primitívek kiválasztása és beágyazása a VHDL forrásainkba (Xilinx ISE → Edit menü → Language Templates), ezzel ekvivalens, ha az ikonsoron a  Villanykörte ikont választjuk ki ☺
- c.) Egy további lehetséges, és a mostani feladat szempontjából egyszerűbb és optimálisabb megoldás is, ha egy nagy bitszélességű (N) bináris számlálót használunk a bejövő órajel leosztására.

Jelen feladatban a c.) megvalósítási mód szerint, tehát egy bináris számlálót használunk az órajel leosztására. Tekintsük a `q(i)` kimeneteit egy számlálónak, ahol `az i=0` esetén a `q(0)` egy felezett órajelet állít elő az első lépésben (mivel lehetséges értékek 1 biten a ’0’ illetve ’1’). Ezáltal, ha egy számlálót használunk az órajel folyamatos leosztására, akkor az i. lépésben a kimeneti osztott órajel

$$q(i): f_i = \frac{f}{2^{i+1}} \tag{2.3}$$

frekvenciájú lesz. Ha például közelítően ~ 1 másodpercenként (0.745 Hz) szeretnénk az órajel-osztóként használt számláló kimenetét beállítani, amivel az N-bites bináris számlálót

összekapcsoljuk, akkor `q(25)` lépésig, azaz 26-bites számlálót kell tervezni. A számlálók értékeinek, az órajelek, illetve periódus idők beállítását foglalja össze a következő *2.13. táblázat*:

2.13. táblázat: Az számláló konfigurálható kimenete az órajel osztáshoz

q(i)	Frekvencia [Hz]	Periódus [ms]
i	**50 000000**	**0.00002**
0	25000000	0.00004
1	12500000	0.00008
2	6250000	0.00016
3	3125000	0.00032
4	1562500	0.00064
5	781250	0.00128
6	390625	0.00256
7	195312.5	0.00512
8	97656.25	0.01024
9	48828.13	0.02048
10	24414.06	0.04096
11	12207.03	0.08192
12	6103.52	0.16384
13	3051.76	0.32768
14	1525.88	0.65536
15	762.94	1.31072
16	381.47	2.62144
17	190.73	5.24288
18	95.37	10.48576
19	47.68	20.97152
20	23.84	41.94304
21	11.92	83.88608
22	5.96	167.77216
23	2.98	335.54432
24	1.49	671.08864
25	0.745	1342.17728

Első lépésként tehát egy 26-bites számlálót (`q(25)`) kimenettel kell tervezni a VHDL nyelven ahhoz, hogy a bejövő 50 MHz-es órajelből a kimeneten ~1 Hz-es órajelet kapjunk. Az entitásnak a neve legyen `clk_div`, amelynek VHDL leírása a következő:

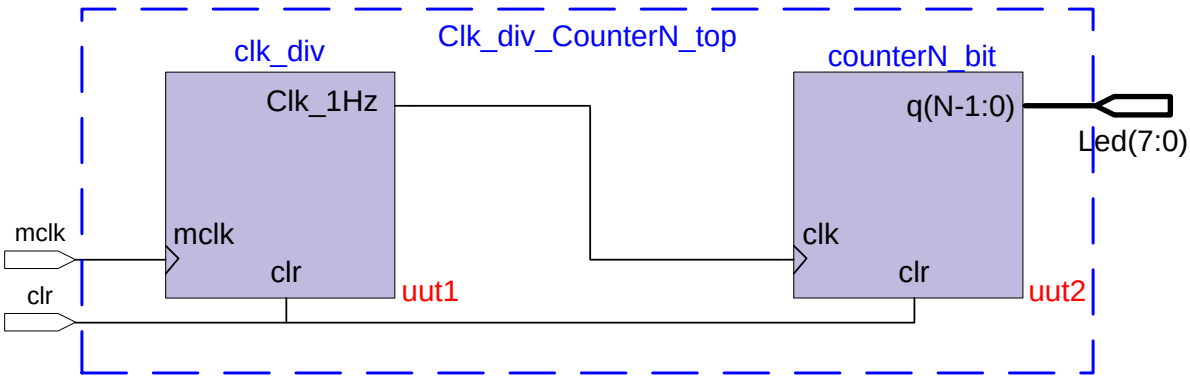
```
-- N bites órajel osztó áramkör
library IEEE;
use IEEE.STD_LOGIC_1164.all;
USE ieee.numeric_std.ALL;

entity clk_div is
  generic(N : integer := 26);
  port(
   clr : in STD_LOGIC;
   mclk : in STD_LOGIC;
   clk_1_hz : out STD_LOGIC
      );
end clk_div;

architecture behav of clk_div is
    signal count_reg: unsigned(N-1 downto 0);
    signal count_next: unsigned(N-1 downto 0);
    signal q : STD_LOGIC_VECTOR(N-1 downto 0);
begin
  process(mclk, clr)
  begin
    if clr = '1' then
    count_reg <= (others => '0');
    elsif mclk'event and mclk = '1' then
    count_reg <= count_next;
    end if;
  end process;
  --kovetkezo allapot meghatározása konkurens when-else utasítással
  count_next <= count_reg + 1;
  --kimeneti logika
  q <= STD_LOGIC_VECTOR(count_reg);
  clk_1_hz <= q(N-1); -- ~1 Hz generált kimeneti órajel
end behav;
```

Szimuláció: egy fontos megjegyzés az órajel osztó áramkör (`clk_div`) szimulációjához. Mivel a `q(i)` bitszélessége ebben az esetben `N=26` ,amely igen nagy értéket jelent ciklus pontos szimuláció esetén (kis órajelet szeretnénk előállítani, azaz nagy periódus idejű jelet). A futtatandó szimulációs idő ebben a feladatban legalább ~50 millió ciklus x 20 ns = ~ 1 sec-ra kellene az ISim szimulátor futási idejét beállítani ahhoz, hogy a `clk_1hz` kimeneten legalább egy 1Hz-es frekvenciájú órajel ciklus megjelenjen. Ezt a számítógépen futtatva igen

időigényes lehet (főként sok szimulációs jel esetén), amely akár kb. néhány perces futtatási tesztet is jelenthet. Ezért érdemesebb lehet a `q(i)` bitszélességét, N értékét kisebbre, pl: N .= 4-re beállítani direkt módon a `clk_div` generic deklarációjában. Majd ezután kell összeállítani a tesztágyat, bele helyezni a legfelsőbb szintű, jelen esetben `clk_div` entitást és elindítani a szimulációt. (A végén természetesen a bitfájl generálása során ne felejtsük el visszaállítani az N = 26 értéket).

Megjegyeznénk továbbá, hogy egy lehetséges alternatíva a szimulációs futtatás gyorsítására, az ISim szimulátor felbontásának (resolution), vagyis a szimulációs időegység finomságának a beállítása: az alapértéken definiált [ps]-ról, „durvább” [ns]-os, vagy esetleg [us] felbontásokra (`„Time Resolution for simulation is 1ps.”`). Ezekről további részletes információt az ISim leírásban olvashatunk [*ISIM*].

2.) Az 1 Hz-es órajellel működő N-bites bináris számláló blokkszintű áramköri megvalósítása a következő *2.24. ábrán* látható:



2.24. ábra: Osztott órajellel működtetett N-bites bináris számláló blokkszintű felépítése

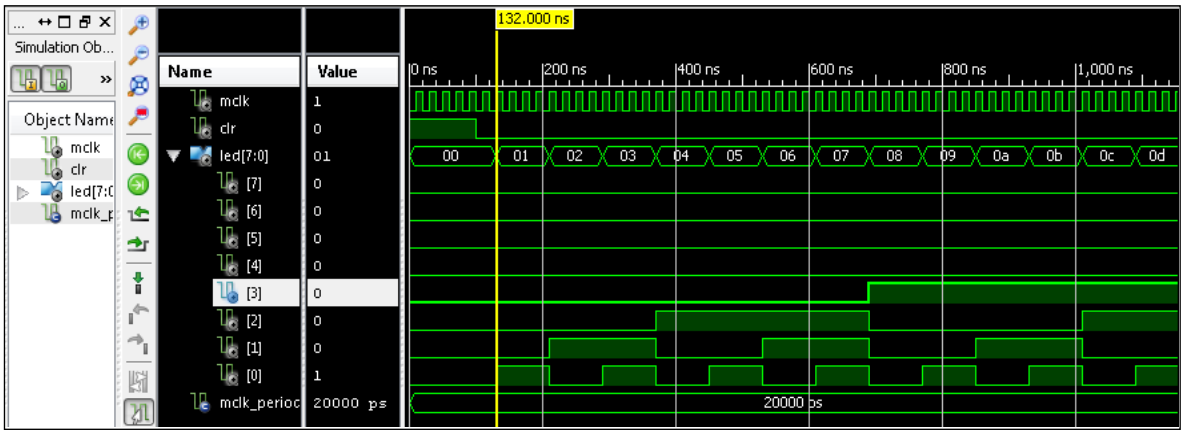
Ahogy látható, a `clk_div` modul a bejövő külső 50 MHz (FPGA – B8 lábon [*NEXYS2*]) órajelet leosztja, és a kimenetén `clk_1hz` már egy 1 Hz es órajelet kapunk. Ezt az órajelet kell egy belső jellel (`int_clk`) összekötni a `counterN_bit` modullal, amelyet már korábban ismertetettünk hagyományos N-bites bináris számláló néven. A legfelső szintű entitásnak a neve legyen `clk_div_counterN_top.vhd`. Tehát VHDL leírás a következő:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity clk_div_with_counterN_top is
generic (N : integer := 8);
 port(
      mclk : in std_logic;
      clr : in STD_LOGIC;
      LED : out std_logic_vector(N-1 downto 0)
 );
end clk_div_with_counterN_top;
```

```
architecture behav of clk_div_with_counterN_top is
component clk_div
 port (
      clr : in std_logic;
      mclk : in std_logic;
      clk_1hz : out std_logic
 );
end component;

component counterN_bit
 generic(  N : INTEGER := 8 );
 port (
      clk : in std_logic;
      clr : in std_logic;
      max_tick : out STD_LOGIC;
      q : out std_logic_vector(N-1 downto 0)
 );
end component;

signal int_clk : std_logic := '0'; --belső signal a clk összekötéséhez

begin
uut1 : clk_div
 port map( clr => clr, mclk => mclk, clk_1hz => int_clk);

uut2 : counterN_bit
 generic map ( N => 8)
 port map(   clk => int_clk, clr => clr, max_tick => open, q => LED );

end behav;
```

A fenti VHDL leírásban, mivel a `counterN_bit` nevű példányosított entitás `max_tick` portját nem akarjuk FPGA lábra kikötni, ezért kell megadnunk az **'open'** kulcsszót (ez a kulcsszó a kimeneti portot szabadon hagyja, azaz nem szükséges hozzá kötni más jeleket, vagy FPGA lábat sem). Az `uut1` (unit under test) néven példányosított `clk_div` entitás illetve az `uut2` néven példányosított `counterN_bit` entitás között a belső `int_clk jel` teremt kapcsolatot, amely tulajdonképpen a két blokk között húzótó leosztott órajelet jelenti.

(Megjegyezném, hogy érdemes kisebbre állítani a clk_div számláló N generic értékét (N=26 helyett pl. N:=2-re). Ekkor a szimulátor futtatási ideje, ha a teljes vizsgálatot meg szeretnénk tenni, lerövidül.). Ekkor az ISim szimulátorban a következő hullámforma alakot kapjuk:



2.25. ábra: a leosztott órajellel működtetett CounterN számláló viselkedési szimulációja során kapott hullámalakok (itt N=2, azaz 4 ciklus ideig tart egy-egy növekmény)

Implementáció: implementáljuk a tervet, majd töltsük le az Nexys-2 FPGA-ra, felhasználva a korábban tárgyalt lépéseket. Ehhez használjuk bemenetként clr-nek a `btn(0)` nyomógombot, illetve `mclk`-nak (master clock, nem azonos az osztott clk-val!) a külső 50 MHz-es órajelet (B8 láb), valamint a `q(7:0)` kimeneteket kössük rá a megfelelő `Led(7:0)` lábakra. Ezeket a kényszerfeltételek megfelelő beállításával tudjuk megtenni (lásd korábban tárgyalt .ucf fájl), a gyári Digilent „`Nexys2_1200General.ucf`” fájl paraméterei alapján. Az implementált tervet töltsük fel az FPGA-ra, és ellenőrizzük a működését. A Led-eknek mostmár kb. 1 sec-os késleltetéssel egymás után kell felvillanniuk, ahogyan a számláló értéke növekszik.

Feladat 3: Univerzális N-bites bináris számláló

Az univerzális számláló, ahogy a neve is mutatja sokoldalúbb alkalmazási lehetőségeket rejt. Külső jelekkel vezérelve képes lehet akár felfelé, vagy lefelé számolni; megállítható, sőt beállítható egy inicializált érték, amelyről a léptetés elkezdődhet, valamint külső szinkron `sync_clr` jel hatására törölhető is. Ha jelezni szeretnénk a lefelé számlás esetén is a számláló átfordulását a `min_tick` néven is be kell vezetni (tudjuk, hogy a `max_tick` a felső korlátot jelzi). A kívánt működés a biztosításához az előző N-bites bináris számlálót a következő jelekkel kell kiegészíteni: `sync_clr, load, dir, min_tick`. A `dir` jel (direction) a felfelé, vagy lefelé számlálást irányát adja meg: `dir = '1'` felfelé számoljon, míg `dir = '0'` visszafelé számoljon. `Load` jel biztosítja a számláló aktuális kezdőértékének párhuzamos betöltését, amelyről indulhat a számolás a `dir`-el megadott irány szerint léptetve. A számláló entitás neve legyen: `univ_counterN`. Az univerzális számláló legyen N=4 bites, **generic**-el megadva. A számláló VHDL leírása a következő:

```
-- Univerzális N-bites bináris számláló
library IEEE;
use IEEE.STD_LOGIC_1164.all;
USE ieee.numeric_std.ALL;

entity univ_counterN is
  generic(N : integer := 4);
  port(
   clr : in STD_LOGIC;
   clk : in STD_LOGIC;
   d: in std_logic_vector(N-1 downto 0);
   syn_clr , load, dir : in STD_LOGIC;
   max_tick, min_tick : out STD_LOGIC;
   q : out STD_LOGIC_VECTOR(N-1 downto 0)
      );
end univ_counterN;
```

```
architecture behav of univ_counterN is
    signal count_reg: unsigned(N-1 downto 0);
    signal count_next: unsigned(N-1 downto 0);
begin
  process(clk, clr)
  begin
    if clr = '1' then
    count_reg <= (others => '0');
    elsif clk'event and clk = '1' then
    count_reg <= count_next;
    end if;
  end process;
  --kovetkezo allapot meghatározása konkurens when-else utasítással
  count_next <= (others =>'0') when syn_clr='1' else
                unsigned(d)    when load='1' else
                count_reg + 1  when dir='1' else
                count_reg - 1  when dir='0' else
                count_reg;
  --kimeneti logika
  q <= STD_LOGIC_VECTOR(count_reg);
  max_tick <= '1' when count_reg = (2**N-1) else '0';
  min_tick <= '1' when count_reg = 0 else '0';
end behav;
```

Szimuláció és implementáció: gyakorlásként ehhez a feladathoz is készítsünk egy tesztágyat, helyezzük el benne a számlálót, majd szimuláljuk le a viselkedését ISim segítéségével, végül pedig implementáljuk a tervet és töltsük le FPGA-ra.

Feladat 4: Modulo-m N-bites bináris számláló

Tervezzünk egy olyan ún. *'modulo-m'* bináris számlálót, amely '0'-tól 'm−1'-ig számol, majd pedig átfordul. A paraméterezhető modulo-m számláló két generic-et is használ: egyet a maradékos osztás meghatározásához (M), egyet pedig a bitszélesség megadásához (N), és amelynek egyenlőnek kell lennie $\lceil \log_2 M \rceil$-el. A modulo-m számláló neve legyen `modulo_m_counter`. A számláló VHDL leírása a következő, ahol az N=4, illetve M=10 (azaz 4-bites, modulo-10 számláló):

```
-- Modulo-m N-bites bináris számláló
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity modulo_m_counter is
   generic(
      N: integer := 4;     -- számláló bitszélessége
      M: integer := 10     -- modulo-M
  );
   port(
      clk, reset: in std_logic;
      max_tick: out std_logic;
      q: out std_logic_vector(N-1 downto 0)
   );
end modulo_m_counter;
```

```
architecture behav of modulo_m_counter is
   signal count_reg: unsigned(N-1 downto 0);
   signal count_next: unsigned(N-1 downto 0);
begin
   -- regiszter
   process(clk, reset)
   begin
      if (reset='1') then
         count_reg <= (others=>'0');
      elsif rising_edge(clk) then
         count_reg <= count_next;
      end if;
   end process;
   -- next-state logikai, konkurens when-else értékadással
   count_next <= (others=>'0') when count_reg = (M-1) else
                 count_reg + 1;
   -- kimeneti logika, konkurens értékadásokkal
   q <= std_logic_vector(count_reg);
   max_tick <= '1' when count_reg=(M-1) else '0';
end behav;
```

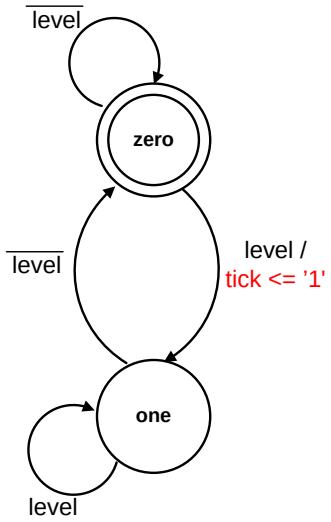
A fenti kód szerint, amikor a modulo-M számláló eléri az ’M −1’-et átfordul, és új értéke ’0’ lesz, minden más esetben növekszik az értéke 1-el.

Szimuláció és implementáció: gyakorlásként ehhez a feladathoz is készítsünk egy tesztágyat, helyezzük el benne a számlálót, majd szimuláljuk le a viselkedését ISim segítéségével, végül pedig implementáljuk a tervet és töltsük le FPGA-ra

Feladat 5: Felfutó-élt detektáló áramkör Mealy esetben

Ebben a feladatban tervezzünk meg és szimuláljuk, valamint implementáljuk egy Mealy modell szerint működő felfutó élt detektáló áramkört.

A felfutó-élt detektáló áramkör egy olyan építőelem, amely 1 órajel ciklus ideig tartó ún.'tüskét', impulzust (tick) generál, amikor a bemenet '0' → '1' változását érzékeli. Ezt általában nem egy pediódikusan és gyorsan, hanem lassan váltakozó jel esetén alkalmazzák, pl. valamilyen 'trigger' (indítóimpulzus) esemény vizsgálatára. Az entitás nevének adjuk meg: `edge_detect_mealy`. Az állapotgépnek két állapota van, a `zero`, és a `one`, amely a nullás és egyes állapot mellett vizsgálja a `level` (jelszint) értékének változását. Az állapotokhoz saját felsorolt típust definiálunk, és ezzel a típussal két belső jelet (`state_reg`, `state_next`), adunk meg.



2.26. ábra: Felfutó élt detektáló áramkör állapotdiagramja Mealy modell esetén

A VHDL arra is lehetőséget biztosít, hogy ebben a feladatban a következő állapot (next-state logic) és kimeneti logikát egyetlen kombinált process() állításban, és nem két elkülönült szekvenciális folyamatban adjuk meg (mint a korábbi példákban):

```
-- felfutó élt detektáló áramkör mealy modellje
library ieee;
use ieee.std_logic_1164.all;

entity edge_detect_mealy is
   port(
      clk, reset: in std_logic;
      level: in std_logic;
      tick: out std_logic
   );
end edge_detect_mealy;
```

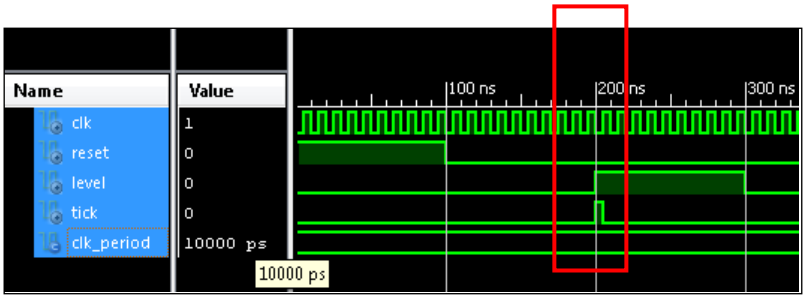
```
architecture mealy_arch of edge_detect_mealy is
   --saját felsorot típus definiciója: zero állapot, one állapotra
   -- zero = 0, one = 1 felsorolt típus értékei
   type state_type is (zero, one);
   signal state_reg, state_next: state_type;
begin
   -- állapotregiszter
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= zero;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;

-- következő állapot és kimeneti logika kombinált megvalósítása
-- Mealy esetben
   process(state_reg,level)
   begin
      state_next <= state_reg;
      tick <= '0';
      case state_reg is
         when zero=>
            if level= '1' then
               state_next <= one;
               tick <= '1';
            end if;
         when one =>
            if level= '0' then
               state_next <= zero;
            end if;
      end case;
   end process;
end mealy_arch;
```

Szimuláció és implementáció: gyakorlásként ehhez a feladathoz is készítsünk egy tesztágyat, helyezzük el benne a `edge_detect_mealy` néven definiált entitásunkat majd szimuláljuk le a viselkedését ISim segítségével, végül pedig implementáljuk a tervet, és töltsük le FPGA-ra,

ahol a `level`, `reset` mint külső input jeleket kössük egy nyomogombra (vagy esetleg kapcsolóra) a kimeneti `tick` jelet pedig a `Led(0)`-ra.

Megjegyzés: ahhoz hogy a felvillanás időtartama érzékelhető legyen, alkalmazzuk a korábban megismert `clk_div` órajel osztó áramkört (valójában egy számlálót), amelynek leosztott órajel kimenetét kössük be az él-detektáló áramkör `clk` órajel bemenetére.

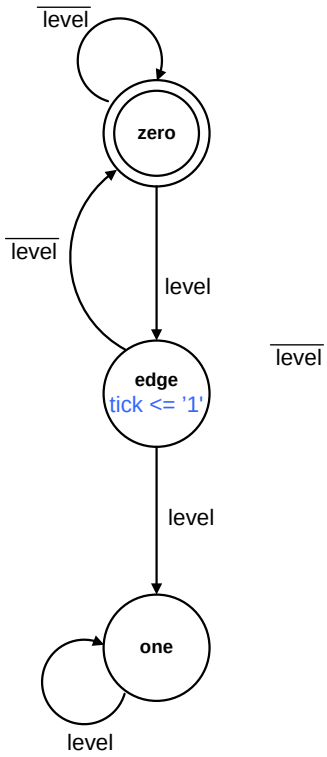
Szimulációs eredményként a következőt kell kapni:



2.27. ábra: Felfutó élt detektáló áramkör szimulációs eredménye Mealy modell esetén

Feladat 6: Felfutó-élt detektáló áramkör Moore esetben

Ebben a feladatban tervezzünk meg és szimuláljuk, valamint implementáljuk Moore modell szerint a felfutó-élt detektáló áramkört Az entitás nevének adjuk meg: `edge_detect_moore`. Az előző Moore modellhez képest adjunk most meg három állapotot: (`zero, one, edge`), tehát külön azt az állapotot is, amikor éppen felfutó él van a vizsgált jel esetén.



2.28. ábra: Felfutó-élt detektáló áramkör állapotdiagramja Moore modell esetén

```
-- felfutó élt detektáló áramkör Moore modellje
library ieee;
use ieee.std_logic_1164.all;

entity edge_detect_moore is
   port(
      clk, reset: in std_logic;
      level: in std_logic;
      tick: out std_logic
   );
end edge_detect_moore;

architecture moore_arch of edge_detect_moore is
   --saját felsorot típus definiciója: zero állapot, one állapotra
   --edge: él állapot
   -- zero = 0, edge = 1, one = 2 felsorolt típus értékei
   type state_type is (zero, edge, one);
   signal state_reg, state_next: state_type;
begin
   -- állapotregiszter
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= zero;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
```

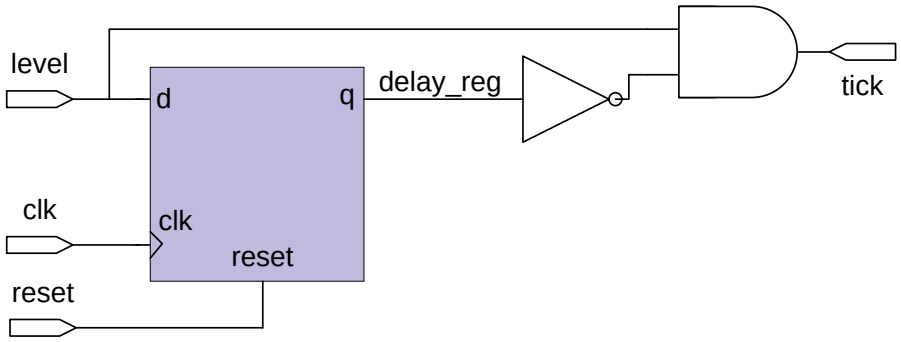
```
-- következő állapot és kimeneti logika kombinált megvalósítása
-- Moore esetben
process(state_reg,level)
   begin
      state_next <= state_reg;
      tick <= '0';
      case state_reg is
         when zero=>
            if level= '1' then
               state_next <= edge;
            end if;
         when edge =>
            tick <= '1';
            if level= '1' then
               state_next <= one;
            else
               state_next <= zero;
            end if;
         when one =>
            if level= '0' then
               state_next <= zero;
            end if;
      end case;
   end process;
end moore_arch;
```

Szimuláció és implementáció: gyakorlásként ehhez a feladathoz is készítsünk egy tesztágyat, helyezzük el benne a `edge_detect_moore` néven definiált entitásunkat majd szimuláljuk le a viselkedését ISim segítségével, végül pedig implementáljuk a tervet és töltsük le FPGA-ra, ahol a `level`, `reset` mint külső input jeleket kössük egy nyomogombra (vagy esetleg kapcsolóra) a kimeneti `tick` jelet pedig a `Led(0)`-ra.

Megjegyzés: ahhoz hogy a felvillanás időtartama érzékelhető legyen, alkalmazzuk a korábban megismert `clk_div` órajel osztó áramkört, amelynek leosztott órajel kimenetét kösse be az éldetektáló áramkör `clk` órajel bemenetére.

Feladat 7: Felfutó-élt detektáló áramkör megvalósítása egyszerű kapu-szintű strukturális VHDL leírással

Ebben a feladatban tervezzünk meg a korábbi felfutó-élt detektáló áramkör egy strukturális kapu-szintű VHDL leírását a megadott *2.29. ábra* szerint, `edge_detect_gate` néven:



2.29. ábra: Felfutó-élt detektáló áramkör megadása egyszerű kapuszintű kapcsolással

A korábbi Mealy, Moore féle FSM modell leírással megadott felfutó élt detektáló áramkört egy egyszerűbb, kapuszintű strukturális VHDL leírás formájában is meg lehet adni. Amikor például, az FSM `zero` állapotban van, és a bemeneti szint (`level`) ’1’ –re változik, a kimenet is azonnal megváltozik (Mealy modell esetén). Az FSM ezután `one` állapotba lép át, a `clk` jel következő felfutó élének hatására, majd pedig a kimenet (`tick`) is megváltozik. A jelterjedési és megszólalási időknek köszönhetően még a következő felfutó `clk`-órajel fázisban is a `tick` ’1’-es logikai értéken marad, mielőtt visszaállna ’0’-ra. A *2.29. ábrán* lévő kapuszintű áramköri diagram bemenetére ha ’0’-t vagy ’1’-et adunk, az reprezentálja az FSM modell `zero` illetve `one` állapotait, azaz az elemi D-FF állapottáblájának működése szerint.

```
-- felfutó élt detektáló áramkör egyszerű strukturális kapuszintű leírása
library ieee;
use ieee.std_logic_1164.all;

entity edge_detect_gate is
   port(
      clk, reset: in std_logic;
      level: in std_logic;
      tick: out std_logic
   );
end edge_detect_gate;

architecture gate_level_arch of edge_detect_gate is
   signal delay_reg: std_logic;
begin
```

```
   --itt nincsenek definiált állapotok
   -- késleltető regiszter direkt módon adott, amely megvalósítja egy FSM
   -- one illetve zero allapotait.
   process(clk,reset)
   begin
      if (reset='1') then
         delay_reg <= '0';
      elsif (clk'event and clk='1') then
         delay_reg <= level;
      end if;
   end process;
   -- dekódoló/ kimeneti logika megvalósítása
   tick <= (not delay_reg) and level;
end gate_level_arch;
```

Szimuláció és implementáció: gyakorlásként ehhez a feladathoz is készítsünk egy tesztágyat, helyezzük el benne az `edge_detect_gate` néven definiált entitásunkat majd szimuláljuk le a viselkedését az ISim segítségével, végül pedig implementáljuk a tervet és töltsük le FPGA-ra, ahol a `level`, `reset` mint külső input jeleket pedig kössük egy nyomógombra (vagy esetleg kapcsolóra) a kimeneti `tick` jelet pedig a `Led(0)`-ra.

Megjegyzés: ahhoz hogy a felvillanás időtartama érzékelhető legyen, alkalmazzuk a korábban megismert `clk_div` órajel osztó áramkört, amelynek leosztott órajel kimenetét kösse be az éldetektáló áramkör `clk` órajel bemenetére.

2.8. Komplex rendszerek tervezése VHDL segítségével

A korábbi fejezetekben tárgyalt egyszerűbb VHDL leírásokat felhasználva, egyre nagyobb komplexitású, és többszintű hierarchiával rendelkező digitális rendszerek implementálhatók VHDL nyelven, különböző programozható digitális áramkörök alkalmazásával. Minden esetben érdemes olyan modulokat tervezni, amelyeket a későbbi fejlesztések során is fel tudunk használni: emiatt például érdemes lehet az I/O port-listákat minden hierarchia szinten egyeztetni azért, hogy a komponens deklarációknál és példányosításoknál ne jelentsen problémát az elnevezésük, típusuk, bitszélességük stb. (Megjegyeznénk, hogy a bitszélességek megadására a korábban tárgyalt generikusokat, vagy constant típusokat lehet alkalmazni).

A modularitás végett ebben a komplex feladatban a korábban tárgyalt fontosabb VHDL nyelvi konstrukciókat, mint példányosított komponenseket használjuk fel.

Egy konkrét példán keresztül szemléltetjük a bonyolultabb rendszerek tervezését VHDL-ben: a VGA vezérlő segítségével kimeneti kép generálása, illetve a későbbi *2.9. fejezetben* a szabványos UART soros port segítségével adatok fogadását, és küldését vizsgáljuk meg egy FPGA-s céleszköz és gazda számítógép között.

2.8.1. Esettanulmány: VGA vezérlő megvalósítása

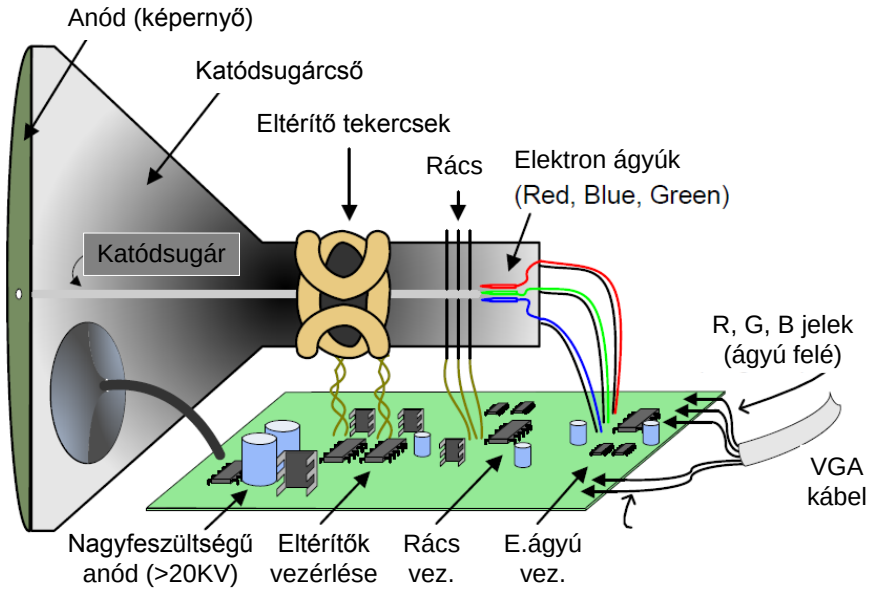
Ahhoz hogy VHDL nyelven egy VGA-vezérlőt meg tudjuk tervezni először a VGA szabványt, annak hátterét, és pontos időzítését is tanulmányoznunk kell. Az ismertetésre kerülő legtöbb ábra az megtalálható Digilent Nexys-2 kártya [*NEXYS2*] referencia adatlapjában.

VGA szabvány

A VGA (Video Graphic Array) jeleket és időzítésüket a VESA (Video Electronics Standards Association) – mint a számítógépes megjelenítéssel foglalkozó legnagyobb elektronikai gyártókat tömörítő szabványügyi szervezet – dolgozta ki és fogadtatta el az 1980-as évek végén [*VESA*], amely mára a számítógépes grafika és megjelenítés területén széles körben támogatott szabvánnyá vált. A VGA-vezérlő tervezésekor az egyes időzítési paraméterek beállítását a VESA oldaláról is elérhetjük, amelyek táblázat formájában ingyenesen letölthetők. A VESA CVT – Coordinated Video Timing Format v1.1 (CVT_d6r1.xls) olyan VGA megjelenítési formátumokat tartalmazó adatlap, amelyből a kívánt felbontás, és a vertikális képfrissítési frekvencia alapján generálhatók a szükséges időzítési paraméterek, amelyeket különböző felbontási módok megadásakor sem kell feleslegesen. Ennek ellenére előfordulhat, hogy adott felbontás beállítása mellett kis mértékben növelni, vagy csökkenteni kell a szinkronizáló jelek táblázatban megadott konkrét értékeit, azért, hogy a monitoron a kisebb (1-2 pixeles) csúszásokat korrigáljuk. A mai VGA kijelzők (CRT/LCD) képesek a különböző képfelbontási módokat akár automatikusan is kezelni, amelyeket a VGA vezérlő-áramkör a különböző időzítési paraméterek megfelelő hangolásával határoz meg.

CRT monitor működése

A következő *2.30. ábra* egy hagyományos CRT (Cathode Ray Tube) katódsugárcsöves monitor felépítését mutatja, amelynek a funkcionális működését használják mindmáig a VGA interfészek tervezési feladatai során. Itt kell megemlíteni, hogy a korai CRT, és a korszerű LCD monitorok időzítése és vezérlő jelei hasonlóan kezelhetőek.



2.30. ábra: hagyományos CRT monitor elvi felépítése

Az CRT monitor elvi felépítésén is látható, hogy az elektron ágyúk (vagy katódok az R-G-B komponensekre külön-külön) egy-egy fókuszálható elektron nyalábot bocsátanak ki, amelyek a vákuum csövön keresztül haladva végül a fluoreszcens anyaggal bevont képcső belső falának ütköznek, és fényt emittálnak. A foszforeszkáló anyaggal bevont képernyő felületére eközben ~ 20 kV-os anódfeszültség van kapcsolva. Az elektron ágyútól az anódig megtett úton az elektron nyalábot számos külső hatás éri: szabályozó rácson kereszül, majd függőleges (vertikális), illetve vízszintes (horizontális) eltérítő tekercsekkel szabályozzák, vezérlik az elektron sugár mozgását (pozícionálását). Ezek közül a két utóbbi vezérlő jel, nevezetesen a horizontális, és vertikális eltérítések azok, amelyek pontos időzítését kell elsődlegesen a VHDL-implementációban is megvalósítani.

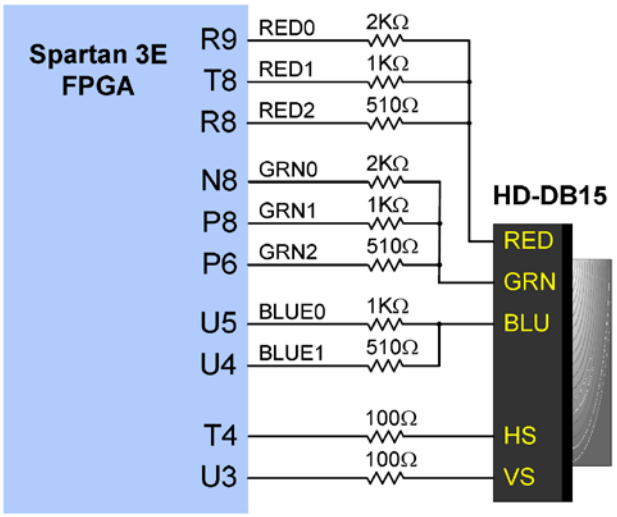
A képernyőn az információ ’raszter’ minta szerint, átlósan jeleníthető meg: horizontálisan balról-jobbra, míg vertikálisan fentről-lefelé kell mozgatni, eltéríteni a katód sugarakat. A képernyő tartalmát mindig csak akkor frissíthetjük újból, amikor minden raszter ponton sorban végighaladtunk. Természetesen a nyaláb újboli horizontális – vertikális pozicionálásának (azaz a sor végéről a következő sor elejére ugrásnak) is van némi ideje, ezeket a szakirodalomban többféleképpen ’retrace’ vagy ’blanking’ periódusoknak is szokás nevezni, amikor tehát tényleges információ nem jelenik meg a képernyőn.

Általánosan a képfelbontást az elektron nyaláb mérete, a nyaláb frekvencia modulációja és a képfrissítés frekvenciája együttesen határozzák meg. A N×M-es méretű képfelbontás esetén az N jelöli a sorokat, míg M az oszlopokat. A képi elemet pixelnek, vagy képpontnak nevez-

zük. A mai tipikus képernyő felontások 320-1920 oszlopból, illetve 200-1080 sorból állnak (ahol 320×200 a szabványos QVGA, míg 1920×1080 a HD-1080 felbontásokat jelentik). A mostani példánkban egy szabványos VGA 640×480 felbontású és 60Hz es képfrissítési beállítással implementáljuka VGA vezérlőt.

VGA port a Nexys-2 kártyán

A Digilent Nexys-2 kártyán a következő ábra szerint [NEXYS2] az R, G, B színkomponenseket, illetve a HS, VS (horizontális és vertikális szinkron) jeleket kell csupán az FPGA lábai felé bekötni (kényszerfeltételek megfelelő megadásával .ucf fájlban). A VGA kimenet 8 színkomponensből áll, amelyből 3-3 a Red, illetve a Green, valamint 2- a Blue színcsatornát jelenti: ezért a 8-bites videó jel segítségével 2^N, azaz egyszerre 256 különböző színárnyalatot tudunk egyidőben megjeleníteni. A további két vezérlő szinkron jel a Vertical Synch (VS), illetve a Horizontal Synch (HS) jeleket definiálja. Ezek a jelek fizikailag egy szabványos D-Sub 15 lábú analóg VGA csatlakozóra vannak bekötve (lásd alábbi *2.31. ábra*).



2.31. ábra. VGA kimenet jelei és FPGA fizikai lábai közötti kapcsolat soros ellenállásokon keresztül (Digilent Nexys-2 fejlesztőkártya [NEXYS2])

VGA szinkronizáció

A VGA szinkronizáló áramkör feladata, hogy az egyes video jelek időzítését betartva megfelelő időben generálja a vertikális és horizontális szinkron jeleket, amelyeket az FPGA lábain keresztül a VGA csatlakozó (DB15) felé egymás után küld el. A `h_sync` horizontális szinkron jel azt az időintervallumot specifikálja, mialatt a sugárnyaláb balról-jobbra végigpásztáz egy soron, míg a `v_sync` vertikális szinkron jel azt az időegységet szabja meg, mialatt a sugárnyaláb fentről-lefelé haladva végigpásztáz egy teljes, esetünkben most VGA felbontású képkeretet (frame-t). A `video_en` jel ahhoz kell, hogy jelezzük, mikor van érvényes, ténylegesen megjeleníthető képi-információ a pásztázás során.

Videó órajel és szinkron jelek időzítése

A 640×480-as VGA felbontáshoz ~25MHz-es videó órajelet (video clock-rate) kell beállítani 60 Hz-es vertikális és 60 KHz-es horizontális frissítési frekvencia mellett. Ezekhez a paraméterekhez kell viszonyítani a szinkronizálciós (timing) adatoknak a pontos beállításait, amelyeket akár a Nexys-2 gyártó oldaláról, akár a VESA szabványügyi szervezet oldaláról érhetünk el [NEXYS2], [VESA]. Ez azt is jelenti egyben, hogy az FPGA 50 MHz-es külső bemeneti órajelét valamilyen módon 25 MHz-re kell leosztani. Kétféleképpen is megtehető ez: vagy egy órajel osztót építünk számláló segítségével (lásd korábbi *2.7. fejezet Feladat 2.*), vagy egy Xilinx DCM primitívet használunk a tetszőleges órajel előállításához Xilinx CoreGenerator segítségével. Jelenlegi példánkban az első megvalósítást követjük, tehát egy egyszerű órajel osztót építünk N=1 bites számláló segítségével.

A 640×480 felbontás az aktív, azaz látható tartományt (régiót) jeleti, amelyen kívül található az ún. nem-látható (fekete régió) tartomány, amikor a tekercsek és a sugárnyalábok megfelelő beállási idejét kell figyelembe venni. Tehát valójában egy 800×525 méretű képkockát kell kezelni, amelyből az aktív régiót a VGA 640×480-as felbontású leszűkített területe jelenti.

A VGA 640×480 szabvány órajelként 25 MHz-et kövekel meg, és a következő paraméterekkel kell számolnunk:

- *pn (pixel number)* = 800 képpont / képsor,
- *ln (line number)* = 525 képsor / képkocka,
- *sn (screen number)* = 60 képkocka / másodperc.

Ezekből a paraméterekből a pixelek frissítési gyakorisága a következő módon számítható:

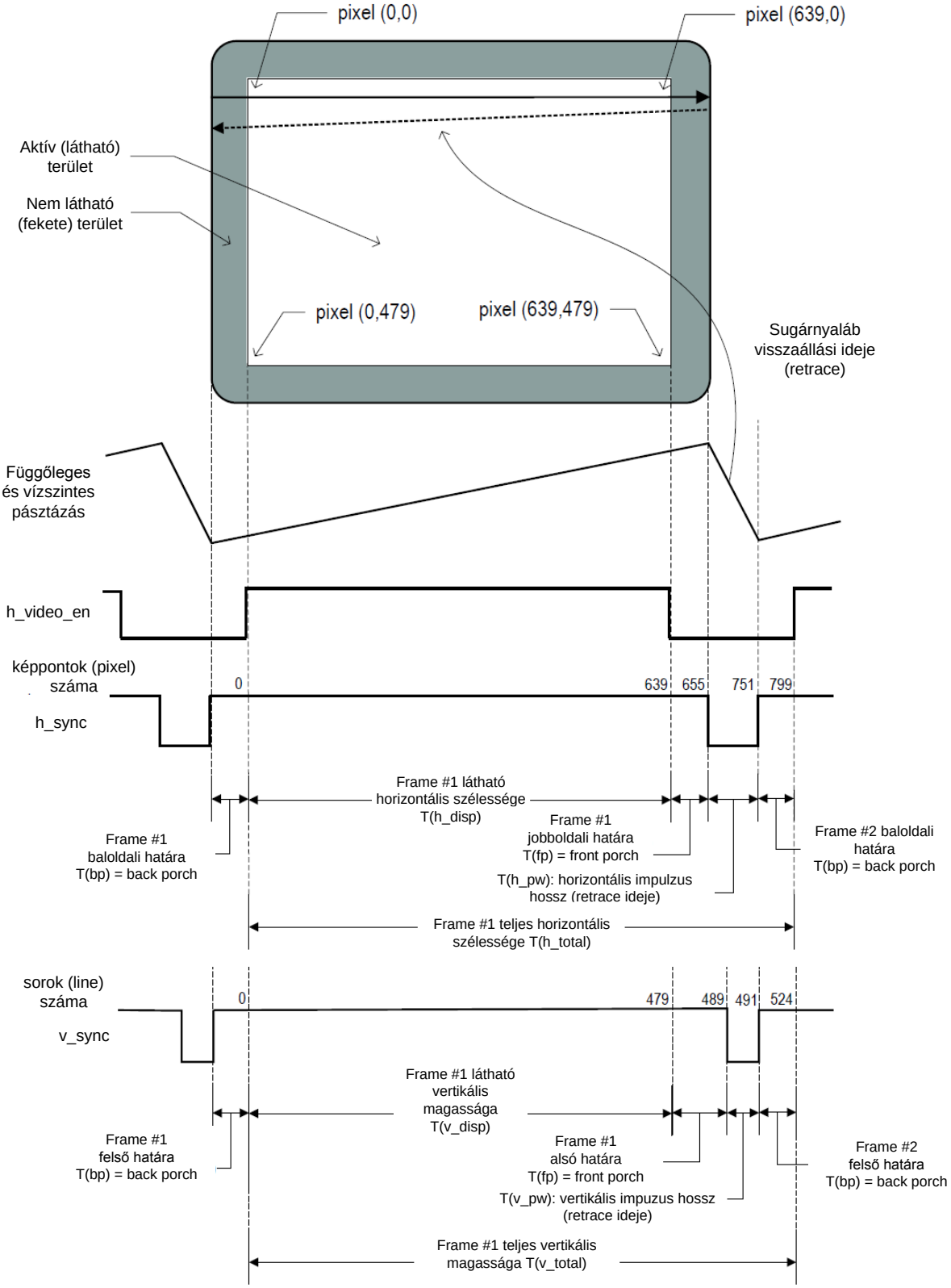
- *ps (pixel_rate)* = pn * ln * sn ≈ 25 millió képpont / másodperc.

Horizontális szinkronizáció

A `h_sync` a horizontális szinkron jel, melynek teljes intervallumát 800 képpontra kell állítani a VGA szabvány szerint. A `h_sync` jelet egy speciális modulo-800 számlálóval lehet megvalósítani. A `h_sync` jel előállítását a teljes horizontális tartományon belül a következő fontosabb fázisokra lehet felosztani:

- T(h_total): az teljes időegység, vagy ekvivalens pixelszám, amelybe mind a képernyő látható (aktív), mind pedig a nem látható régiói is beletartoznak. (Megjegyzés: A horizontális jelek szinkronizáló pareméterei esetén ezeket az időegységeket mindig a velük ekvivalens képpont számokkal (800 pixel) feleltetjük meg.)
- T(h_disp): megjelenítési (display) idő, amely időegység alatt a képernyő aktív (látható) tartományában a képpontok (640 pixel) megjelennek.
- T(h_pw): horizontális impulzus idő, amely időegység alatt az elektronányú a jobb oldalról visszatér (retrace) a képernyő bal oldalára. A `video_en` jelet ezen tartomány alatt inaktív jelszinten kell tartani a fekete, nem-látható terület miatt (96 pixel)
- T(h_fp): vertikális front porch érték: képkocka jobboldali határa, azaz T(h_pw) impulzus idő előtti intervallum. A `video_en` jelet ezen front porch tartomány alatt inaktív jelszinten kell tartani a fekete, nem-látható terület miatt. (16 pixel)

- T(h_bp): vertikális back porch érték: képkocka baloldali határa, azaz T(h_pw) impulzus idő utáni intervallum. A `video_en` jelet ezen back porch tartomány alatt inaktív jelszinten kell tartani a fekete, nem-látható terület miatt (48 pixel)



2.32. ábra: Horizontális és vertikális szinkronizáció időzítés VGA mód (640×480) esetén

Vertikális szinkronizáció

A horizontális szinkronizációs jelekhez hasonlóan itt is intervallumokra osztjuk a `v_sync` vertikális szinkron videó jelet, melynek teljes intervallumát 525 sorra kell beállítani a VGA szabvány szerint (ismerve azt, hogy 1 képsor 800 teljes képpontból áll). A `v_sync` jelet egy speciális modulo-525 számlálóval lehet megvalósítani. A `v_sync` jel előállítását a teljes vertikális tartományon belül a következő fontosabb fázisokra lehet felosztani:

- T(v_total): az a teljes időegység, vagy ekvivalens sorszám, amelybe mind a képernyő látható (aktív), mind pedig a nem-látható régiói is beletartoznak. (Megjegyzés: a vertikális jelek szinkronizáló pareméterei esetén ezeket az időegységeket mindig a velük ekvivalens sorok számával (525 képsor) feleltetjük meg.)
- T(v_disp): megjelenítési (display) idő, amely időegység alatt a képernyő aktív (látható) tartományában megjelennek a képsorok (480 képsor).
- T(v_pw): horizontális impulzus idő, amely időegység alatt az elektronáyú a jobb oldalról visszatér (retrace) a képernyő bal oldalára. A `video_en` jelet ebben a tartományon belül inaktív jelszinten kell tartani a fekete, nem-látható terület miatt (2 képsor).
- T(v_fp): vertikális front porch érték: képkocka jobboldali határa, T(h_pw) impulzus idő előtti intervallum. A `video_en` jelet ezen front porch tartomány alatt inaktív jelszinten kell tartani a fekete, nem-látható terület miatt. (10 képsor).
- T(v_bp): vertikális back porch érték: képkocka baloldali határa, T(h_pw) impulzus idő utáni intervallum. A `video_en` jelet ezen back porch tartomány alatt inaktív jelszinten kell tartani a fekete, nem-látható terület miatt (33 képsor).

A horizontális és vertikális VGA szinkronizációs jeleket a következő *2.32. ábra* szemlélteti.

A következő *2.14. táblázatban* összegyűjtve találhatóak a VGA-vezérlő VHDL leírásához szükséges legfontosabb időzítési paraméterek, konkrét értékeikkel:

2.14. táblázat: VGA (640×480) szinkron időzítési paraméterei

VGA (640×480)@60Hz/25 MHz: szinkron időzítési paraméterek				
jelölés	**név**	**pixel (órajel)**	**Idő**	**Kép-sor**
T(h_total)	horizontális teljes régió (látható + nem látható)	**800**	32 us	-
T(h_disp)	horizontális aktív (látható) régió	**640**	25.6 us	-
T(h_pw)	horizontális impulzus hossz (h_sync ideje)	**96**	3.84 us	-
T(h_fp)	horizontális front porch	**16**	640 ns	-
T(h_bp)	horizontális back porch	**48**	1.92 us	-
T(v_total)	vertikális teljes régió (látható + nem látható)	416,800	16.7ms	**525**
T(v_disp)	vertikális aktív (látható) régió	384,000	15.36ms	**480**
T(v_pw)	vertikális impulzus hossz (v_sync ideje)	1,600	64 us	**2**
T(v_fp)	vertikális front porch	8,000	320 us	**10**
T(v_bp)	vertikális back porch	23,200	928 us	**33**

VGA időzítő-szinkronizáló VHDL implementációja

A fejezetben ismertetett VGA szinkronizációs időzítési paramétereknek megfelelően (lásd *2.14. táblázat*) tervezzük meg a VGA vezérlőt VHDL leíró nyelv segítségével. Az entitás neve legyen: `VGA_sync`. A szinkronizációs időzítési paramétereket definiáljuk integer adattípusú konstansként.

A VGA időzítő-vezérlő „lelke” két különböző, de ágymástól függő számláló: egy a horizontális, egy pedig a vertikális irányokban számol. Horizontálisan pixeleket számolunk a videó órajel ciklusoknak megfelelően, vertikálisan pedig a pixelekből álló sorokat (amikor a horizontális irányban elértük a sor végét, akkor növeljük, inkrementáljuk 1-el a vertikális számláló értékét is). A VGA időzítő-szinkronizáló megvalósítására a következő VHDL leírás mutat egy lehetséges módszert:

```
-- VGA időzítő szinkronizáló implementációja
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;

entity VGA_sync is
   port(
      mclk, rst: in std_logic;
      hs, vs: out std_logic;
      video_en : out std_logic
    );
end VGA_sync;

architecture Behavioral of VGA_sync is
   -- VGA 640x480 időzítési paraméterek: konstansok használatával
   --más felbontáshoz átparaméterezhetőek az értékek
   constant C_HDISP : integer:= 640; --horizontal display time
   constant C_HFP   : integer:=  16; --horiz. front porch
   constant C_HBP   : integer:=  48; --horiz. back porch
   constant C_HPW   : integer:=  96; --horiz. pulse with (retrace time)

   constant C_VDISP : integer:= 480; --vertical display time
   constant C_VFP   : integer:=  10; --v. front porch
   constant C_VBP   : integer:=  33; --v. back porch
   constant C_VPW   : integer:=   2; --vert. pulse with (retrace time)
```

```
   --belső számláló jelek: max 10 biten tárolunk     CEIL(log2(800))!
   signal v_count_reg, v_count_next: unsigned(9 downto 0); --525
   signal h_count_reg, h_count_next: unsigned(9 downto 0); --800
   -- kimeneti regiszterelt jelek
   signal v_sync_reg, h_sync_reg: std_logic;
   signal v_sync_next, h_sync_next: std_logic;
   -- belső állapot jelek ,vert. és horizontális timing-nal hol van vége
   signal h_end, v_end : std_logic;

   signal pixel_tick : std_logic;  --belső jel – mclk órajel szintjét
   --vizsgáljuk

begin

   -- 25 MHz pixel_tick, ami a későbbi feltétel vizsgálathoz
   -- a külső órajel '1' vagy '0' szintjét vizsgálja
   pixel_tick <= '0' when mclk='1' else '1';

   process (mclk, rst)
   begin
      -- resetelni minden regisztert
      if rst='1' then
         v_count_reg <= (others=>'0');
         h_count_reg <= (others=>'0');
         v_sync_reg <= '0';
         h_sync_reg <= '0';
      elsif (mclk'event and mclk='1') then
         v_count_reg <= v_count_next;
         h_count_reg <= h_count_next;
         v_sync_reg <= v_sync_next;
         h_sync_reg <= h_sync_next;
      end if;
   end process;
```

```
-- státusz jelek generálása/ számlálók végállapotának vizsgálata
h_end <=  -- ha elérjük a horiz. számláló végét ’1’
   '1' when h_count_reg=(C_HDISP + C_HFP + C_HBP + C_HPW-1) else --799
   '0';
v_end <=  -- ha elérjük a vert. számláló végét ’0’
   '1' when v_count_reg=(C_VDISP + C_VFP + C_VBP + C_VPW-1) else --525
   '0';

-- horizontális számláló növelése, majd 0-ázása amennnyiben
---elértük a végértéket, átfordul.
process (h_count_reg,h_end,pixel_tick)
begin
   if pixel_tick = '1' then  -- 25 MHz
      if h_end='1' then
         h_count_next <= (others=>'0');
      else
         h_count_next <= h_count_reg + 1;
      end if;
   else
      h_count_next <= h_count_reg;
   end if;
end process;

-- vertikális számláló növelése, majd 0-ázása amennnyiben
---elértük a végértéket, átfordul.
process (v_count_reg,h_end,v_end,pixel_tick)
begin
   if pixel_tick = '1' and h_end='1' then
      if (v_end='1') then
         v_count_next <= (others=>'0');
      else
         v_count_next <= v_count_reg + 1;
      end if;
   else
      v_count_next <= v_count_reg;
   end if;
end process;
```

```
  -- horizontalis and vertikalis szinkron jelek, regiszteretlek a tüskék –
   -- elkerülése végett
   h_sync_next <=
      '1' when (h_count_reg >= (C_HDISP + C_HFP))                  --656
           and (h_count_reg <= (C_HDISP + C_HFP + C_HPW - 1)) else --751
      '0';
   v_sync_next <=
      '1' when (v_count_reg >= (C_VDISP+C_VFP))                --490
           and (v_count_reg <= (C_VDISP+C_VFP+C_VPW-1)) else  --491
      '0';

   -- video enable (aktív régióban megjelenítés engedélyezése)
   video_en <=
      '1' when (h_count_reg < C_HDISP) and (v_count_reg < C_VDISP) else
      '0';

   -- regiszterelt vertikalis es horizontalis szinkron jelek kimenetekre
   --kötése
   hs <= h_sync_reg;
   vs <= v_sync_reg;

end Behavioral;
```

VGA időzítő-szinkronizáló szimulációja

Készítsük el a tesztágyat a fent ismertetett VGA időzítő-szinkronizáló egységünknek, `VGA_sync_tb` néven melybe helyezzük el a `VGA_sync` modulunkat, majd példányosítsuk pl. `uut (unit under test)` néven. A szimulátor `mclk_period` órajel periódusának állítsunk be 40 ns-ot (= 25 MHz-et), amely mint tudjuk, megegyezik a VGA videó-órajelével.

```
-- VGA időzítő vezérlő tesztágya
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY VGA_sync_tb IS
END VGA_sync_tb;

ARCHITECTURE behavior OF VGA_sync_tb IS
```

```
    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT VGA_sync
    PORT(
         mclk : IN  std_logic;
         rst : IN  std_logic;
         hs : OUT  std_logic;
         vs : OUT  std_logic;
         video_en : OUT  std_logic
        );
    END COMPONENT;

   --Inputs
   signal mclk : std_logic;
   signal rst : std_logic := '0';

      --Outputs
   signal hs : std_logic;
   signal vs : std_logic;
   signal video_en : std_logic;

   -- Clock period definitions
   constant mclk_period : time := 40 ns; -- 25 MHz VGA órajelhez

BEGIN
  -- Instantiate the Unit Under Test (UUT)
   uut: VGA_sync PORT MAP (
          mclk => mclk,
          rst => rst,
          hs => hs,
          vs => vs,
          video_en => video_en
        );
```

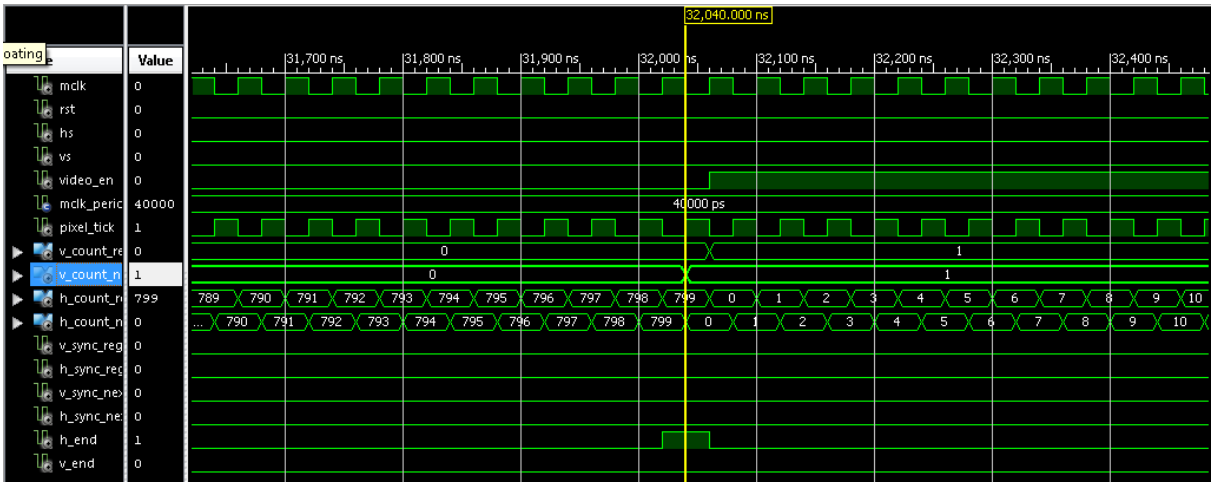
```
   -- Clock process definitions
   mclk_process :process
   begin
      mclk <= '0';
      wait for mclk_period/2;
      mclk <= '1';
      wait for mclk_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      rst <= '1';
      wait for 100 ns;
      rst <= '0';
      wait for mclk_period*10;
      -- insert stimulus here
      wait;
   end process;

END;
```

Ezután indítsuk el az ISim szimulátort, és adjuk hozzá a hullámforma (Wave) ablakhoz a következő felsorolásban lévő belső jeleket, melyek az `uut` néven példányosított `VGA_sync` entitás belső jelei voltak (ez azért szükséges, hogy pontosan követhessük a belső jelek változásait is, ne csak az IO portlistás szereplő jeleket):

- `v_count_reg`
- `v_count_next`
- `h_count_reg`
- `h_count_next`
- `v_sync_reg`
- `h_sync_reg`
- `v_sync_next`
- `h_sync_next`
- `h_end`
- `v_end`
- `pixel_tick`

A jelek hozzáadása után újraindítjuk a szimulátort (ekkor eltünnek a jelalakok a vizsgált/hozzáadott jelek mellől), `Simulation` → `Restart`, majd adjunk meg futási időnek kb. 17 ms-ot! Itt fontos megjegyezni, hogy a korábbi *2.14. táblázatnak* megfelelően a szimu-

látor futasi idejét azért állítjuk be legalább T(v_total)-ra (vagy többszörösére), hogy lássuk is egy teljes képkocka megjelenítéséhez szükséges jelváltásokat (`v_end` után): azaz min. ~16.7 ms ideig kell tehát futtatni a szimulációt. Az *2.33. ábrán* a jelek szimulációs eredménye látható.



2.33. ábra: ISim szimuláció eredménye VGA_sync entitás futtatására

A fenti szimulációs eredmény egy kisebb intervallumát megvizsgálva, valamint a korábbi szinkronizációs időzítési táblázattal (*2.14. tábla*) összevetve a T(h_total) teljes horizontális időre kb. 32 us-ot kell, hogy kapjunk. Ez látható a mellékelt ábrán is.

További feladatként vizsgáljuk meg a fontosabb vezérlő jelek beállításait, változásait: `h_end, v_end, h_sync, v_sync, video_en, pixel_tick` a `v_count, h_count` aktuális változásai, végértékei mellett!)

Teljes VGA-vezérlő implementációja

A következő lépésben tervezzük meg a VGA vezérlőt (legyen a neve `VGA_disp_top.vhd`), mint legmagasabb hierarchia szinten lévő entitást, amely rendelkezik minden szükséges I/O port-al. Legyenek a `vgaRed(2:0), vgaGreen(2:0), vgaBlue(1:0)` a színkomponensenkénti kimenetek (amelyek std_logic_vector típusúak), `h_sync`, illetve `v_sync` szinkronizációs kimenetek, illetve rendelkezzen a külső bemeneti `rst` reset jellel, valamint külső 50 MHz-es bemeneti `mclk` órajellel.

Ebből a bejövő 50 MHz-es órajelből (FPGA 'B8'-as láb) állítsunk elő a felére leosztott 25 MHz-es órajelet egy bináris számláló segítségével, melyhez használjuk fel a korábban tárgyalt `clk_div` entitást (`q(N-1)` kimenetének bitszélessége csak `N = 1`!). Ezért a VGA vezérlőbe komponensként kell meghívni, majd példányosítani a `clk_div` nevű entitást, amelynek kimeneti `clk_25_mhz` órajel portját egy belső jelen (`clk_25_mhz_sig`) keresztül a fenti `VGA_sync` modul `mclk` órajel bemenetével kell összekötni.

A VGA_disp_top.vhd nevű VGA vezérlő implementációja VHDL nyelven a következő:

```
-- VGA vezérlő implementációja
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity VGA_disp_top is
      port(
      mclk, rst: in std_logic;
      h_sync, v_sync: out  std_logic;
      vgaRed : out std_logic_vector(2 downto 0);
      vgaGreen : out std_logic_vector(2 downto 0);
      vgaBlue : out std_logic_vector(1 downto 0)
      );
end VGA_disp_top;

architecture Behavioral of VGA_disp_top is

   signal VGA_out_reg: std_logic_vector(7 downto 0);
   signal video_en: std_logic;
   signal clk_25_mhz_sig : std_logic; --belső jel a clk összekapcsolására

   -- VGA_sync komponens deklarációja példényosításhoz
   COMPONENT VGA_sync
   PORT(
         mclk : IN  std_logic;
         rst : IN  std_logic;
         hs : OUT  std_logic;
         vs : OUT  std_logic;
         video_en : OUT  std_logic
        );
   END COMPONENT;

   COMPONENT clk_div
   PORT(
      clr : in STD_LOGIC;
      mclk : in STD_LOGIC;
      clk_25_mhz : out STD_LOGIC);
   END COMPONENT;
```

```
begin
   --példányosítása clk_div entitásnak uut1 néven
   uut1: entity work.clk_div(behav)
      port map(mclk=>mclk, clr=>rst,
               clk_25_mhz => clk_25_mhz_sig);
   --példányosítása VGA_sync entitásnak uut2 néven
   uut2: entity work.VGA_sync(Behavioral)
      port map(mclk=>clk_25_mhz_sig, rst=>rst, hs=>h_sync, vs=>v_sync,
               video_en => video_en);

   --vga kimeneti buffer implementációja a generált adat megjelenítéshez
   process (clk_25_mhz_sig, rst)
   begin
      if rst='1' then
         VGA_out_reg <= (others=>'0');
      elsif (clk_25_mhz_sig'event and clk_25_mhz_sig='1') then
         --VGA_out_reg <= sw;
         VGA_out_reg <= "11101001";  --legyen tetszőleges a kimeneti kép
      end if;
   end process;
   -- VGA_out_reg jeleinek szétosztása, mivel vgaRed(2:0), vgaGreen(2:0),
   --vgaBlue(1:0) std_vector_logic típusúak
   vgaRed <= VGA_out_reg(7 downto 5) when video_en='1'
   else (others => '0');
   vgaGreen <= VGA_out_reg(4 downto 2) when video_en='1'
   else (others => '0');
   vgaBlue <= VGA_out_reg(1 downto 0) when video_en='1'
   else (others => '0');

end Behavioral;
```

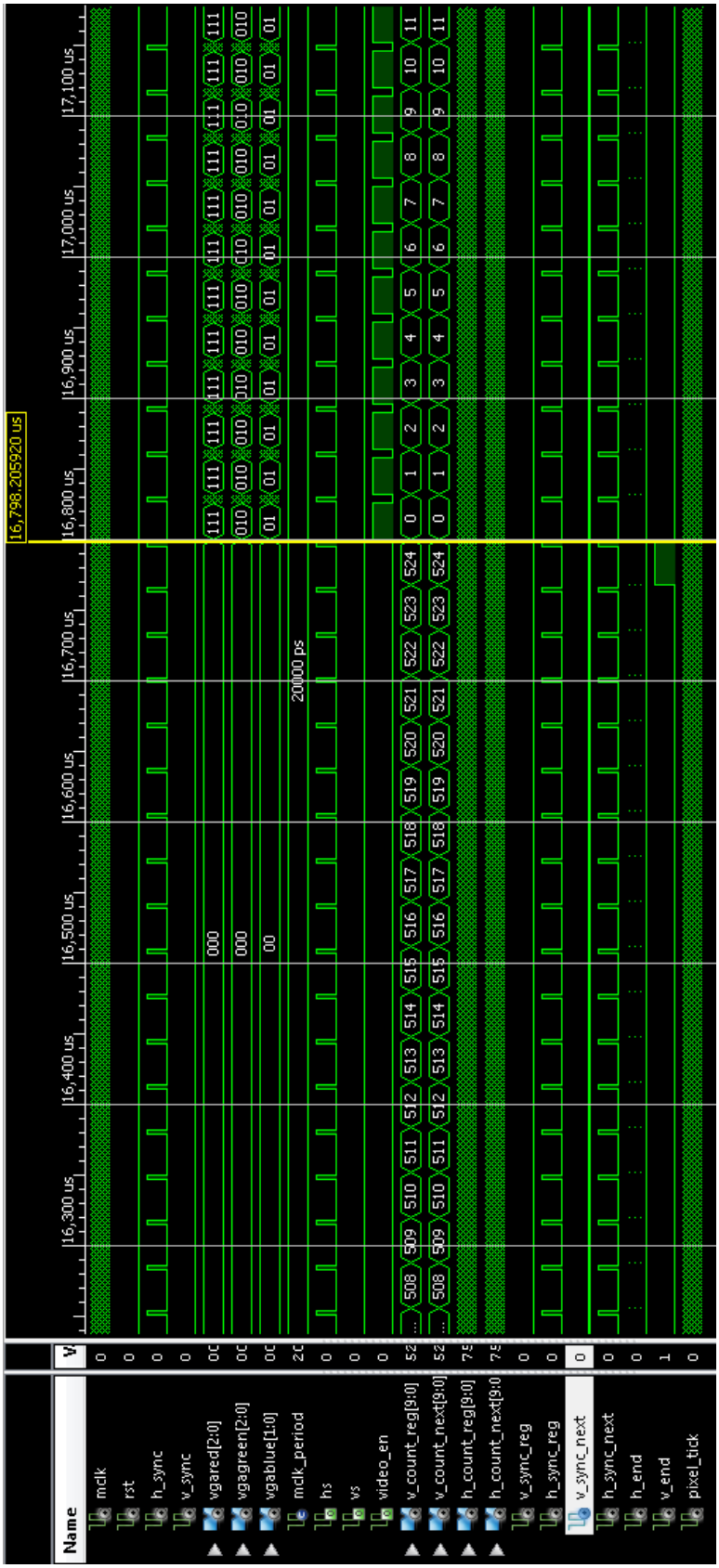
Teljes VGA vezérlő szimulációja

A `VGA_disp_top` nevű entitást, mint a VGA vezérlő legfelsőbb hierarchia szintjén lévő modult példányosítsuk a `VGA_disp_top_tb` nevű tesztágyban, és vizsgáljuk meg a működését. A gerjesztéseknél a `rst` jeleket kell egyedül megfelelően beállítanunk: a reset jel elengedésével (inaktív alacsony szinten) elindul a VGA vezérlő. Futtassuk el a szimulátort legalább **20 ms** szimulációs ideig várakozva (mivel tudjuk, hogy egy teljes v_sync periódus kb. ennyi időt vesz igénybe)! A teljes VGA vezérlő tesztágyának VHDL leírása a következő:

```
-- VGA vezérlő testbench
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY VGA_disp_top_tb IS
END VGA_disp_top_tb;

ARCHITECTURE behavior OF VGA_disp_top_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT VGA_disp_top
    PORT(
         mclk : IN  std_logic;
         rst : IN  std_logic;
         h_sync : OUT  std_logic;
         v_sync : OUT  std_logic;
         vgaRed : OUT  std_logic_vector(2 downto 0);
         vgaGreen : OUT  std_logic_vector(2 downto 0);
         vgaBlue : OUT  std_logic_vector(1 downto 0)
        );
    END COMPONENT;
   --Inputs
   signal mclk : std_logic := '0';
   signal rst : std_logic := '0';
 --Outputs
   signal h_sync : std_logic;
   signal v_sync : std_logic;
   signal vgaRed : std_logic_vector(2 downto 0);
   signal vgaGreen : std_logic_vector(2 downto 0);
   signal vgaBlue : std_logic_vector(1 downto 0);
   -- Clock period definitions
   constant mclk_period : time := 20 ns;  --50 MHz
```

```
BEGIN

    -- Instantiate the Unit Under Test (UUT)
   uut: VGA_disp_top PORT MAP (
          mclk => mclk,
          rst => rst,
          h_sync => h_sync,
          v_sync => v_sync,
          vgaRed => vgaRed,
          vgaGreen => vgaGreen,
          vgaBlue => vgaBlue
        );

   -- Clock process definitions
   mclk_process :process
   begin
      mclk <= '0';
      wait for mclk_period/2;
      mclk <= '1';
      wait for mclk_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      rst <= '1';
      wait for 100 ns;
      rst <= '0';
      wait for mclk_period*10;
      -- insert stimulus here
      wait;
   end process;
END;
```

A VGA vezérlő szimulációs eredménye a következő *2.34. ábrán* látható:



2.34. ábra: A VGA vezérlő szimulációs eredménye (részlet)

VGA vezérlő: Szintézis és impelementáció

Szintetizáljuk, majd generáljuk le a `VGA_disp_top.vhd` legfelsőbb hierarchia szinten lévő entitást. Ehhez a következő felhasználói kényszerfeltételeket is meg kell adni (.ucf):

```
NET "mclk"   LOC = "B8"; # Bank = 0, Pin name = IP_L13P_0/GCLK8, Type =
GCLK, Sch name = GCLK0

# VGA Connector
NET "vgaRed<0>"   LOC = "R9"; # Bank = 2, Pin name = IO/D5, Type = DUAL,
Sch name = RED0
NET "vgaRed<1>"   LOC = "T8"; # Bank = 2, Pin name = IO_L10N_2, Type =
I/O, Sch name = RED1
NET "vgaRed<2>"   LOC = "R8"; # Bank = 2, Pin name = IO_L10P_2, Type =
I/O, Sch name = RED2
NET "vgaGreen<0>" LOC = "N8"; # Bank = 2, Pin name = IO_L09N_2, Type =
I/O, Sch name = GRN0
NET "vgaGreen<1>" LOC = "P8"; # Bank = 2, Pin name = IO_L09P_2, Type =
I/O, Sch name = GRN1
NET "vgaGreen<2>" LOC = "P6"; # Bank = 2, Pin name = IO_L05N_2, Type =
I/O, Sch name = GRN2
NET "vgaBlue<0>"  LOC = "U5"; # Bank = 2, Pin name = IO/VREF_2, Type =
VREF, Sch name = BLU1
NET "vgaBlue<1>"  LOC = "U4"; # Bank = 2, Pin name = IO_L03P_2/DOUT/BUSY,
Type = DUAL, Sch name = BLU2

NET "h_sync" LOC = "T4"; # Bank = 2, Pin name = IO_L03N_2/MOSI/CSI_B, Type
= DUAL, Sch name = HSYNC
NET "v_sync" LOC = "U3"; # Bank = 2, Pin name = IO_L01P_2/CSO_B, Type =
DUAL, Sch name = VSYNC

# Buttons
NET "rst" LOC = "B18"; # Bank = 1, Pin name = IP, Type = INPUT, Sch name =
BTN0
```

Az elkészített és legenerált bitfájlt végül töltsük le a Digilent Nexys-2 kártyára és vizsgáljuk meg a működését CRT, vagy LCD monitoron.

További önálló feladat:

Ahhoz, hogy a CRT monitoron a kimeneti kép színe állítható legyen, vezessen be kapcsolókat (switch). Erre a célra a kártyán lévő 8 db programozható kapcsolót (`sw(7:0)`) használja fel. A 8 db programozható kapcsoló (8-bites) segítségével pont a 256 lehetséges színkombináció közül egyet lehet megjeleníteni a monitoron. Módosítsa a `VGA_disp_top` entitást úgy, hogy a kapcsolók aktuális értékét a `VGA_out_reg(7 downto 0)` jelhez rendelje. Szimulálja le a

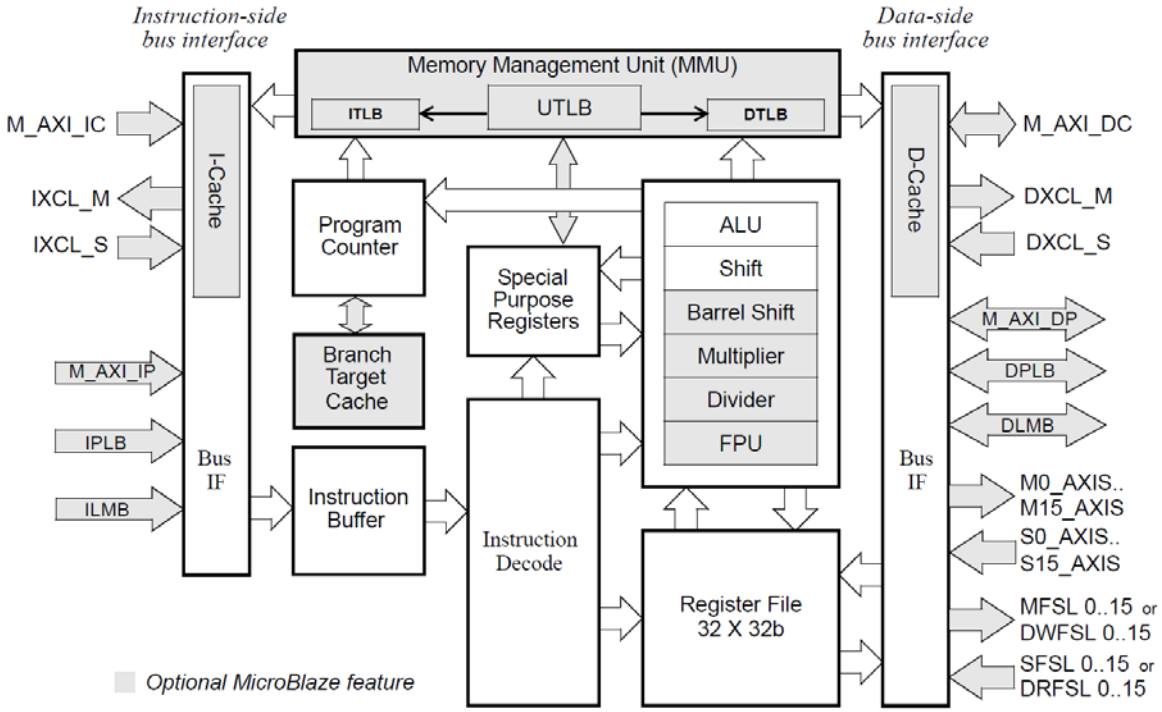
viselkedését, majd pedig szintetizálja le, és generáljon hozzá futtatható bitstream fájlt. Végül töltse fel az FPGA-ra, és vizsgálja meg a programozott kapcsolók állításával a kimeneti kép színének változását.

2.9. Beágyazott rendszer összeállítása Xilinx EDK használatával

Ebben a fejezetben a Xilinx EDK (Embedded Development Kit) – beágyazott rendszertervező környezet használatát ismertetjük röviden, főként a konfigurálható MicroBlaze szoft-processzor alapú rendszertervezésre fókuszálva.

2.9.1. MicroBlaze – beágyazott szoft-processzor mag

A Xilinx MicroBlaze™ egy 32-bites, RISC (csökkentett) utasítás-készletű, és Harvard architektúrát követő (elkülönített utasítás-adat tárolás) ún. 'szoft processzor mag', amely gazdag funkcionalitással és utasítás készlet architektúrával támogatja az FPGA alapú beágyazott alkalmazások fejlesztését és megvalósítását. Egyrészt, a MicroBlaze maghoz többfajta, gyártó (Xilinx) specifikus, vagy független gyártók által (3rd party) előre-definiált (ingyenes, vagy megvásárolható) IP – szellemi termékek integrálhatók. Ezek listáját a `Xilinx EDK → IP Catalogue` katalógus nézetében lehet megtekiteni [XILINX_EDK]. Másrészt arra is lehetőség van, hogy akár saját, egyedi (custom) szellemi termékeket implementáljunk, és integráljunk egy beágyazott alaprendszerhez. Ez temészetesen hosszabb, időigényesebbb folyamat, amely nagy tapasztalatot kíván, de a legoptimálisabb megoldás lehet egy konfigurálható saját beágyazott rendszer megvalósításához.



2.35. ábra: Xilinx MicroBlaze™ szoft-processzor mag belső felépítése [MICROBLAZE]:

Egy konfigurálható MicroBlaze mag a következő blokkszintű belső felépítéssel rendelkezik, (lásd *2.35. ábra*).

A MicroBlaze processzor legfontosabb paraméterei a következők [*MICROBLAZE*]:

- Konfigurálható 32-bites processzor mag
  - Jelenlegi legújabb verziója v.8.10 (Xilinx EDK 13.1 rendszerben)
  - Pipeline feldolgozást támogató architektúra (3-, vagy 5-lépcsős)
    - Konfigurálható 3-lépcsős (erőforrásra optimalizált) vagy 5-lépcsős pipeline (teljesítményre optimalizált változat)
  - Konfigurálható, opcionális utasítás-, és adat cache memória (`I-Cache, D-Cache`)
- Közvetlen leképezésű (1-utas csoport asszociatív cache)
  - 32 db, egyenként 32-bites általános célú regiszterrel rendelkezik (`Register File`)
  - Skálázható 32-bites, 64-bites, vagy 128-bites adatbusz
  - 32-bites címbusza van
  - 32-bites utasítás busszal rendelkezik (az utasítás kódolást tekinve maximálisan a 3-című utasításokat, valamint két címzési módot támogat: regiszteres, illetve az Imm-azonnali címzési módokat).
    - Aritmetikai
    - Logikai
    - Load/Store (Memória-regiszter transzfer műveletekhez – RISC sajátossága)
    - Programszervező utasítások (pl. Branch = if feltételes végrehajtás)
    - Speciális utasítások
  - Opcionális Memória Menedzselő és Memória Védelmi Egységgel (`MMU – Memory Management Unit`) rendelkezhet.
    - Akkor szükséges, ha a MicroBlaze magon egy Linux OS (beágyazott operációs rendszert akarunk használni – jelenleg az EDK-ban a Linux 2.6 verziót támogatott).
  - Opcionális processzor magba ágyazható lebegő-pontos atirmetikai egység (`FPU` – Floating Point Unit)
    - Szabványos IEEE 754 formátum szerinti működés
  - `Barrel Shifter`: tetszőlegesen léptethető regiszter (opcionális)
  - Hardveres szorzó (opcionális): ha be van kapcsolva 3 db MULT18x18-as dedikált blokkot használ el a Spartan3E FPGA-n. Természetesen ez is platform specifikus műveletvégző egység).
    - 32×32 –bites operanduszok szorzása →64-bites eredménnyel
  - Hardveres osztó áramkör (opcionális): szorzó áramköröket, logikai cellákat és esetlegesen dedikált BRAM cellákat is lefoglalhat).
  - `Fast Simplex Link (FSL)` FIFO csatornák: nagyon gyors, arbitráció nélküli, pont-pont összeköttetések (pl. akár két beágyazott MicroBlaze processzor és/vagy hardver gyorsító feldolgozó egységek közötti kommunikációra).
  - Hardveresen támogatott hibakeresés (széleskörű debug funkciókkal – `MDM – MicroBlaze Debug Module, Xilinx ChipScope`) és beágyazható, de más gyártó által szolgáltatott logikai analizátor modulok (pl. Agilent Trace Module).
  - Alapvetően `Big-Endian`, azaz bit-fordított tárolási formátumot követ (de a jelenlegi 13.x rendszertől kezdve már konfigurálható az endianitás a következő két byte/bit-sorrend szerint:

- `Little-endian:` a legújabb ARM magos `AXI-buszos` rendszerek támogatására.
- `Big-endian: PLB (Processor Local Bus)` buszra csatlakoztatható I/O periféria eszközök támogatására.

– Multi-processzoros képesség: a beágyazható MicroBlaze processzor magok `Mailbox IP`-n keresztül üzenet átadással kommunikálhatnak, valamint `Mutex (Mutual Exclusion)` IP-n biztosítja a processzorok közötti szinkronizációt.

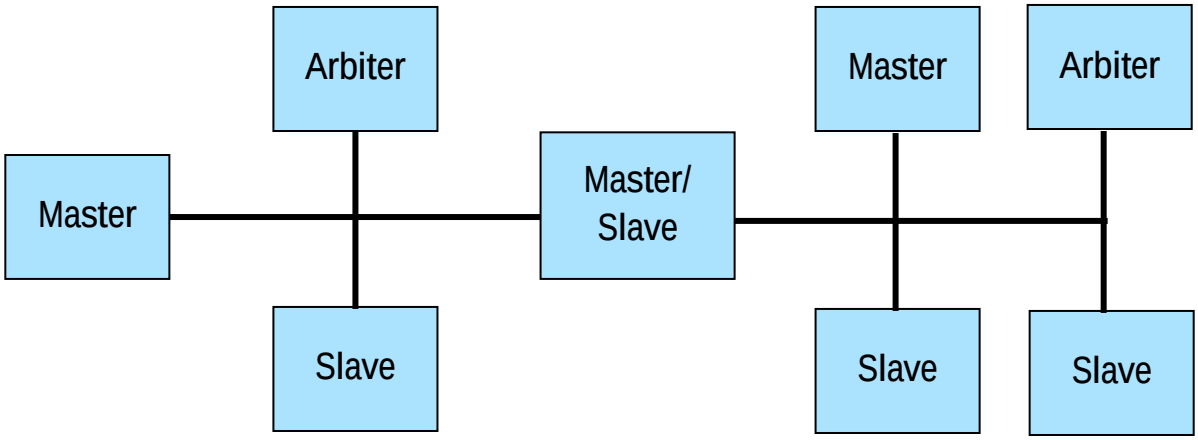
– Az új `AMBA (Advanced Microcontroller Bus Architecture)` busz architektúrának köszönhetően az ARM magos processzorokat már közvetlenül össze lehet kötni a MicroBlaze AXI rendszerével (EDK 13.x) [XILINX_EDK].

A MicroBlaze magról további bővebb információkat a Xilinx gyártó oldalán olvashatunk [MICROBLAZE].

2.9.2 MicroBlaze maghoz kapcsolódó buszrendszerek

A beágyazható MicroBlaze processzor mag [MICROBLAZE]:a különböző I/O perifériákkal (IP magok) Xilinx specifikus buszokon keresztül tud kommunikálni. A buszokat általánosan többvonalas, egy-, vagy kétirányú összeköttetéseknek tekintjük, amelyeken keresztül a szükséges információk az írási-, és olvasási tranzakcióknak megfelelően továbbítódnak. A következő információk továbbítására van lehetőség a buszvonalakon:

- Cím,
- Adat, és
- Vezérlési információkat különböztethetünk meg



2.36. ábra: Beágyazott rendszer IO perifériáinak osztályozása és kapcsolata [XILINX_EDK]

A mikroprocesszorhoz kapcsolható, konfigurálható I/O perifériákat a következő fontosabb paraméterek szerint osztályozhatjuk (*2.36. ábra*):

- `Arbiter` (arbitrációs egység) – Arbitrációs jelleggel működő,
- `Master` (mester, vagy 'másnéven' a kezdeményező egy buszon),
- `Slave` (szolga, vagy 'fogadó' fél egy buszon), illetve akár

- Együttes `Master-Slave` funkciójú (például különböző, vagy azonos buszvonalakat összekötö hidak – Bridge).

MicroBlaze belső buszrendszerei:

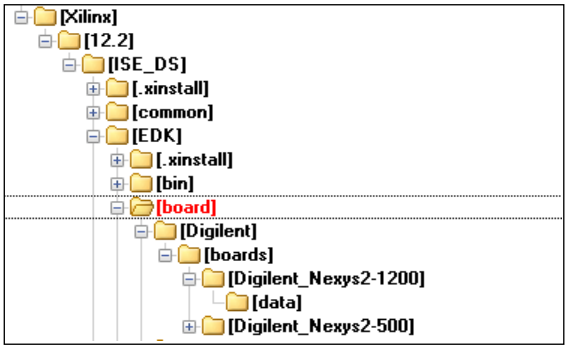
- Advanced Extensible Interface (AXI): ARM magos processzorok AMBA protokollja felé nyújtott támogatás
- **Processor Local Bus (PLB): MicroBlaze processzor (vagy akár periféri) busz rendszere, amelyre számos IO periféria köthető. Konfigurálható 32-, 64-bites adapt /ill. címbusszal, vagy esetenként 128-bites adat busszal.**
- On-chip Peripheral Bus (OPB): régebbi, ma már nem támogatott (visszafele kompatibilitás céljából hagyták meg) 32-bites buszrendszer. IBM szabványra épül, funkcionalitása azonos a PLB busszal.
- **Local Memory Bus (LMB): MicroBlaze helyi memória busza, amelyen keresztül (de memória vezérlőkön keresztül) az adat, illetve utasítás részek elérhetőek – Harvard architektúra. Dedikált BRAM memóriát használ.**
- Fast Simplex Links (FSL): gyors, és közvetlen pont-pont összeköttetés processzor, illetve feldolgozó processzor (accelerator), valamint periféria között. Lényegében egy 32-bit széles, és 16-mélységű programozható FIFO-t definiál.
- **Xilinx Cache Link (XCL): MicroBlaze 32-bites Cache vonala, külön adat-, és utasítás memória konfigurálható (opcionálisan) – Harvard architektúra. A MicroBlaze magot a Cache vezérlőkkel köti össze.**

Megjegyzés: vastagított betűvel azokat a részeket emeltem ki, amelyek a jelenlegi beágyazott rendszer használatának bevezetését szolgáló feladatban fel lettek használva. Ezekről a busz interfészekről részletesen a Xilinx oldalán a [*MICROBLAZE*], illetve [*XILINX_EDK*] leírásokban további részleteket olvashatunk.

2.9.3. Beágyazott alaprendszer összeállítása Xilinx XPS-ben

Mielőtt elindítanánk a `Xilinx XPS Platform Studio`-t, győződjünk meg róla, hogy a Digilent gyártó által a Nexys-2 kártyához biztosított ún. `Xilinx Board Definition (.xbd) fájl` (`Digilent_Nexys2-1200/500_v2_2_0.xbd`) a megfelelő helyen van-e [*NEXYS2*].

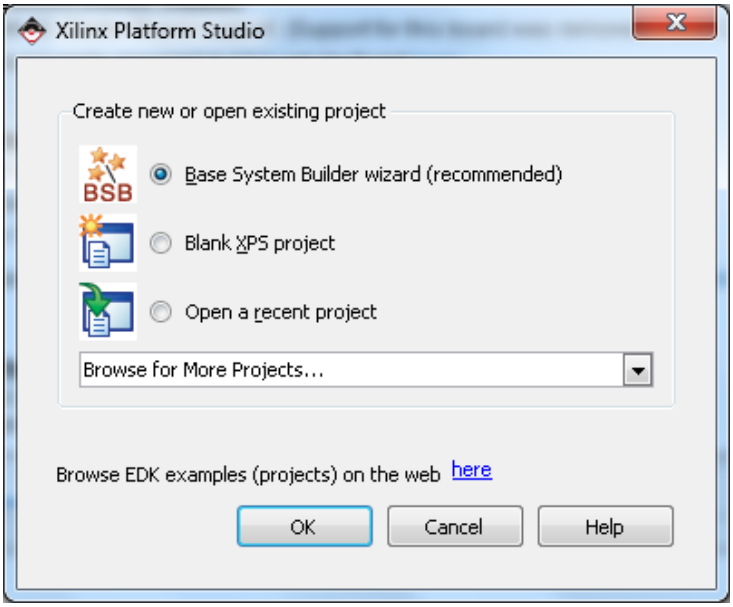
Ha nincs beálltva – például egy új Xilinx rendszer telepítése után, akkor ehhez először a gyártó oldaláról le kell tölteni a támogató csomagot (`BSP – Board Support Package`-t *`Digilent_Nexys_EDK_Board_Support_Files_V_2_1.zip`* [*DIGILENT*]), amelyet egy ideiglenes könyvtárba kell kicsomagolni. Majd pedig ezt a csomagot a \Digilent\ alkönyvtár szintjéről a telepített <Drive>\Xilinx\12.2\ISE_DS\EDK\board\ almappába kell másolni (a következő *2.37. ábra* szerinti könyvtár hierarchia szerint – ábra csak a szemléltetés végett van).



2.37. ábra: Xilinx EDK könyvtárszerkezete az .xbd fájl(ok) beállításához

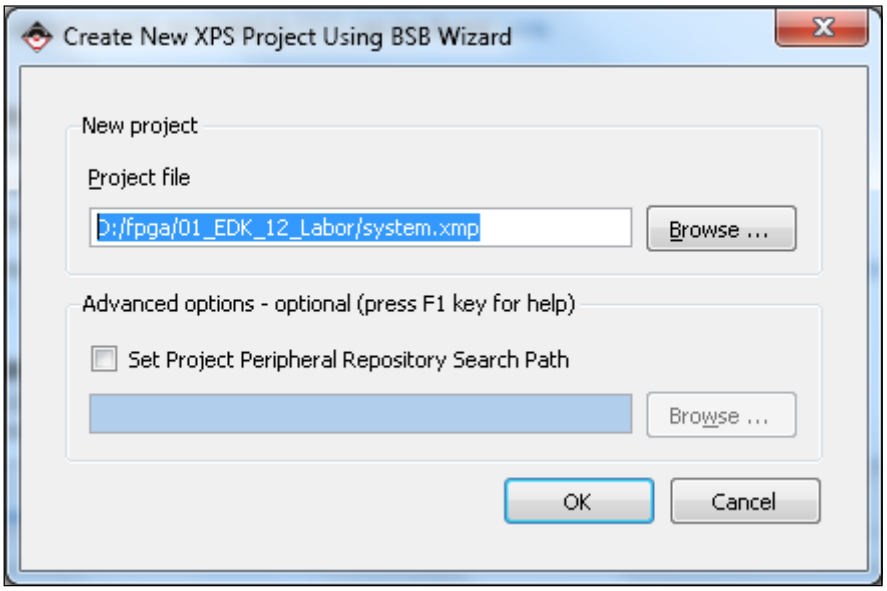
Megjegyzés: A kártya definíciós állomány (`.xbd`) beállítása már a tervezés kezdeti fázisában azért lehet nagyon fontos lépés, mivel esetünkben szerencsés módon a gyártó (Digilent Inc) – de akár más gyártók is – a rendelkezésünkre bocsátanak előre definiált beállításokat és paramétereket tartalmazó konfigurációs állományokat, amikor egy új beágyazott alaprendszert kívánunk összeállítani. Amennyiben a gyártó nem ad ilyen jellegű támogatást, vagy egyedi (saját tervezésű) FPGA-s kártyát kívánunk használni, akkor saját magunknak kell egy ilyen `.xbd` (board definition) fájlt létrehozni a `Xilinx XPS/EDK` segítségével [*XILINX_EDK*]. Ez azonban például a külső memóriák FPGA lábhozzárendelését, illetve pontos paraméterezését tekintve igen komplex feladat is lehet, még gyakorlott FPGA-alapú beágyazott rendszert tervező mérnököknek is. Természetesen az `.xbd` beállítását csak egyetlen egyszer, az adott kártyával történő első tervezéskor kell elvégezni.

1.) Indítsuk el ezután a `Xilinx -> Platform Studio (XSP)` integrált beágyazott rendszer tervező programját. Majd a File -> New Project -> Base System Builder Wizard opcióját válasszuk ki, a beágyazott alaprendszer összeállításához. Ez egy varázsló, amely végigvezet grafikus ablakokon keresztül a tervezési lépéseken.



2.38. ábra: Xilinx Platform Studio-alaprendszer összeállítása

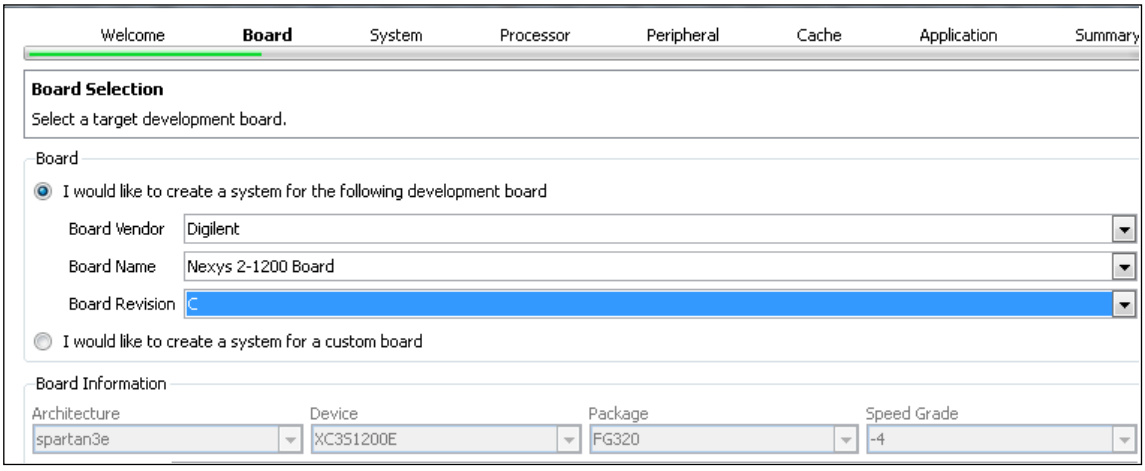
2.) OK-ra kattintva írjuk be az új projekt file nevének a következőt: `system.xmp`. Természetesen ez a név tetszőleges is lehet. Az `.xmp` kiterjesztés a Xilinx Project fájl formátumát jelenti. Válasszunk ki (Browse…) és hozzunk létre egy könyvtárat a `<Drive>\alkönyvtár\system.xmp` néven. A **lényeges megkötések** a file nevére, és könyvtár nevére vonatkozóan ugyanazok, mint amiket a Xilinx ISE környezetben megismerhettünk (lásd korábbi *2.4.2. alfejezet*).



2.39. ábra: – Xilinx Platform Studio –projekt elérési útjának megadása

3.) Ezután az üdvözlő „`Welcome` ablakban” válasszuk ki az `New Design` opciót.

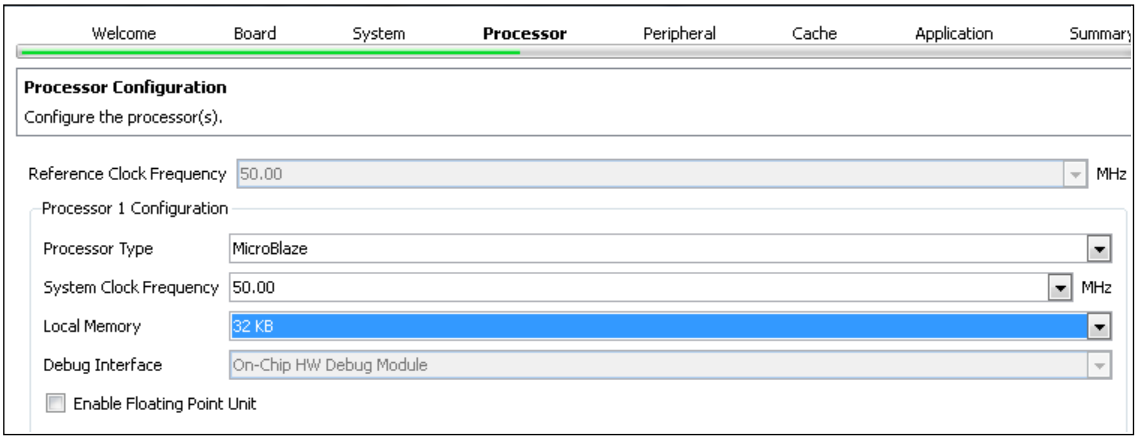
4.) A következő „`Board`” ablakban” válasszuk ki a korábban ismertetett `Digilent -> Nexys-2` (1200K, vagy 500K) fejlesztőkártyákat – rendelkezésre álló platformtól függően. Alul a `Board Information` részen ekkor megjelenik a pontos FPGA típusa: Xilinx Spartan 3E- 1200 / vagy -500, FG320-as tokozással, illetve -4 speed grade-el.



2.40. ábra: – Xilinx Platform Studio –Digilent Nexys-2 1200 kártya kiválasztása (amit a telepített .xbd fájl támogat)

5.) A következő „`System ablakban`” az egy-processzoros rendszert válasszuk ki (`single-processzor system` opciót). Természetesen lehetőség van arra is, akár ebben a lépésben, vagy a későbbiekben, hogy további beágyazható processzorokat adjunk a rendszerhez (`multi-processor system`).

6.) A következő ablakban a beágyazni kívánt processzor magot, annak órajelét és paramétereit tudjuk beállítani. Mivel a feladat során használt Spartan-3E FPGA-n csak szoft-processzor magokat lehet kialakítani (logikai és dedikált erőforrásoktól függően maximálisan 8-at engedélyez elhelyezni!) ezért a processzor típusánál a MicroBlaze magot válasszuk ki, alap 50 MHz-es rendszerórajel mellett. A `MicroBlaze`-hez kapcsolódó lokális memória (`LMB`) méretét növeljük meg 8KB → 32KB-ra, ami azért szükséges, hogy a későbbi „lefordított” alkalmazói programjaink beleférjenek az FPGA-n lévő belső dedikált memóriákba (ne kelljen külső memóriát, vagy boot-loadert használni). Továbbá ne engedélyezzük a lebegőpontos szorzók (`FPU`) használatát, mivel jelen megvalósítás során nem lesz szükség rájuk, és így jelentős hardver erőforrásokat takaríthatunk meg (min. 3 szorzót + további logikai erőforrásokat).



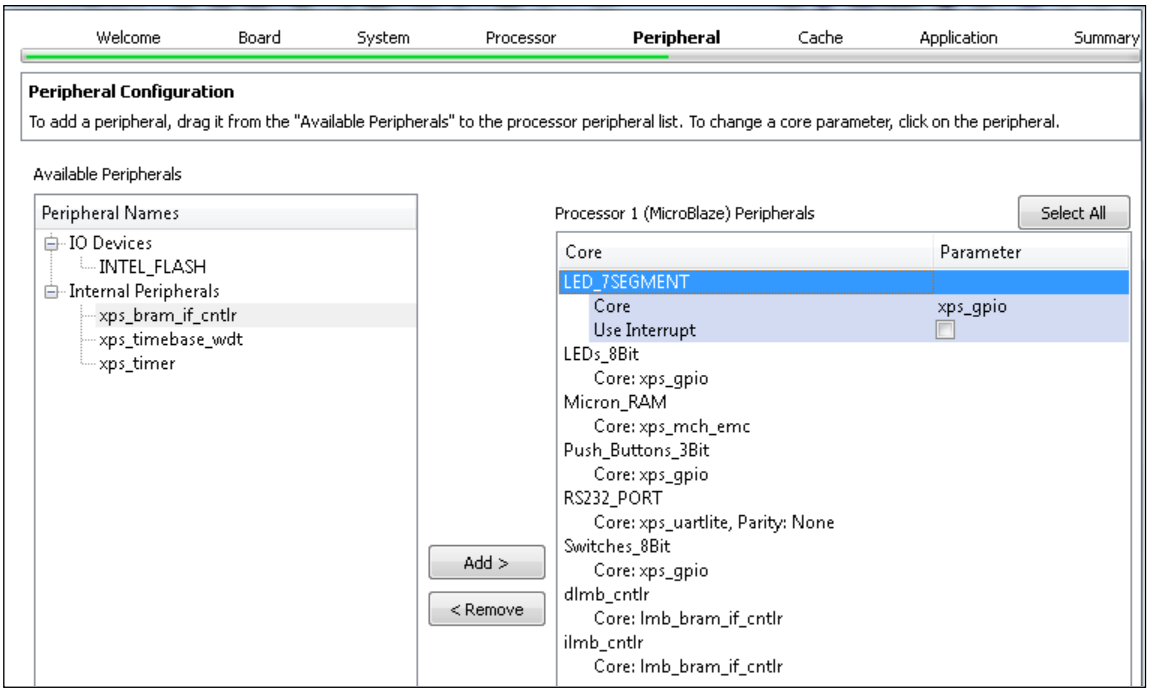
2.41. ábra: – Xilinx Platform Studio –MicroBlaze processzor konfigurálása

7.) A következő `'Peripheral'` ablakban lehet megadni a MicroBlaze beágyazott szoft-processzor maghoz integrálni kívánt IO periféria modulokat, amelyek a gyártó saját szellemi termékeit, azaz ún. `IP (Intellectual Property)` magok hozzáadását jelentik a meglévő tervünkhöz. Alapértelmezettként a következő IP modulok maradjanak bent (mivel ezeket a korábban ismertetett xbd fájl alapján olvassa be, vagy egy új rendszer összeállításakor adjuk hozzá):

- 7-segmenses LED kijelzők,
- 8-db LED,
- külső 16Mbyte-os Mircon SRAM memória,
- 8 db nyomógomb,
- egy RS232 soros port,
- 8 kapcsoló, illetve

- 1-1 adat és utasítás oldali blokk-ram vezérlő (Harvard architektúra!), amely a korábban definiált 32KB méretű lokális memória bank-ot tudja elérni (LMB).

Egyik hozzáadott periféria esetén se állítsuk be a megszakítás kezelését szolgáló belső jelet (mivel ebben a feladatban nem kívánjuk a perifériák megszakítás-kéréseit a MicroBlaze oldalon kezelni).



2.42. ábra: – Xilinx Platform Studio –IO perifériák (IP magok) hozzáadása az alaprendszerhez

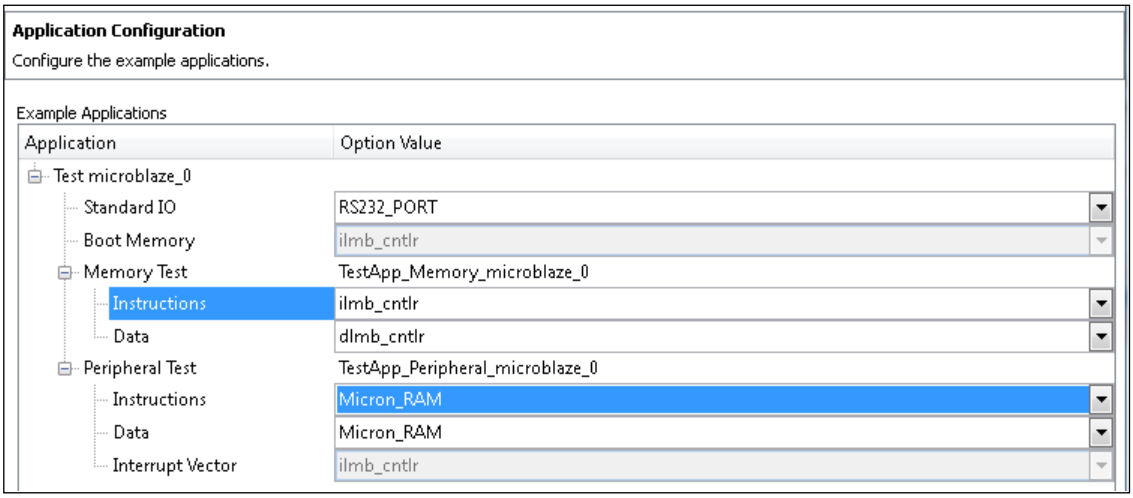
8.) A következő „`Cache` ablakban” a MicroBlaze processzor maghoz rendelhető beágyazott cache memóriákat lehet konfigurálni, és méretüket lehet beállítani. Mivel a külső Micron SRAM memóriát adtunk hozzá a rendszerhez, ezért mindenképpen érdemes bekapcsolni mind az adat-, mind pedig az utasítás oldali cache memóriákat (a memória teszteléshez), amelyek méretét egyenként állítsuk 2KB-os méretűre.

9.) Az alkalmazói programokat konfiguráló ablakban (`Application Configuration`) egyrészt adjuk meg, hogy a gyártó által összeállított mely szoftver alakalmazásokat kívábjuk a saját beágyazott rendszerünkhöz illeszteni,másrészt, hogy a konzolon (terminal) keresztül mely I/O porthoz csatlakozunk. Ez utóbbinak állítsuk be a szabványos `'RS232_PORT'`-ot (mivel később majd egy soros kábelen keresztül egy Terminal alkalmazásban vizsgálhatjuk a kéréseket, ill.küldéseket). Az előre definiált tesztalkalmazások legyenek a következők: `'TestApp_Memory'`, illetve `'TestApp_Peripheral'`. Természetesen a későbbiekben, saját egyedi alkalmazói programokat is hozzá tudunk adni a rendszerhez. Az alkalmazói programokat akár a Xilinx EDK rendszerében (bár ez az EDK 10.1 és korábbi rendszerekben megszokott módszer, és kevésbé rugalmas mód), akár az Eclipse alapú Xilinx SDK beágyazott szoftverfejlesztő rend-

szerében is létre lehet hozni (`Project menü → Export Hardware Design to SDK…`).

Még ebben az alkalmazó programokat konfiguráló ablakban az alkalmazások utasításainak és adatainak tárolási helyét is meg kell adni, amelyek a következők lehetnek:

- `ilmb`: MicroBlaze utasítás-oldali lokális memóriája (16 Blokk-Ramot foglal le, méretét korábban 32 KByte-ra növeltük!),
- `dlmb`: MicroBlaze adat-oldali lokáli memóriája (16 Blokk-Ram-ot foglal le, méretét korábban 32 KByte-ra növeltük), esetleg
- `Micron_SRAM`: 16 MByte méretű külső memória (ez nem Blokk-Ram használ).



2.43. ábra: – Xilinx Platform Studio –'Memory Test' és 'Peripheral Test' alkalmazások adat-, illetve utasításrészeinek tárolási helyei

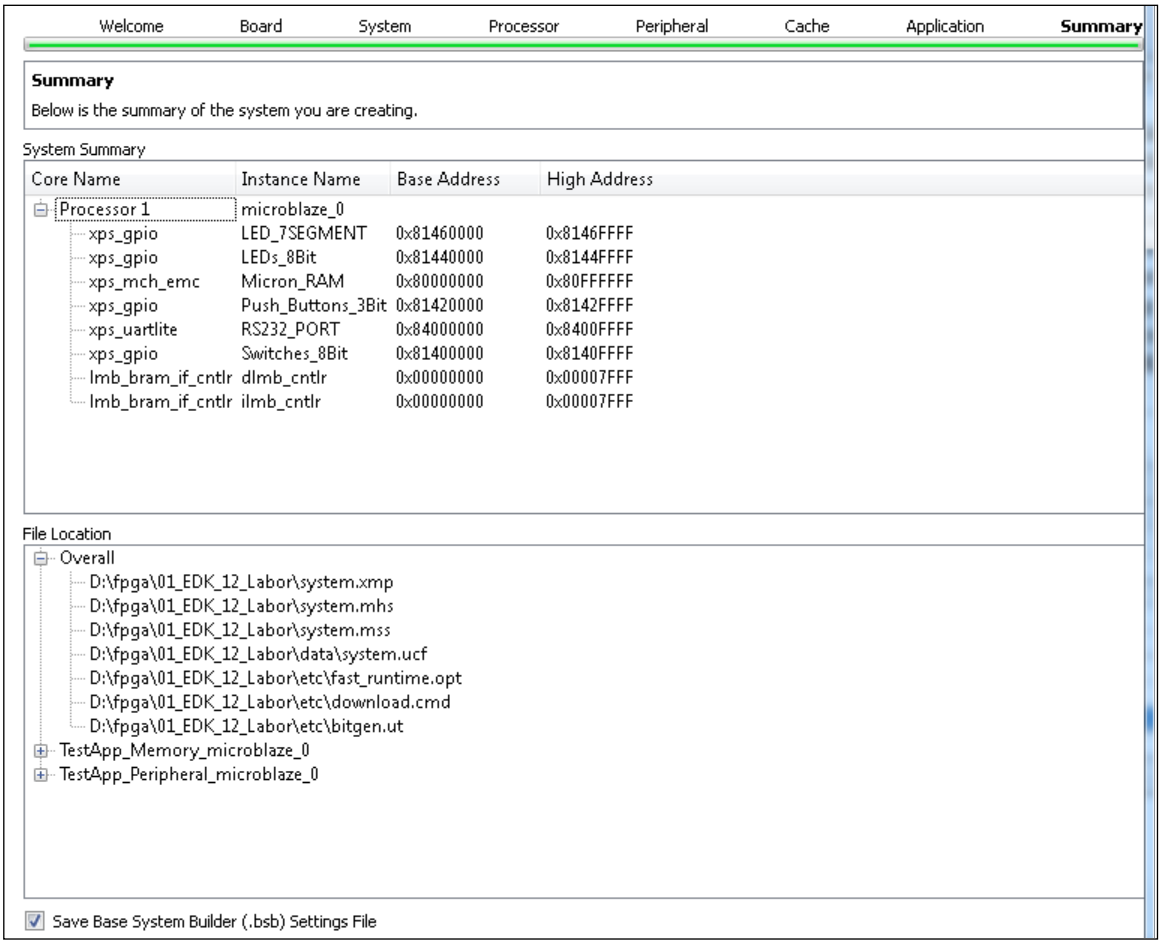
10.) Az beágyazott alaprendszer összeállításának utolsó lépéseként az összefoglaló ablakban (`Summary`) a MicroBlaze maghoz integrált IP perifériák gyári nevei (`Core Name` – ezek nem változtathatók meg, Xilinx specifikus nevek), azok példányosított nevei (`Instance name` – megváltoztathatók), a hozzájuk rendelt báziscímek (`Base address` - .xbd alapján kiosztott alapértelmezett), illetve felső címtartomány (`High address`) értékek kerülnek listázásra. Látható, hogy a MicroBlaze maghoz

- 4 db `GPIO` (általános célú IO – `xps_gpio`) periféria,
- 1 soros port kezelő periféria (`xps_uartlite`),
- 1 külső memória vezérlő (`xps_mch_emc`),
- valamint 1-1 adat-, utasítás-oldali Lokális Blokk-Ram memória-interfész vezérlő csatlakozik (`lmb_bram_if_ctrl`).

A „`File location`" ablak részben az EDK projekt állományai kerülnek listázásra, amelyek közül a következő fontos formátumokat szükséges kiemelni:

- **`Xmp`**: Xilinx EDK projekt fájl (ez fogja össze a teljes beágyazott projektet)

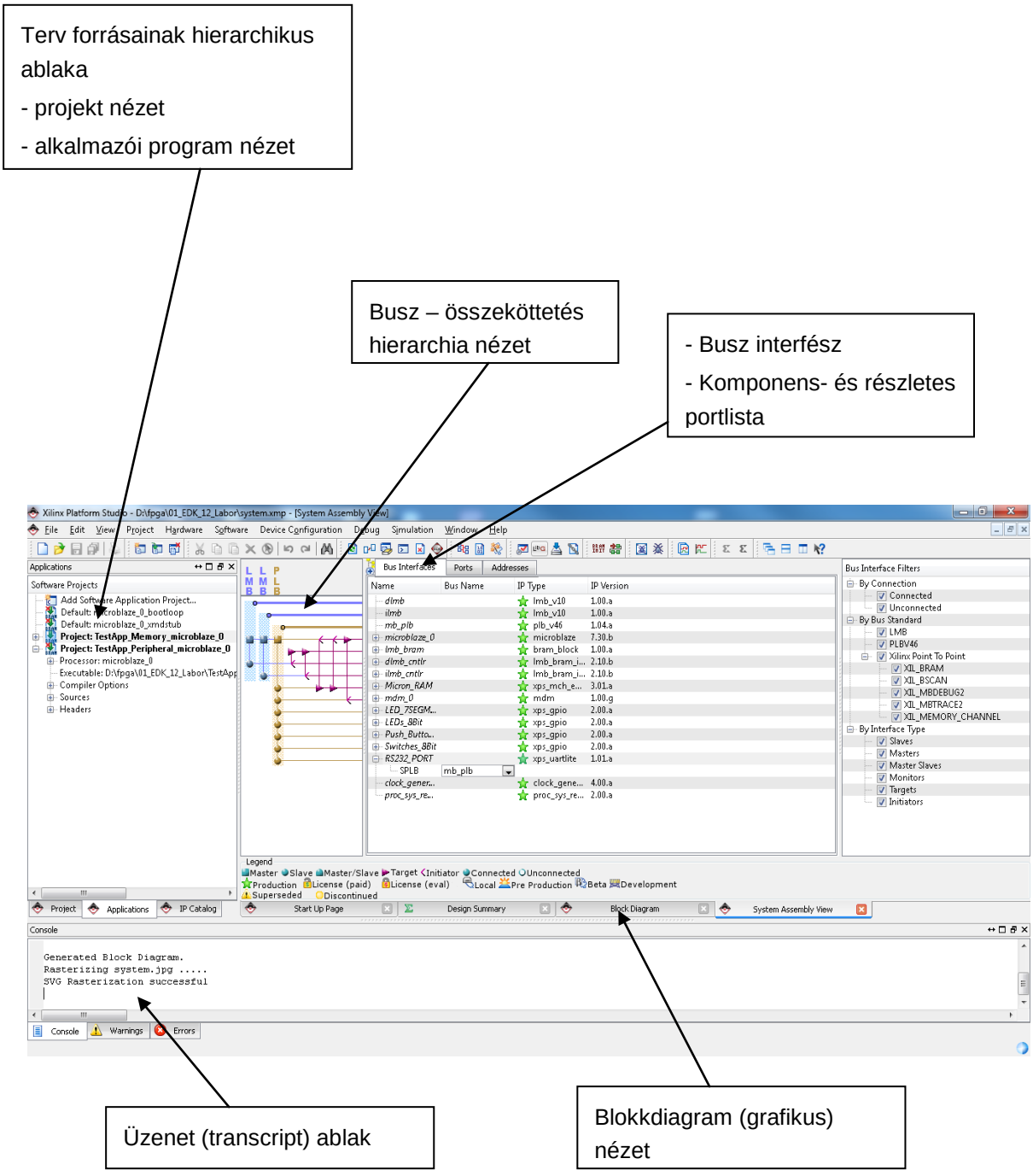
- **Mhs**: `MicroBlaze Hardvare Description`: a beágyazott rendszer szöveges hardver leírója
- **Mss**: `MicroBlaze Software Description`: a beágyazott rendszer szöveges szoftver leírója
- **Ucf**: `User Constraints File:` felhasználói / tervezői kényszerfeltételeket tartalmazó szöveges állomány
- **Bsb**: Ha a `BSB (Base System Builder)` mentését bejelöltük, akkor egy későbbi beágyazott rendszer tervezésekor, nem kell az alaprendszert ismételten összeállítani, elég csak megnyitni a korábban összeállított `.bsb` leíró fájlt.



2.44. ábra: – Xilinx Platform Studio – Summary ablak, az alaprendszer összeállításának utolsó fázisában

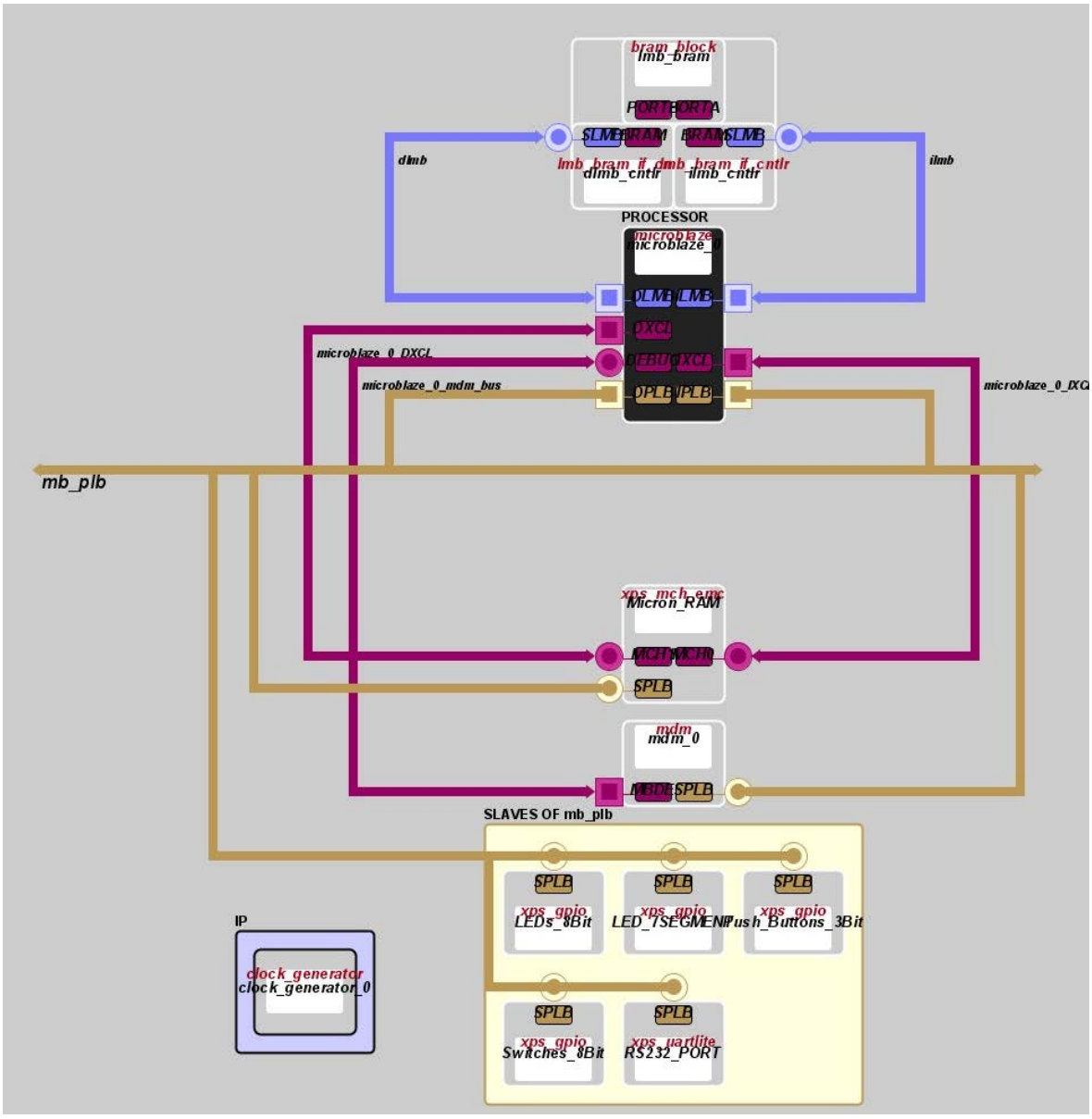
11.) Végül a MicroBlaze-alapú beágyazott hardver alaprendszer összeállítása után a Xilinx Platform Studio tényleges használata megkezdődhet, amely egyrészt alkalmas a beágyazott rendszer (hw) további beállításainak finomítására, I/O perifériák (IP magok) hozzáadására, illetve akár az alkalmazói programok (sw) részek létrehozására is.

A MicroBlaze-alapú beágyazott alaprendszer sikeres összeállítása után a következő tervezői nézet fogad bennünket a Xilinx Platform Studio-jának grafikus ablakában (*2.45. ábra*):



2.45. ábra: Az összeállított MicroBlaze alapú beágyazott tesztrendszer grafikus felülete és ablakai, a projekt megnyitásakor EDK/XPS-ben [XILINX_EDK]

A következő *2.46. ábra* az összeállított alaprendszer, illetve az integrált I/O perifériákat (mint IP magokat) és azok busz-szintű összeköttetéseit szemlélteti (ezt blokkdiagramot az XPS-ben lehet generáltatni).



2.46. ábra: a XPS-ben összeállított beágyazott rendszer grafikus, IP blokk-szintű felépítése

MHS: MicroBlaze Hardver leíró állomány

A következő forráskód a hardver tesztrendszer összeállítása során keletkezett MicroBlaze hardver leíró fájl belső struktúráját és nyelvi konstrukcióit szemlélteti [*XILINX_EDK*]:

```
# ##############################################################################
# Created by Base System Builder Wizard for Xilinx EDK 12.2 Build EDK_MS2.63c
# Thu Apr 08 03:02:29 2010
# Target Board:  Digilent Nexys 2-1200 Board Rev C
# Family:    spartan3e
# Device:    XC3S1200E
# Package:   FG320
# Speed Grade:  -4
# Processor number: 1
# Processor 1: microblaze_0
# System clock frequency: 50.0
# Debug Interface: On-Chip HW Debug Module
# ##############################################################################
 PARAMETER VERSION = 2.1.0


 PORT fpga_0_LEDs_8Bit_GPIO_IO_O_pin = fpga_0_LEDs_8Bit_GPIO_IO_O_pin, DIR = O,
VEC = [0:7]
 PORT fpga_0_LED_7SEGMENT_GPIO_IO_O_pin = fpga_0_LED_7SEGMENT_GPIO_IO_O_pin, DIR =
O, VEC = [0:11]
 PORT fpga_0_Push_Buttons_3Bit_GPIO_IO_I_pin =
fpga_0_Push_Buttons_3Bit_GPIO_IO_I_pin, DIR = I, VEC = [0:2]
 PORT fpga_0_Switches_8Bit_GPIO_IO_I_pin = fpga_0_Switches_8Bit_GPIO_IO_I_pin, DIR
= I, VEC = [0:7]
 PORT fpga_0_RS232_PORT_RX_pin = fpga_0_RS232_PORT_RX_pin, DIR = I
 PORT fpga_0_RS232_PORT_TX_pin = fpga_0_RS232_PORT_TX_pin, DIR = O
 PORT fpga_0_Micron_RAM_Mem_A_pin =
fpga_0_Micron_RAM_Mem_A_pin_vslice_9_31_concat, DIR = O, VEC = [9:31]
 PORT fpga_0_Micron_RAM_Mem_OEN_pin = fpga_0_Micron_RAM_Mem_OEN_pin, DIR = O
 PORT fpga_0_Micron_RAM_Mem_WEN_pin = fpga_0_Micron_RAM_Mem_WEN_pin, DIR = O
 PORT fpga_0_Micron_RAM_Mem_BEN_pin = fpga_0_Micron_RAM_Mem_BEN_pin, DIR = O, VEC
= [0:1]
 PORT fpga_0_Micron_RAM_Mem_DQ_pin = fpga_0_Micron_RAM_Mem_DQ_pin, DIR = IO, VEC =
[0:15]
 PORT fpga_0_clk_1_sys_clk_pin = dcm_clk_s, DIR = I, SIGIS = CLK, CLK_FREQ =
50000000
 PORT fpga_0_rst_1_sys_rst_pin = sys_rst_s, DIR = I, SIGIS = RST, RST_POLARITY = 1
```

```
BEGIN microblaze
 PARAMETER INSTANCE = microblaze_0
 PARAMETER C_AREA_OPTIMIZED = 1
 PARAMETER C_DEBUG_ENABLED = 1
 PARAMETER C_ICACHE_BASEADDR = 0x80000000
 PARAMETER C_ICACHE_HIGHADDR = 0x80ffffff
 PARAMETER C_CACHE_BYTE_SIZE = 2048
 PARAMETER C_ICACHE_ALWAYS_USED = 1
 PARAMETER C_DCACHE_BASEADDR = 0x80000000
 PARAMETER C_DCACHE_HIGHADDR = 0x80ffffff
 PARAMETER C_DCACHE_BYTE_SIZE = 2048
 PARAMETER C_DCACHE_ALWAYS_USED = 1
 PARAMETER HW_VER = 7.30.b
 PARAMETER C_USE_ICACHE = 1
 PARAMETER C_USE_DCACHE = 1
 BUS_INTERFACE DLMB = dlmb
 BUS_INTERFACE ILMB = ilmb
 BUS_INTERFACE DPLB = mb_plb
 BUS_INTERFACE IPLB = mb_plb
 BUS_INTERFACE DXCL = microblaze_0_DXCL
 BUS_INTERFACE IXCL = microblaze_0_IXCL
 BUS_INTERFACE DEBUG = microblaze_0_mdm_bus
 PORT MB_RESET = mb_reset
END

BEGIN plb_v46
 PARAMETER INSTANCE = mb_plb
 PARAMETER HW_VER = 1.04.a
 PORT PLB_Clk = clk_50_0000MHz
 PORT SYS_Rst = sys_bus_reset
END

BEGIN lmb_v10
 PARAMETER INSTANCE = ilmb
 PARAMETER HW_VER = 1.00.a
 PORT LMB_Clk = clk_50_0000MHz
 PORT SYS_Rst = sys_bus_reset
END
```

```
BEGIN lmb_v10
 PARAMETER INSTANCE = dlmb
 PARAMETER HW_VER = 1.00.a
 PORT LMB_Clk = clk_50_0000MHz
 PORT SYS_Rst = sys_bus_reset
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = dlmb_cntlr
 PARAMETER HW_VER = 2.10.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00007fff
 BUS_INTERFACE SLMB = dlmb
 BUS_INTERFACE BRAM_PORT = dlmb_port
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = ilmb_cntlr
 PARAMETER HW_VER = 2.10.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00007fff
 BUS_INTERFACE SLMB = ilmb
 BUS_INTERFACE BRAM_PORT = ilmb_port
END

BEGIN bram_block
 PARAMETER INSTANCE = lmb_bram
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = ilmb_port
 BUS_INTERFACE PORTB = dlmb_port
END

BEGIN xps_gpio
 PARAMETER INSTANCE = LEDs_8Bit
 PARAMETER C_ALL_INPUTS = 0
 PARAMETER C_GPIO_WIDTH = 8
 PARAMETER C_INTERRUPT_PRESENT = 0
 PARAMETER C_IS_DUAL = 0
 PARAMETER HW_VER = 2.00.a
```

```
 PARAMETER C_BASEADDR = 0x81440000
 PARAMETER C_HIGHADDR = 0x8144ffff
 BUS_INTERFACE SPLB = mb_plb
 PORT GPIO_IO_O = fpga_0_LEDs_8Bit_GPIO_IO_O_pin
END

BEGIN xps_gpio
 PARAMETER INSTANCE = LED_7SEGMENT
 PARAMETER C_ALL_INPUTS = 0
 PARAMETER C_GPIO_WIDTH = 12
 PARAMETER C_INTERRUPT_PRESENT = 0
 PARAMETER C_IS_DUAL = 0
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_BASEADDR = 0x81460000
 PARAMETER C_HIGHADDR = 0x8146ffff
 BUS_INTERFACE SPLB = mb_plb
 PORT GPIO_IO_O = fpga_0_LED_7SEGMENT_GPIO_IO_O_pin
END

BEGIN xps_gpio
 PARAMETER INSTANCE = Push_Buttons_3Bit
 PARAMETER C_ALL_INPUTS = 1
 PARAMETER C_GPIO_WIDTH = 3
 PARAMETER C_INTERRUPT_PRESENT = 0
 PARAMETER C_IS_DUAL = 0
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_BASEADDR = 0x81420000
 PARAMETER C_HIGHADDR = 0x8142ffff
 BUS_INTERFACE SPLB = mb_plb
 PORT GPIO_IO_I = fpga_0_Push_Buttons_3Bit_GPIO_IO_I_pin
END

BEGIN xps_gpio
 PARAMETER INSTANCE = Switches_8Bit
 PARAMETER C_ALL_INPUTS = 1
 PARAMETER C_GPIO_WIDTH = 8
 PARAMETER C_INTERRUPT_PRESENT = 0
 PARAMETER C_IS_DUAL = 0
 PARAMETER HW_VER = 2.00.a
```

```
 PARAMETER C_BASEADDR = 0x81400000
 PARAMETER C_HIGHADDR = 0x8140ffff
 BUS_INTERFACE SPLB = mb_plb
 PORT GPIO_IO_I = fpga_0_Switches_8Bit_GPIO_IO_I_pin
END

BEGIN xps_uartlite
 PARAMETER INSTANCE = RS232_PORT
 PARAMETER C_BAUDRATE = 9600
 PARAMETER C_DATA_BITS = 8
 PARAMETER C_USE_PARITY = 0
 PARAMETER C_ODD_PARITY = 0
 PARAMETER HW_VER = 1.01.a
 PARAMETER C_BASEADDR = 0x84000000
 PARAMETER C_HIGHADDR = 0x8400ffff
 BUS_INTERFACE SPLB = mb_plb
 PORT RX = fpga_0_RS232_PORT_RX_pin
 PORT TX = fpga_0_RS232_PORT_TX_pin
END

BEGIN xps_mch_emc
 PARAMETER INSTANCE = Micron_RAM
 PARAMETER C_NUM_BANKS_MEM = 1
 PARAMETER C_NUM_CHANNELS = 2
 PARAMETER C_MEM0_WIDTH = 16
 PARAMETER C_MAX_MEM_WIDTH = 16
 PARAMETER C_INCLUDE_DATAWIDTH_MATCHING_0 = 1
 PARAMETER C_SYNCH_MEM_0 = 0
 PARAMETER C_TCEDV_PS_MEM_0 = 85000
 PARAMETER C_TAVDV_PS_MEM_0 = 85000
 PARAMETER C_THZCE_PS_MEM_0 = 8000
 PARAMETER C_THZOE_PS_MEM_0 = 8000
 PARAMETER C_TWC_PS_MEM_0 = 85000
 PARAMETER C_TWP_PS_MEM_0 = 55000
 PARAMETER C_TLZWE_PS_MEM_0 = 5000
 PARAMETER HW_VER = 3.01.a
 PARAMETER C_MEM0_BASEADDR = 0x80000000
 PARAMETER C_MEM0_HIGHADDR = 0x80ffffff
 BUS_INTERFACE SPLB = mb_plb
```

```
 BUS_INTERFACE MCH0 = microblaze_0_IXCL
 BUS_INTERFACE MCH1 = microblaze_0_DXCL
 PORT RdClk = clk_50_0000MHz
 PORT Mem_A = 0b000000000 & fpga_0_Micron_RAM_Mem_A_pin_vslice_9_31_concat
 PORT Mem_OEN = fpga_0_Micron_RAM_Mem_OEN_pin
 PORT Mem_WEN = fpga_0_Micron_RAM_Mem_WEN_pin
 PORT Mem_BEN = fpga_0_Micron_RAM_Mem_BEN_pin
 PORT Mem_DQ = fpga_0_Micron_RAM_Mem_DQ_pin
END

BEGIN clock_generator
 PARAMETER INSTANCE = clock_generator_0
 PARAMETER C_CLKIN_FREQ = 50000000
 PARAMETER C_CLKOUT0_FREQ = 50000000
 PARAMETER C_CLKOUT0_PHASE = 0
 PARAMETER C_CLKOUT0_GROUP = NONE
 PARAMETER C_CLKOUT0_BUF = TRUE
 PARAMETER C_EXT_RESET_HIGH = 1
 PARAMETER HW_VER = 4.00.a
 PORT CLKIN = dcm_clk_s
 PORT CLKOUT0 = clk_50_0000MHz
 PORT RST = sys_rst_s
 PORT LOCKED = Dcm_all_locked
END

BEGIN mdm
 PARAMETER INSTANCE = mdm_0
 PARAMETER C_MB_DBG_PORTS = 1
 PARAMETER C_USE_UART = 1
 PARAMETER C_UART_WIDTH = 8
 PARAMETER HW_VER = 1.00.g
 PARAMETER C_BASEADDR = 0x84400000
 PARAMETER C_HIGHADDR = 0x8440ffff
 BUS_INTERFACE SPLB = mb_plb
 BUS_INTERFACE MBDEBUG_0 = microblaze_0_mdm_bus
 PORT Debug_SYS_Rst = Debug_SYS_Rst
END
```

```
BEGIN proc_sys_reset
 PARAMETER INSTANCE = proc_sys_reset_0
 PARAMETER C_EXT_RESET_HIGH = 1
 PARAMETER HW_VER = 2.00.a
 PORT Slowest_sync_clk = clk_50_0000MHz
 PORT Ext_Reset_In = sys_rst_s
 PORT MB_Debug_Sys_Rst = Debug_SYS_Rst
 PORT Dcm_locked = Dcm_all_locked
 PORT MB_Reset = mb_reset
 PORT Bus_Struct_Reset = sys_bus_reset
 PORT Peripheral_Reset = sys_periph_reset
END
```

MHS leíró fájl főbb komponensei a következőek:

- A # jelenti a kommentet.
- `PARAMETER VERSION = 2.1.0`. Ez az MHS fájl paramétereinek, és nyelvi konstrukciójának aktuális verzió számát jelenti (ne változtassunk rajta!)
- `PORT`: ez egy külső (external) port, amely az .ucf (kényszerfeltétel) fájlban megegyező névvel kell, hogy szerepeljen. Definiálja egy I/O port irányultságát, bitszélességét (endianitását), illetve azt, hogy milyen jelről van szó (lehet akár CLK – órajel, vagy RST – reset/inicializáló jel).
- A `BEGIN-END` kulcsszók közötti részeken egy-egy IP mag (szellemi termék) leírása szerepel, amely lehet akár egy MicroBlaze mag, vagy egy I/O periféria is.
    - A `BEGIN` kulcsszó után a periféria gyári neve kell, hogy szerepeljen (ezen NE változtassunk, különben a fordító hibaüzenetet küld).
    - A `PARAMETER` kulcsszó után a példányosított név áll: ezt átnevezhetjük (hasonlóan a VHDL példányosításhoz).
    - A paraméterek között általában a `HW_VER` a beágyazott IP mag aktuális hardver változatát jelenit (IP katalógus adat).
    - `BUS_INTERFACE` kulcsszó azt defninálja, hogy az integrált IP mag a beágyazott rendszer melyik szabványos, a Xilinx által definiált belső buszrendszeréhez csatlakozik (például PLB, XCL, LMB)
    - A `PORT` kulcsszó után a beágyazott IP magok portjainak nevei szerepelnek, a következő hozzárendelés szerint: `gyári_portnév = felhasználó_által_definiált_port / vagy belső jelnév`:

Ezekről a paraméterekről részletes leírást a Xilinx EDK referencia segédlet (reference guide) nyújt [*XILINK_EDK*].

MSS: MicroBlaze Szoftver (Driver) leíró állomány

A következő forráskód a szoftver/driver (de nem alkalmazás!) tesztrendszer összeállítása során generált MicroBlaze driver leíró fájl belső struktúráját és nyelvi konstrukcióit szemlélteti. Az itt lévő alacsony-szintű driverek általában az összeállított beágyazott rendszerben lévő IP magok meghajtó gyári programjai (hacsak nem egyedi, a felhasználó által definiált IP termékről van szó.)

```
 PARAMETER VERSION = 2.2.0



BEGIN OS
 PARAMETER OS_NAME = standalone
 PARAMETER OS_VER = 3.00.a
 PARAMETER PROC_INSTANCE = microblaze_0
 PARAMETER STDIN = RS232_PORT
 PARAMETER STDOUT = RS232_PORT
END



BEGIN PROCESSOR
 PARAMETER DRIVER_NAME = cpu
 PARAMETER DRIVER_VER = 1.12.b
 PARAMETER HW_INSTANCE = microblaze_0
 PARAMETER COMPILER = mb-gcc
 PARAMETER ARCHIVER = mb-ar
END



BEGIN DRIVER
 PARAMETER DRIVER_NAME = bram
 PARAMETER DRIVER_VER = 2.00.a
 PARAMETER HW_INSTANCE = dlmb_cntlr
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = bram
 PARAMETER DRIVER_VER = 2.00.a
 PARAMETER HW_INSTANCE = ilmb_cntlr
END
```

```
BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = lmb_bram
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = gpio
 PARAMETER DRIVER_VER = 3.00.a
 PARAMETER HW_INSTANCE = LEDs_8Bit
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = gpio
 PARAMETER DRIVER_VER = 3.00.a
 PARAMETER HW_INSTANCE = LED_7SEGMENT
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = gpio
 PARAMETER DRIVER_VER = 3.00.a
 PARAMETER HW_INSTANCE = Push_Buttons_3Bit
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = gpio
 PARAMETER DRIVER_VER = 3.00.a
 PARAMETER HW_INSTANCE = Switches_8Bit
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = uartlite
 PARAMETER DRIVER_VER = 2.00.a
 PARAMETER HW_INSTANCE = RS232_PORT
END
```

```
BEGIN DRIVER
 PARAMETER DRIVER_NAME = emc
 PARAMETER DRIVER_VER = 3.00.a
 PARAMETER HW_INSTANCE = Micron_RAM
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = clock_generator_0
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = uartlite
 PARAMETER DRIVER_VER = 2.00.a
 PARAMETER HW_INSTANCE = mdm_0
END

BEGIN DRIVER
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = proc_sys_reset_0
END
```

MSS driver/szoftver leíró fájl főbb komponensei a következőek:

- A # jelenti a kommentet.
- `PARAMETER VERSION = 2.2.0.` Ez az MSS fájl paramétereinek, és nyelvi konstrukciójának aktuális verzió számát jelenti (ne változtassunk rajta!)
- A `BEGIN-END` kulcsszók közötti részeken egy-egy IP mag (szellemi termék) meghajtó programja (drivere) szerepel. Ez rendelhető akár MicroBlaze maghoz, akár IO perifériához is.
    - A `BEGIN DRIVER` kulcsszó után a periféria neve kell, hogy szerepeljen. A driver neve lehet akár gyári (pl: `GPIO, UARTLITE` stb.) vagy lehet akár `GENERIC`, amely azt jelzi, hogy az adott perifériához nem tartozik driver (de akkor is meg kell adni).
        - `BEGIN OS`: MicroBlaze mag operációs rendszere (ha nincs beállítva, akkor ún. standalone módban fut)
        - `STD_IN:` szabványos input (ezt érdemes az UART I/O periféria modul RX vonalára állítani, azért, hogy egy terminálon keresztül tudjunk adatok küldeni, vagy akár fogadni – lásd *2.43. ábra*)

        - `STD_OUT:` szabványos output (ezt szintén érdemes az UART IO periféria modul TX vonalára állítani)
        - `BEGIN PROCESSOR`: MicroBlaze-mag fordító programjának beállítása: pl. szabványos GNU compiler (gcc)
    - A `DRIVER_VER:` jelzi a meghajtó program aktuálisan használt verzió számát (`<főverzió.alverzió.revizió>` formátumban pl. `<1.01.b>`)
    - A `HW_INSTANCE` kulcsszó definiálja a driver egy példányosított nevét (ez szintén tetszőleges lehet MHS-hez hasonlóan).

Hardver implementálása és generálása

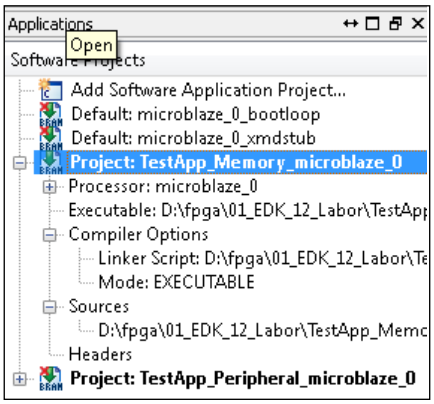
A beágyazottt alaprendszer elkészítése a rendszer hardver részeinek összeállítását jelenti. Ezt a Xilinx ISE fordítási (szintetizálási) lépéseihez hasonlóan kell implementálni, és a végén a bitstream fájl-t legenerálni. Megjegyezném, hogy a Xilinx XPS rendszer „háttér" utasításokként a Xilinx XST eszköz parancsait hívja meg (akár script alapú, parancs-soros üzemmódban is dolgozhatunk).

1. A Hardver rendszer implementálásához az `XPS PlatGen` nevezetű program modulját kell elindítani, a **`Hardware menü→ Generate Bitstream`** opció, vagy az Ikonsor  eszköztárának kiválasztásával. Ez a beágyazott rendszer komplexitásától függően néhány percet (több processzor esetén akár órát! is igénybe vehet).
    a. Ezekben a lépésekben történik a korábban megismert EDIF logikai netlista fájl előállítása: **`Genarate Netlist`**  ,
    b. a terv (hardver) szintetizálása, implementálása (fizikai leképezés / MAP, elhelyezés / PLACE – illetve összekötés /ROUTE fázisok végrehajtása). **`Genarate Bitstream`**: 
2. Az előző 1./b lépésben az implementált terv egy `system.bit` nevű bitstream állomány formájában generálódik, a projektünk `<drive>\ProjektNév\implementation\` alkönyvtára alatt.

Fontos! Mivel ez a fájl még csak az elkészült beágyazott tesztrendszer hardver részeit tartalmazza (hw), mindenféle szoftver alkalmazás (sw) lefordítása nélkül, ezért ha ezt a `system.bit` állományt letöltjük az FPGA-s kártyára, csak a áramköröket (CLB, BRAM stb.) konfigurálása történik meg, valójában érdemi változás a tesztrendszer működésének ellenőrzéséhez nem figyelhető meg!

Alkalmazói szoftver implementálása és fordítása

A Xilinx XPS – `Application folyamat` ablakában a szoftver projektek listáját láthatjuk (*2.47. ábra*)



2.47. ábra: Szoftver alkalmazások

Minden szoftver alkalmazáshoz hozzá kell rendelni egy beágyazott processzort, amin a programjainkat futtatni szeretnénk (esetünkben, ha mást nem adtunk meg névnek az .mhs fájl leíróban, akkor ez a `microblaze_0` nevű processzor). Az `'Executable'` opció a legenerált és futtatható szofver kód nevét definiálja egy szabványos `ELF (Executable and Linkable Format)` fájl formájában (<Projektnév.elf>). Ez töltődik le, mint egy tárgykódú szoftver (sw) állomány az FPGA-ra. A `'Compiler'` opciók között lehet beállítani az ún. `Linker Script`-et, amellyel az egyes szegmensekhez (pl: `instruction`-kód, `data`-adat, `stack`-verem, `heap`-kupac) rendelhető akár a belső LMB BRAM memória (esetünkben), akár egy külső (Nexys-2 kártyán Micron_SRAM) memória. Ezeket – hacsak nem akarunk boot-loadert írni az alkalmazásunkkhoz, amely a külső memóriából tölti be az adatokat – hagyjuk a belső `lmb_Xbram` memóriákban definiálva (X: jelöli az `d-adat`, illetve `i-utasítás` részeket).

A `'Sources'` menüpont alatt pedig a tényleges alkalmazói program forráskódjait találhatjuk, egy hierarchikus listába szervezve. Itt meg lehet adni akár szabványos forrás .c, .cpp, vagy fordítási könyvtáraknak .h, .hcc az állományait is. Jelenlegi példánkban csak egyetlen generált forrás állomány található: `<Projektnév.c>` néven, amely a program belépési pontját, azaz a `main()` függvényt tartalmazza.

A szoftver állományok fordítása, valamint a futtatható tárgykódú állomány előállítása (.elf) a következő lépésekből áll:

1. **`Library Generate`** : az IO perifériák (IP magokhoz) kapcsolódó driverek könyvtárainak (függvényeinek) előállítása
2. Forráskódok (.c, .cpp) illetve header-fájlok (.h, .hpp) **`fordítása (compiler)`**
3. Ha szükséges, akkor a **`Linker-script`** generálása : amely pontosan definiálja az egyes szegmensek, mely címtartományra (belső, vagy külső memória) kerüljenek. **Megjegyeznénk**, éppen ezért növeltük 32KByte méretűre a belső BRAM memóriát, hogy a lefordított alkalmazói programokból képzett futtatható (.elf) fájl beleférjen. Ha

erre nincs lehetőség, akkor külső memóriát (pl SRAM, DDR, vagy Flash memóriát) kell használni, és boot-loader-el kell betölteni a futtatandó alkalmazói programjainkat. Ez azonban túlmutat a jelenlegi jegyzet funkcionalitásán. További részletekért a Xilinx leírásait [XILINX_EDK] érdemes áttekinteni, vagy akár a Xilinx Platform Studio – HELP is használható a program használatának megértéséhez.

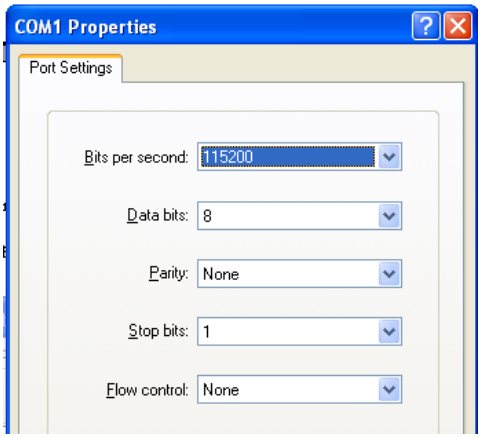
2.10. Beágyazott tesztalkalmazás (TestMemory) letöltése és tesztelése az FPGA kártyán

Miután minden szükséges hardver-szoftver rendszert összeállítottunk, implementáltunk, illetve lefordítottunk, akkor következhet a generált bitstream fájl FPGA-ra töltése és tesztelése. A korábban előállított futtatható tárgykódú .ELF alkalmazói állományt (TestMemory.elf), illetve a generált bitstream állományt (system.bit) kell „összefűzni" – ún. mergelni.

Ezt a **Device Configuration** menü→ **BRAM init/ Update Bitstream** opció, vagy az Ikonsor  tudjuk akár egy lépésben megtenni. Ezután előáll a <ProjektNév>\implementation alkönyvtár alatt egy „download.bit" állomány (utalva a letölthetőségre) amely már magában foglalja a futtatható alkalmazói program, illetve beágyazott alaprendszer összefordított részeit is.

A felhasználói / alkalmazói TestMemory program futtatásához a következő lépéseket kell megtenni:

4. Csatlakoztassuk és indítsuk el a Digilent Nexys-2 kátyát.
5. Az FPGA-s kártya, illetve a hoszt PC egy szabványos „**null-modem**" kábelen keresztül kell, hogy kapcsolódjon.
6. Ahhoz, hogy a soros porti adatokat meg tudjuk jeleníteni, illetve el tudjuk küldeni egy Terminál programra is szükség van. Itt lehet használni akár a Windows beépített Hypeterminal programját, vagy más külső programokat (pl. Telnet, Putty stb.)
   a. Mi az első, HyperTerminal beállítását és használatát mutatjuk be röviden.



   b. Szükséges paraméterek: `BaudRate`: 9600 (**Fontos**: ez a beállítás pontosan abból az .mhs paraméterből származik, amit a `C_BAUDRATE` paraméternek állítottunk be az `xps_uartlite` modul esetén, lásd korábbi MHS fájl!).
   c. `Data bit`: 8 / `Parity bit: None` (ne legyen paritás ellenőrzés) / `Stop bit: 1` / `Flow Control: None`.
7. Ezután az Digilent → Adept Suite program segítségével töltsük le az FPGA-ra a „`download.bit`" konfigurációs bitstream állományt.

8. Ha mindent jól állítottunk be, és generáltunk, akkor a következő információknak kell megjelennie soros vonalon keresztül a HyperTerminal ablakában:

```
-- Entering main() –
Starting MemoryTest for Micron_RAM:
  Running 32-bit test...PASSED!
  Running 16-bit test...PASSED!
  Running 8-bit test...PASSED!
-- Exiting main() --
```

(Hiba esetén: 'FAILED!' üzenetet kapunk).

Ezzel az FPGA alapú MicroBlaze beágyzott rendszerek bevezetéséül szolgáló szemléltető tesztrendszer összeállítása, valamint a hardver-szoftver részek együttes tesztelése befejeződött. ☺

További hasznos dokumentációk érhetőek el a Xilinx oldalán, illetve a Xilinx Platform Studio grafikus felületén keresztül az alábbi menüpontokban:

- Help → EDK Online Documentations: terminológia, szoftverek részeinek, komponenseinek, illetve EDK IP moduljainak részletes ismertetéses
- Help → EDK examples on the web: részletes leírások beágyazott rendszerek összeállításához
- Xilinx Egyetemi Program keretében sok, haszos letölthető dokumentációt találunk, nem csak EDK témakörben [*XILINX_UNI*]

Irodalomjegyzék a 2. fejezethez

[ADA] ADA programming language (honlap): http://www.adaic.org/

[ASHENDEN] Peter. J. Ashenden: *The Designer's Guide to VHDL*, 3. Kiadás (2008), Morgan Kaufmann Publisher

[BME] dr. Horváth Tamás, Harangozó Gábor – VHDL segédlet, honlap: http://www.fsz.bme.hu/~tom/vhdl/vhdl_s.pdf

[CATAPULTC] CatapultC Synthesis Tool (honlap - 2011) http://www.mentor.com/esl/catapult/overview

[CHU] Pong P. Chu -FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version, Wiley, 2008. Honlap: http://academic.csuohio.edu/chu_p/rtl/fpga_vhdl.html

[DIGILENT] DigilentInc Hivatalos honlap: http://digilentinc.com (2011)

[DISSZ] Vörösházi Zsolt PhD Disszertáció: Emulált-digitális CNN-UM architektúrák vizsgálata: Retina modell és Celluláris Hullámszámítógép implementációja FPGA-n. (Pannon Egyetem, ITDI 2010) honlap: http://konyvtar.vein.hu/doktori/2010/Voroshazi_Zsolt_dissertation.pdf

[GAJSKI] D. D. Gajski and R. H. Kuhn, "New VLSI Tools," *IEEE Computer*, **Vol. 16**, no. 12 (December 1983), pp. 11–14.

[HANDELC] Handel-C Synthesis Methodology (honlap - 2011) http://www.mentor.com/products/fpga/handel-c/

[HASKELL] Richard E. Haskell & Darrin M. Hanna – *„Introduction to Digital Design VHDL"*, Digilent Inc-LbeBooks, 2009. Honlap: http://digilentinc.com/Data/Textbooks/Intro_Digital_Design-Digilent-VHDL_Online.pdf

[IMPULSEC] Impulse CoDeveloper (honlap - 2011) http://www.impulseaccelerated.com/

[ISIM] Xilinx ISim szimulátor – hivatalos oldal: http://www.xilinx.com/tools/isim.htm

[KERESZTES] Hosszú Gábor – Keresztes Péter: VHDL alapú rendszertervezés (Budapest, 2006) honlap: https://wiki.sch.bme.hu/pub/Villanyszak/TervezesProgramozhatoEszkozokkel/vhdl.pdf

[LABVIEW] NI LabView FPGA Module (honlap - 2011) http://www.ni.com/fpga/
71. oldal:

[MATLAB] Matlab FPGA Design Tool (honlap - 2011) http://www.mathworks.com/fpga-design/

[MICROBLAZE] MicroBlaze Reference Guide (2011) – honlap: http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/mb_ref_guide.pdf

[MODELSIM] ModelSim PE Student Edition – HDL Simulation (holnap - 2011): http://model.com/content/modelsim-pe-student-edition-hdl-simulation

[MOORE] Gordon Moore: Cramming More Components Onto Integrated Circuits [Electronics Magazine, 1965]

[NEXYS2] Digilent Nexys-2 fejlesztő kártya: http://digilentinc.com/Products/Detail.cfm?NavPath=2,400,789&Prod=NEXYS2 (2011)

[PMOD] Digilent Peripheral Modules:
http://digilentinc.com/Products/Catalog.cfm?NavPath=2,401&Cat=9 (2011)

[SPARTAN3] DS312 Spartan-3 FPGA Family: Complete Data Sheet (hivatalos adatlap – 2011): http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf

[SYSTEMC] System-C Language Reference (honlap - IEEE Std 1666-2005)
http://www.systemc.org/home/

[SYSTEM_VERILOG] System Verilog-Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800-2009)
http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5354133

[VDEC] Digilent Video Decoder Board
http://digilentinc.com/Products/Catalog.cfm?NavPath=2,648&Cat=4 (2011)

[VERILOG] Hardware Description Language Based on the Verilog(R) Hardware Description Language (IEEE Std 1364-1995). New York, Institute of Electrical and Electronics Engineers, Inc., 1995.

[VESA] VESA 2011: Honlap: http://www.vesa.org (CVT: Coordinated Video Timing Formula adatlap)

[VHDL87] IEEE Standard VHDL *Language Reference Manual* (IEEE Std 1076-1987). New York, Institute of Electrical and Electronics Engineers, Inc., 1988.

[VHDL93] IEEE Standard VHDL *Language Reference Manual* (IEEE Std 1076-1993). New York, Institute of Electrical and Electronics Engineers, Inc., 1994.

[WEBPACK] Xilinx ISE WebPack – hivatalos honlap (legfrissebb verzió):
http://www.xilinx.com/tools/webpack.htm

[XILINX] Xilinx Hivatalos honlap: http://www.xilinx.com (2011)

[XILINX_DEVICES] Xilinx eszközök hivatalos oldala – Xilinx eszközök:
http://www.xilinx.com/products/silicon-devices/fpga/index.htm (2011)

[XILINX_EDK] Xilinx Embedded Development Kit - Platform Studio (honlap - 2011):
http://www.xilinx.com/tools/platform.htm

[XILINX_SCHEMATIC] Xilinx Schematic Editor (honlap - 2011)
http://www.xilinx.com/itp/xilinx10/isehelp/ise_c_schematic_overview.htm

[XILINX_STATE_CAD] Xilinx StateCAD (honlap - 2011)
http://www.xilinx.com/itp/xilinx10/isehelp/ise_n_statecad_navpage.htm

[XILINX_UNI] Xilinx University Program (honlap – 2011):
http://www.xilinx.com/university/index.htm