Budapesti Műszaki és Gazdaságtudományi Egyetem

Elektronikus Eszközök Tanszéke

Tudományos diákköri dolgozat

Algoritmusok közvetlen hardverbe történő fordítása VHDL alapú modellezés felhasználásával

Készítette: Minkó Bálint (P0Z080)

Konzulens: Dr. Hosszú Gábor egyetemi docens

2011.

Tartalom

1. **Bevezetés**......3

2. **Irodalmi áttekintés**......4

2.1. Hardver-szoftver codesign......4

2.2. Hardver-compilerek......7

2.3. A 8051-es mikrokontroller......15

2.3.1. A 8051-es memóriaszervezése......16

2.3.2. A 8051-es VHDL modellje......18

3. **Kidolgozott eljárás**......19

3.1. Felhasznált eszközök......19

3.2. A VHDL modell programozása gépi kódban......20

3.3. Az RS232-es VHDL modul......23

3.4. Az RS-232-es modul testbench-e......31

3.5. Az RS-232-es modul illesztése a mikrokontroller VHDL modelljéhez......43

4. **Eredmények és értékelésük**......59

5. **Összefoglalás**......63

6. **Irodalomjegyzék**......64

1. Bevezetés

Dolgozatom elején bemutatom a szakirodalomban megtalálható hardver compilerek működését és lehetséges megvalósításuk különböző módjait. Mindezek után egy olyan, VHDL alapú, hardver és szoftver együttes tervezésére alkalmas saját fejlesztésű modellt ismertetek, amely egy (akár FPGA-ban implementálható) mikrokontrolleren és egy hozzá illesztett C fordítón alapszik. Ezért kutatásaim jelentős résza a megfelelő mikrokontroller VHDL modelljének fejlesztésére irányult. Az elvégzett vizsgálatok célja olyan mikroprocesszor modell fejlesztése volt, amely lehetővé teszi egy új fejlesztésű, egyszerű hardver-compiler rendszer részeként történő alkalmazását. Ennek a hardver-compiler rendszernek a bemenete a megvalósítandó digitális rendszer algoritmusának leírása, amely többféle szinten történhet, a gépi kódtól kezdve a magas szintű programnyelven történő leírásáig.

Magas szintű nyelvként a C programozási nyelvet választottam, majd további kutatásokat folytattam a szakirodalomban egy, az Intel 8051-es mikrokontrollerrel kompatibilis C fordító kiválasztásával és működésének áttekintésével. A kidolgozott összetett hardver és szoftver modell részeként kifejlesztésre került egy olyan speciális VHDL modul, amely a C fordító kimenetét egy RS232-es interfészen keresztül közvetlenül elérhetővé teszi a mikrokontroller modell számára. Ezen újonnan kifejlesztett RS232 modul működését szimulációs úton ellenőriztem, majd hozzáillesztettem a mikrokontroller VHDL modelljéhez. Dolgozatom végén az elvégzett fejlesztések helyes működésének szimulációkkal történő igazolását mutatom be.

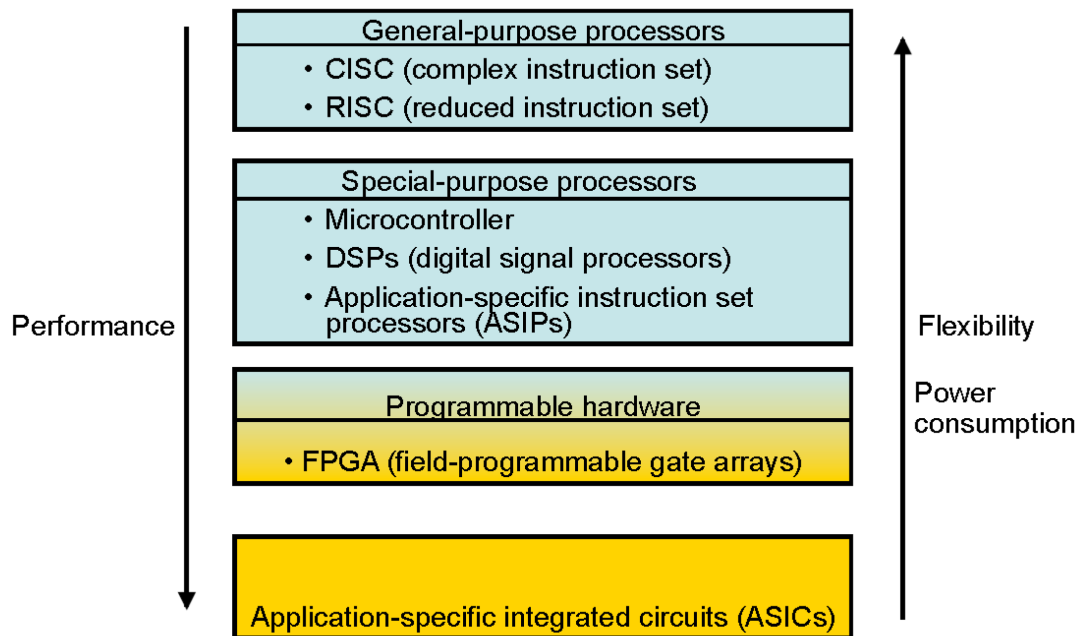
2. Irodalmi áttekintés

A mai rohamosan fejlődő világunkban egyre növekvő jelentőségük van a legkülönfélébb típusú, gyártmányú és architektúrájú mikroprocesszoroknak. Ma már szinte elképzelhetetlen a mindennapi életünk elektronikus eszközeink nélkül, amelyekben manapság legalább egy, de gyakran több mikroprocesszor működik. Éppen ezért különösen fontos, hogy megismerjük ezen eszközöket, tulajdonságaikat, működésüket, határaikat, továbbá ezen eszközök tervezésének folyamatát annak érdekében, hogy minél egyszerűbben, gyorsabban, illetve a sokszínű alkalmazási igények szerint lehessen alakítani, programozni őket. Ezen mikroelektronikai berendezések elterjedése a jövőben minden bizonnyal gyorsuló ütemben fog nőni. Ez a napjainkban megszokott iramú fejlődés azonban nehezen fenntartható, ha nem egészítjük ki eddigi hardver- és szoftverfejlesztési elméleteinket az éppen e fejlődés mentén létrehozott újabb technológiák és kutatásokon alapuló lehetőségekkel. Ennek értelmében először rövid áttekintést adunk a hardver és szoftver együttes tervezéséről, majd a hardver-compilerekkel foglalkozunk. Végül a kutatás eredményeként kifejlesztett egyszerű hardver-compiler rendszerben alkalmazásra kerülő Intel 8051-es mikrokontrollerről adunk egy áttekintést.

2.1. Hardver-szoftver codesign

Az elektronikus eszközök fejlesztésénél sokáig viszonylag kevés hangsúlyt fektettek a hardver és a szoftver közötti együttműködés optimalizációjára, vagyis sokszor kész hardvereken futtattak kész, ugyan a hardverrel kompatibilis, ám az adott hardverhez messze nem optimális szoftvereket. Ennek okai szerteágazóak (és sok esetben kényszer által szültek), ám kijelenthetjük, hogy az elektronikai ipar mai követelményeinek (amelyek között pl. a time-to-market legalább olyan fontos, mint magának az eszköznek a teljesítménye) az így tervezett eszközök nagy valószínűséggel nem, vagy csak nagyon nehezen felelnének meg.

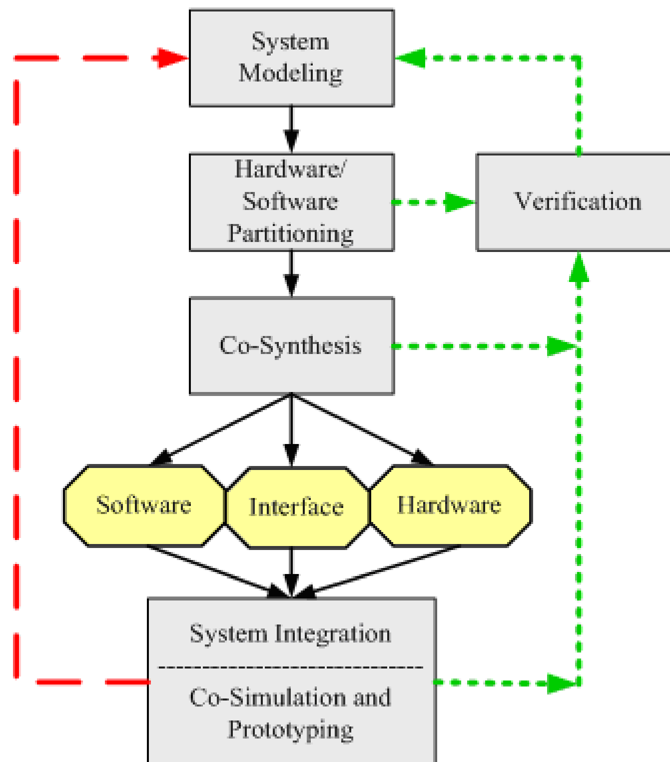
A fentiekben leírt helyzettel szemben a hardver és szoftver együttes tervezésével olyan összetett rendszereket tudunk fejleszteni, amelyek kifejezetten az adott problémára „koncentrálnak”. Ezáltal ugyan kevésbé rugalmasak, de gyorsabbak (nagyobb teljesítményűek) és alacsonyabb fogyasztásúak. (1. ábra)



1. *ábra:* A hardver-szoftver együttes tervezés implementálási alternatívái [5]

Ennek az együttes tervezési folyamatnak (2. ábra) az elején áll az egész rendszer modellezése, annak minden megkötésével együtt. A fejlesztés e szakaszában a terv leírja mind a hardver, mind a szoftver modellt. A modellezés fázisában alapvetően három, egymástól különböző megközelítést igénylő kiindulási pontot különböztethetünk meg. Az első esetben rendelkezünk egy meglévő szoftver implementációval, és ehhez kell egy megfelelő hardvert tervezni. Második esetben pont fordított a felállás, vagyis egy létező hardverre kell megfelelő szoftvert fejleszteni, míg a harmadik eset az, amikor a specifikáción kívül semmi sem adott. Itt élvezheti a tervező a modell megválasztásánál a legnagyobb szabadságot. A 2. ábrán látható folyamatábra alapján következő lépésként a hardver és a szoftver particionálását kell meghatározni szem előtt tartva a piacra kerülési időt, méretet, árat és fogyasztást. Következő lépésként a hardver, szoftver és az interfészek együttes szintézisét kell végrehajtani, a három implementáció közötti lehető legtöbb együttműködést biztosítva. Ezen a ponton a hardvert szintézis

általában VHDL vagy Verilog hardverleíró nyelvre épül, a szoftver szintézis pedig valamilyen magas szintű programnyelvre.



2. *ábra:* A hardver-szoftver együttes tervezés összefoglaló folyamatábrája [6]

A folyamat végén az előző pontban együtt szintetizált HW, SW és IF együtt szimuláljuk valós időben. Mint az az ábrán is látható minden egyes lépés után a verifikáljuk az addigi eredményeket és szükség szerint módosítjuk a rendszermodellünket.

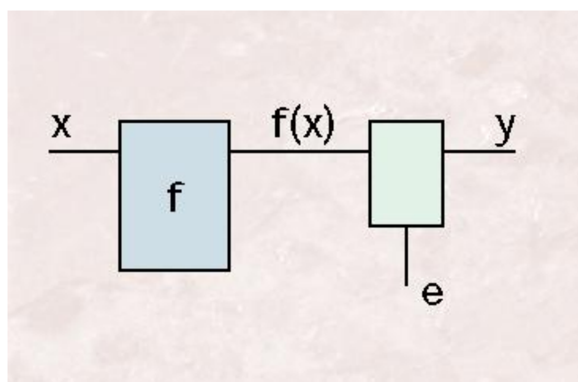
2.2. Hardver-compilerek

A hardver-compilerek [7, 8, 9] (más néven silicon-compilerek) mögött egy olyan, már igen régen felmerült elképzelés rejlik, hogy egy megfelelő fordító egy magas szintű programozási nyelven megírt programot automatikusan egy digitális áramkörre „fordítja le", vagyis egy digitális áramkört hoz létre a magas szintű leírásból. Ez az elképzelés sokáig csak elméleti volt és nem, vagy csak nagyon nehezen tűnt a valóságban kivitelezhetőnek. A félvezetőiparban az elmúlt évtizedekben végbement hatalmas fejlődésnek és az ennek következtében elérhető kvázi kifogyhatatlan mennyiségű tranzisztoroknak hála, az elképzelés már nem csupán elméleti. Ezt a folyamatot tovább erősítette a különböző, könnyen és gyorsan (újra)programozható áramköröket realizáló eszközök elterjedése.

A fent vázolt elképzelés megvalósításának egyik fő problémája, hogy a hardver- és szoftverfejlesztési folyamatokat hagyományosan egymástól különböző műveletnek tekintettük, a szoftvert és hardvert pedig két, egymástól jól elkülönülő entitásnak. A legkézenfekvőbb különbséget a működés szempontjából hardver és szoftver között abban láthatjuk, hogy míg egy programot utasítások rendezett halmazaként értelmezünk, amelyeket egymás után, szekvenciálisan hajtunk végre, addig a hardver olyan áramkörök halmaza, amelyek egymással párhuzamosan működnek. A két elv közötti különbségeket máshogyan megközelítve egy programnak a tervezése véget ér a fordítással, míg egy áramköri leírás fordítása csupán áramkör egy másik elvonatkoztatási szintjét állítja elő, a tervezőnek azt még fizikailag meg kell valósítania, ahol további problémákkal (pl. késleltetés) szembesül. A hardverleíró nyelvek megjelenésével azonban felismerhettük a kettejük között lévő közös jellemvonásokat is. Így már a hardvert is a programhoz hasonlóan szövegként tudtuk leírni, helyettesítve a hagyományos áramköri ábrákat. Miután egy HDL nyelv egy áramkört statikusan ír le, egy programnyelv viszont egy folyamatot, ezért azon célunk elérése érdekében, hogy egy áramkört egy programozási nyelv szövegéből származtassunk, automatikusan elő kell állítanunk egy olyan sorrendi szerkezetet, amely a program vezérlési struktúrájával megegyező. Így az összeadás, szorzás és más műveletek áramkörré fordításához szükséges szabályok felállítása mellett szükségünk van *if*, *while*, *repeat* és más vezérlési struktúrák implementálására. Továbbá definiálnunk kell az alábbi kapcsolatokat: egy

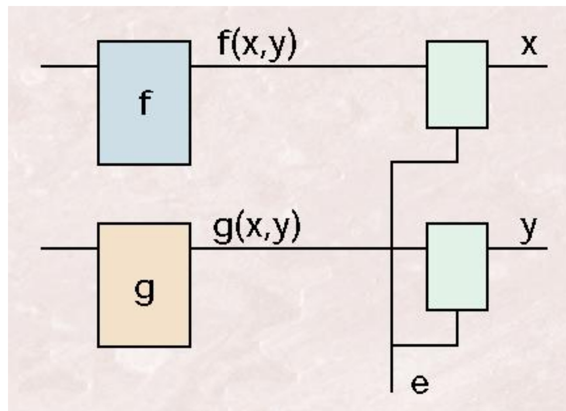
program változóit az áramkörben órajelvezérelt regisztereknek tekintjük, kifejezéseket pedig kapukból felépített kombinációs hálózatoknak. A következőkben láthatjuk, hogy ezek a struktúrák hogyan jelennek meg egy áramkörben.

Változók deklarálását általában a „*var x, y, z: típus*” jelöléssel végezhetjük. A hozzá tartozó áramkörnél a változó tartalmát egy regiszter tárolja, a regiszter kimenete az általa ábrázolt változó nevét jelöli, egy *enable* bemenet pedig meghatározza, hogy a regiszter mikor kap új értéket. Egy programban az értékadást a következőképpen tudjuk jelölni: „*y := f(x)*”, ahol *y* egy változó, *x* változók halmaza, *f* pedig egy x-től függő kifejezés. (3. ábra)



3. *ábra:* Az értékadás folyamata [8]

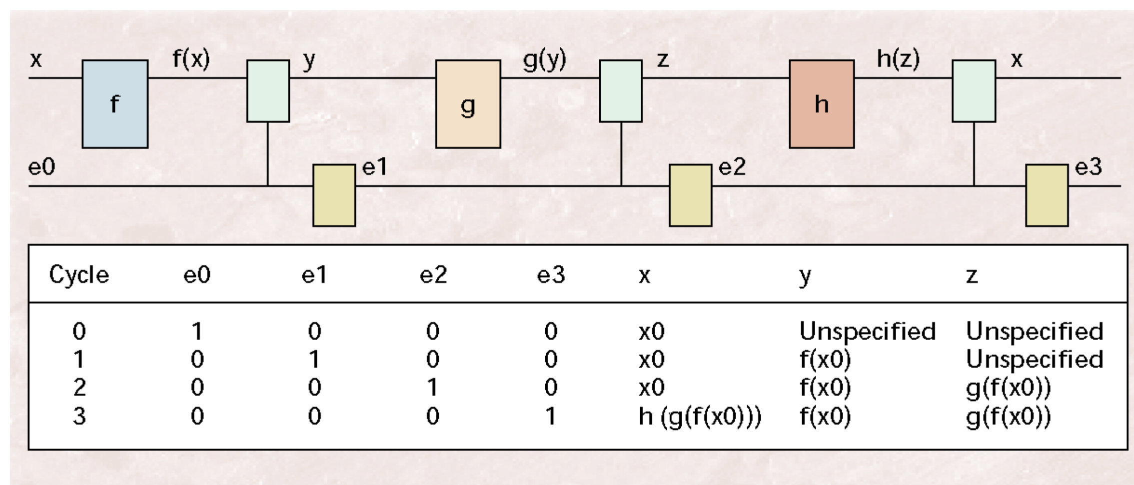
Az áramköri megvalósítás során az *f* függvényt egy kombinációs hálózat valósítja meg, *e(nable)* jel pedig akkor aktív, amikor az értékadásnak meg kell történnie. Itt azonban már figyelembe kell venni, hogy ha a regiszter *t* időpillanatban kapja meg az *x* értéket, a kombinációs logika kimenetén *f(x)* csak *t+Δt* (ahol *Δt* az f logika késleltetési ideje) idő múlva jelenik meg az új érték. Ebből azonban megállapíthatjuk, hogy a következő órajel nem következhet be *t+Δt*-nél hamarabb. Ez azonban meghatározza a legmagasabb órajelfrekvenciát, így (feltételezve, hogy az áramkörök ugyanazon órajelről működnek szinkron) a legnagyobb késleltetési idővel rendelkező *f* áramkör határozza meg az összes (vele együtt szinkron) áramkörnek a legmagasabb órajelfrekvenciáját.



4. *ábra:* A párhuzamos értékadás áramköri modellje [8]

Párhuzamos struktúra esetén azt feltételezzük, hogy két folyamat egymással egy időben (konkurensen) hajtódik végre, ebből következően az *e(nable)* jelet is egyszerre kapják. Ennek áramköri modelljét mutatja a 4. ábra.

Sorrendi struktúra áramkörben történő implementálását mutatja be az 5. ábra.



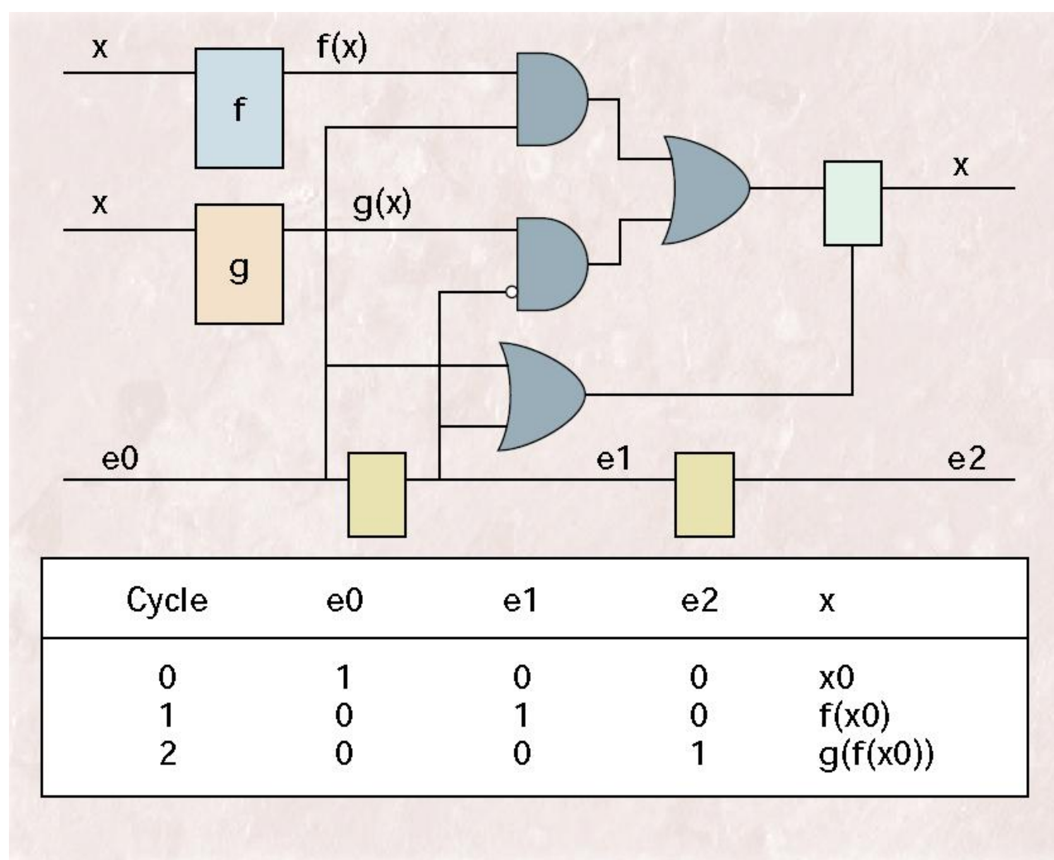
Cycle	e0	e1	e2	e3	x	y	z
0	1	0	0	0	x0	Unspecified	Unspecified
1	0	1	0	0	x0	f(x0)	Unspecified
2	0	0	1	0	x0	f(x0)	g(f(x0))
3	0	0	0	1	h (g(f(x0)))	f(x0)	g(f(x0))

5. *ábra:* A sorrendi értékadás áramköri modellje [8]

Itt sorrendben három értékadás történik az *e0*, *e1* és *e2* engedélyező jelekre: „*y := f(x)*", „*z := g(y)*" valamint „*x := h(z)*". A fenti vonalon láthatjuk a változókat tároló regisztereket és kombinációs logikákat, míg a lenti vonalon a vezérlési szerkezetet, amelyben *e(n)* meghatározza, hogy az értékadás mikor következzen be. Ez a vezérlés egy olyan állapotgépen alapul, amely egy időben egy és csakis egy *enable* jelet

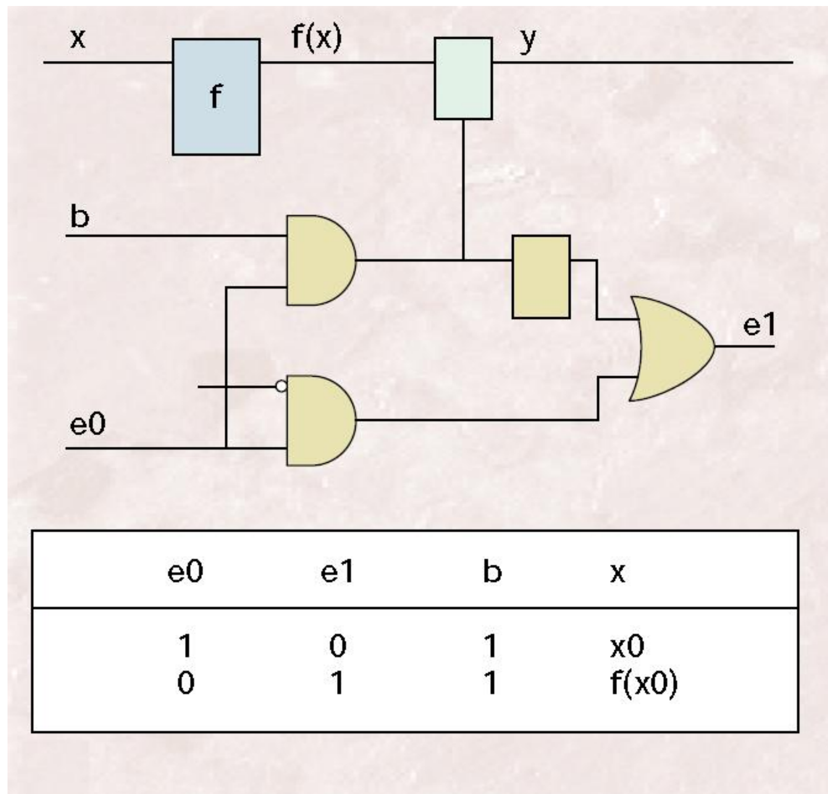
engedélyez. Ebben a példában azzal az egyszerűsítéssel éltünk, hogy egy változó csakis egyszer kap egy értéket. Általánosabban megfogalmazva a problémát, vagyis ha egy változó – különböző időpontban – kap különböző értékeket, az áramkör bonyolódik, egy újabb áramköri elemet, egy multiplexert kell implementálnunk. Tekintsük egy olyan értékadást, ahol „*x := f(x)*”, majd egy későbbi időpontban „*x := g(x)*”. (6. ábra)

Feltételes struktúrát a következők szerint tudunk deklarálni: „IF *b* THEN *S* END”, ahol b egy logikai (boole) változó, S pedig egy értékadás (pl.: *y := f(x)*). (7 .ábra) Ennek az implementációja csupán az enable jelet előállító vezérlési szerkezetben különbözik a 3. ábrán látható áramköri megoldástól. Ebből pedig egyértelműen látható, hogy a programban leírt vezérlési utasításokat az áramkörben az engedélyező jeleket vezérlő állapotgép realizálja, míg az értékadásokat és kifejezéseket a többi áramkör elem.



Cycle	e0	e1	e2	x
0	1	0	0	x0
1	0	1	0	f(x0)
2	0	0	1	g(f(x0))

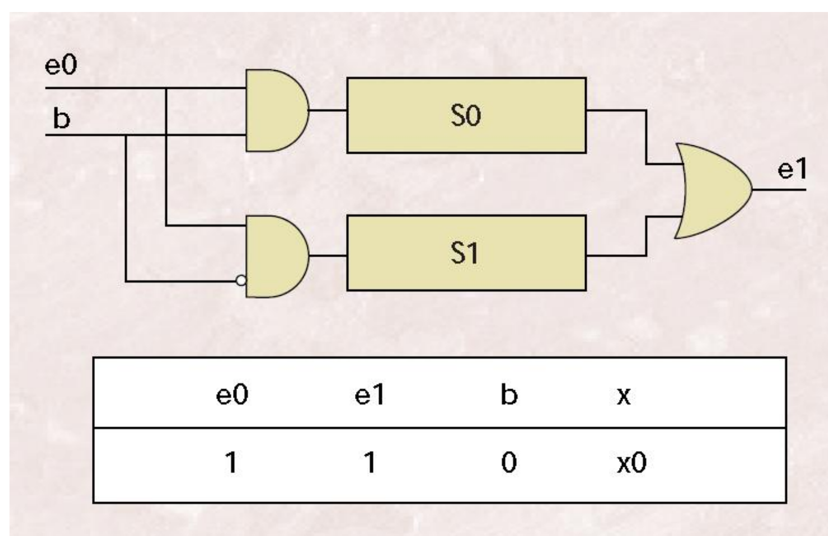
6. *ábra:* A többszörös sorrendi értékadás áramköri modellje [8]



e0	e1	b	x
1	0	1	x0
0	1	1	f(x0)

7. *ábra:* A feltételes értékadás (if-then) áramköri modellje [8]

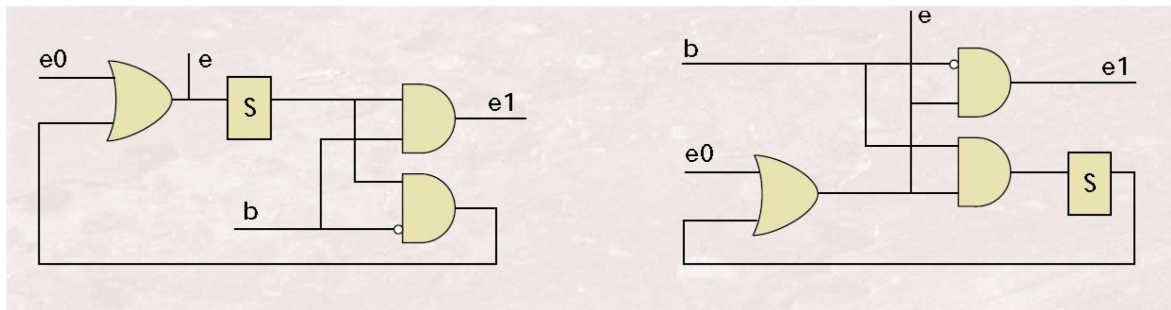
Az előzőek alapján kiegészítve a 7. ábrán látható vezérlési szerkezetet a 8. ábrán láthatjuk az „IF *b0* THEN *S0* ELSE *S1* END" kifejezés vezérlési szerkezetét.



e0	e1	b	x
1	1	0	x0

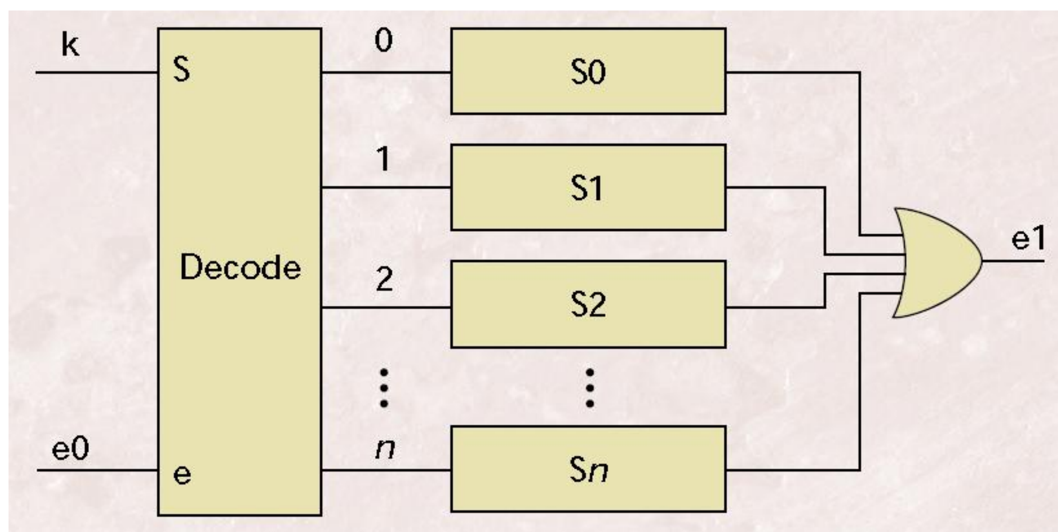
8. *ábra:* A feltételes értékadás (if-then-else, csak vezérlés) áramköri modellje [8]

A következőkben a ciklusszervező szerkezetek, a „REPEAT *S* UNTIL *b*” valamint "WHILE *b* DO *S* END” áramköri reprezentációjának a vezérlőáramköreit láthatjuk (9. ábra), bal oldalon a *repeat*-tel, jobb oldalon a *while*-lal.



9. *ábra:* A repeat és a while szerkezetek áramköri modellje [8]

Szelektív (case) szerkezetet (*case k of 0: S0 | 1: S1 | 2: S2 ...*) vezérlésének áramköri modelljét mutatja a 10. ábra.



10. *ábra:* A case szerkezet áramköri modellje [8]

Az előbbiek alapján beláthatjuk, hogy a leírt szabályok segítségével egy programban leírt értékadásokat, feltételes és ciklikus utasításokat le tudjuk fordítani, le tudjuk írni áramköri elemek segítségével. Ennek ellenére könnyen beláthatjuk azt is,

hogy már egy közepesen összetett program fenti szabályokkal történő áramkörré alakítása is nagyon bonyolult, akár többezres kapuból álló áramkört eredményez. Az áramköri elemek száma azonban csökkenthető, amennyiben sikerül a programozásban ismert szubrutinok és hívásaik áramkörben történő implementálása, vagyis a program különböző részei által ugyanannak az áramköri részegységnek egymás közti megosztása. Ennek érdekében megfigyelhetjük, hogy az ezidáig bemutatásra került áramkörök lényegében állapotgépek. Ha ezek után az összes szubrutint lefordítjuk egy ilyen állapotgéppé, akkor egy szubrutinhívást az alábbiak szerint kell kezelnünk. Először is fel kell függesztenünk a hívó állapotgép működését, majd el kell indítanunk (aktiválnunk kell) a hívott szubrutint. Mindezek után, a hívott szubrutin működésének befejezésekor folytatnunk kell a hívó szubrutin működését. Ennek elérése érdekében első intézkedésként a vezérlési struktúrában található minden regiszternek biztosítani kell egy közös enable jelet, amellyel felfüggeszthetjük, illetve újraindíthatjuk egy adott állapotgép működését. Következő lépésben implementálnunk kell egy vermet (stack), amelyben a felfüggesztett állapotgépeket azonosító számokat (tipikusan 0-tól n-ig) tárolhatjuk. Az állapotgépek sorszámát tartalmazó vermet legegyszerűbben egy m szó hosszúságú és n bit szélességű statikus RAM-mal, valamint egy, memóriacímeket előállító számlálóval modellezhetjük. Egy n bit széles szó minden egyes bitje egy szubrutint modellező áramkört azonosít. Ezek közül a bitek közül viszont csak egynek az értéke 1, mégpedig éppen annak, amely az újraindítandó szubrutint azonosítja. Így a szó kiolvasásával közvetlenül elő tudjuk állítani az újraindítandó szubrutint modellező áramkör számára (és csak az ő számára) az engedélyező enable jelet. Az így kialakított vermet alapvetően két jel működteti. Az egyik a *push* jel, amely megnöveli a számláló értékét, majd az alkalmazott bemenetet eltárolja a memóriában, valamint egy *pop* jelet, amely dekrementálja a számlálót. A *push* jel mindig akkor kerül aktiválásra, ha egy szubrutinhívást reprezentáló állapotregiszter aktiválódik, a *pop* jel pedig akkor aktiválódik, ha a vezérlés eléri egy szubrutint modellező áramkör utolsó állapotát.

Az eddigiek alapján eléggé nyilvánvalónak tűnik, hogy a szubrutinok jelentősen növelik egy áramkör komplexitását, éppen ezért jól megfontolandó, hogy hardveresen implementáljuk-e őket. Amikor egy programleírás alapján áramköröket automatikusan állítunk elő, a legfőbb célunk az implementált hardver minél optimálisabb kihasználása, ezen optimális kihasználtságot pedig akkor érhetjük el, ha a lehető legtöbb implementált

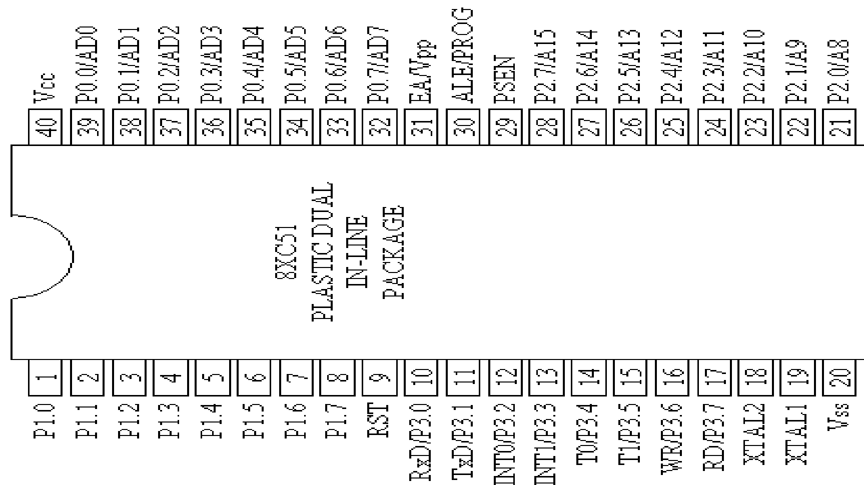
részáramkör hozzájárul a folyamat működéséhez. Ennek a legegyszerűbb módja a szekvenciális működések lehető legnagyobb mértékben történő minimalizálása, hiszen a hardver legnagyobb erőssége pont abban rejlik, hogy részáramkörei egymással párhuzamosan is képesek működni. Ugyan a párhuzamosság fogalma az utóbbi időben megjelent a szoftverfejlesztés világában is, azonban valódi párhuzamosságról ott is csak akkor beszélhetünk, ha a háttérben párhuzamosan működő áramköreink vannak, amelyek támogatják azt. A fentiek alapján röviden tehát akkor beszélhetünk jó áramkörről, ha annak a lehető legtöbb kapuja mindegyik órajelciklusnál részt vesz az eredmény előállításában.

2.3. A 8051-es mikrokontroller

Mint azt már a bevezetőben is említettük, az élet kvázi bármely területén megszámlálhatatlanul sok helyen találkozhatunk mikrovezérlőkkel és mikroprocesszorokkal. Azonban felmerülhet a kérdés, hogy mi a különbség a mikrokontrollererek és a mikroprocesszorok között? A két eszköz közötti határt nem is olyan könnyű meghúzni, ugyanis az többé-kevésbé folytonos. Általánosságban azt mondhatjuk, hogy a mikrokontroller egy olyan félvezetőáramkör, amelyben egy mikroprocesszorral még perifériafunkciókat megvalósító áramkör(öke)t is egyesítenek egy chipen. A mikrokontroller így tulajdonképpen egy chipen megvalósított számítógépes rendszer. Bizonyos mikrokontrollerekre használják a System on Chip (SoC) kifejezést is. A valóságban természetesen a CPU-val egy chipen megvalósított áramkörök száma igen sok lehet, így lehetőség van különböző ROM-ok (EPROM, EEPROM), RAM, Flash memória, USB-, ethernetinterfész, időzítők, watchdog, stb. integrálására is.

Elég sok mai mikrokontroller még a 70-es években kifejlesztett régi mikroprocesszorok architektúráján alapszik (ilyenek pl. a Zilog Z80, Freescale (Motorola) Coldfire), de természetesen ugyanúgy megtalálhatóak a már a kezdetektől kifejezetten mikrokontrollerként tervezett változatok is, mint pl. az Atmel AVR, PICmicro vagy a TI MSP430. Szóhosszúság szempontjából megkülönböztetünk 4, 8, 16 és manapság már 32 bites mikrokontrollert is, az alkalmazásokban eddig a 8 bitest használták legelterjedtebben, ugyanakkor egyre veszít piacából a 16 bites társaival szemben, azok egyre inkább csökkenő előállítási költségei miatt. Egyszerű feladatokhoz továbbra is alkalmazzák a 4 biteseket is.

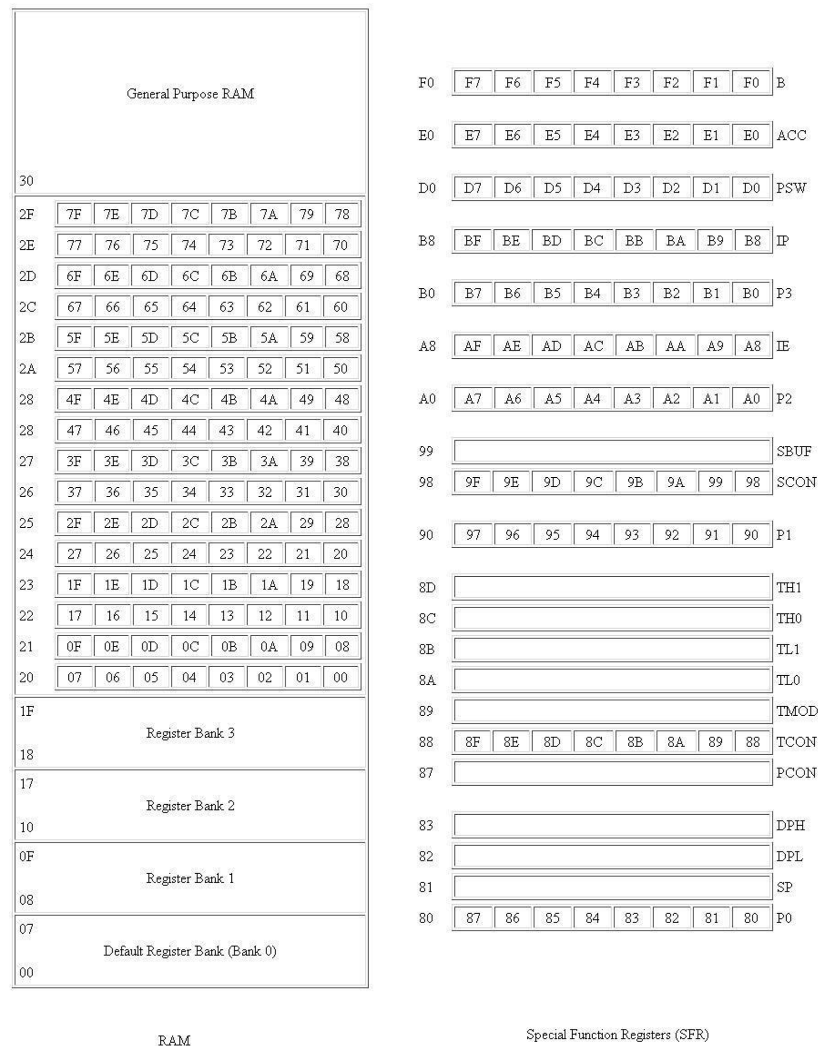
Az Intel 8051-es az 1980-ban bemutatott MCS-51 család egyik tagja. Specifikációja szerint egy 8 bites központi feldolgozóegységgel rendelkezik, amelynek 8 bites adatbusza, valamint 16 bites címbusza van. Így maximálisan 64 kilobyte-nyi memóriát tud megcímezni. Ezen kívül 1 chipre integrálva rendelkezik 4 kilobyte-nyi ROM memóriával ahonnan a végrehajtandó programot olvassa, valamint 256 byte-nyi RAM memóriával, amiből 128 byte áll a felhasználó rendelkezésére. Továbbá rendelkezik még 4 db egyenként 1 byte hosszúságú I/O porttal, UART interfésszel, ill. két 16 bites számlálóval.



11. *ábra:* A 8051-es 40 lábas tokozása [edsim51.com]

2.3.1. A 8051-es memóriaszervezése

A 8051-esnek 4096 byte integrált ROM memóriája van, így 12 bites a belső ROM címbusz és a 000h és FFFh címtartomány között található. Ide kerül beégetésre a végrehajtandó program, amit a ROM típusától függően már gyárilag beégethettek, vagy EPROM esetén a fejlesztő saját maga írhatja be a programját, és végesen sokszor módosíthatja azt. A 256 byte RAM memóriát 8 bites busszal címezhetjük és ezáltal a 00h és az FFh közé esik. A RAM egyik részén, 00h és 7Fh között a fejlesztő tárolhat adatokat, míg a RAM másik felének kitüntetett szerepe van, ugyanis a 80h és FFh közötti tartomány az ún. Special Function Registereknek (SFR) van fenntartva. A RAM azon 128 byte-ja ami a fejlesztő számára van fenntartva tartalmazza R0- tól R7-ig a nyolc 8 bites általános regisztert, alapértelmezésben 00h és 07h között, valamint 20h és 2Fh között tartalmaz egy ún. bitcímezhető (bit addressable) tömböt, ahol 128 bit értéke egyenként változtatható. A 30h és 7Fh közötti általános célú tartományban az írás és olvasás már csak byte-onként történhet.

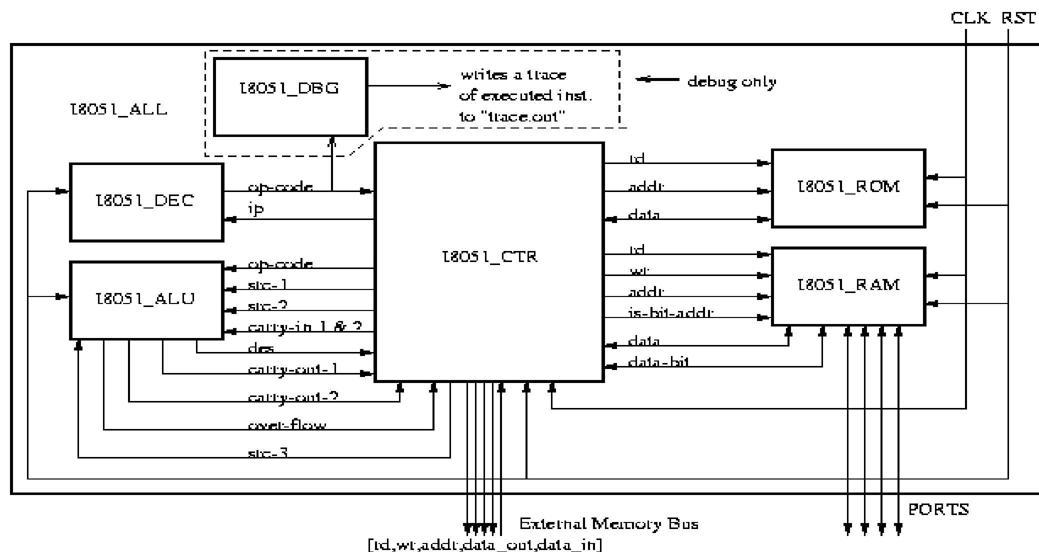


12. *ábra:* A 8051-es memóriatérképe [edsim51.com]

A RAM Special Function Regisztereknek fenntartott 128 byte-jából nem mindegyiket használja a 8051-es, vannak olyan memóriatartományok, amelyeket e mikrokontroller-család más tagjai használnak. Itt található meg többek között az akkumulátor, a stack pointer, a Program Status Word (PSW) (ami tartalmazza a flageket) és az I/O portok címei. Bizonyos regiszterek itt is bit-címezhetőek, többek között a négy I/O port is. A 8051-es többféle címzési módot támogat, méghozzá az immediate, regiszter, direkt, indirekt, relatív, abszolút és indexelt címzést.

2.3.2. A 8051-es VHDL modellje

A fejlesztések során felhasznált 8051-es mikrokontroller VHDL modellje az „eredeti” Intel 8051-essel teljes mértékben utasításkompatibilis. A leírás 9 különböző modulból és egy debugger modulból áll. A 9 modul külön-külön leírja a mikrokontroller egyes funkcionális egységeit, tehát külön modulban található az ALU, az utasításdekódoló, a RAM, a ROM, a vezérlés, és egy külső RAM memória. Ezeken kívül külön modulban (lib) találhatjuk a VHDL file-ok által használt csomagokat (packages), egy külön modul (all) egyesíti a fent felsorolt funkcionális egységeket, és megint csak egy külön modulban, a testbenchben (tsb) állítjuk elő a mikrokontroller meghajtására szolgáló külső jeleket. A debugger a végrehajtott utasítás opkódját egy log file-ba (trace.out) írja ki. A külvilág felé logikailag 11 vezetéken keresztül tartja a kapcsolatot, amelyek közül kettő 1 bites bemenet (CLK, RST). Külső memória illesztése esetén egy 8 bites adatbemeneten, két 1 bites kimeneten (rd, wr), egy 16 bites címkimeneten és egy 8 bites adatkimeneten keresztül tartja a kapcsolatot. Ezen túl rendelkezik a kontroller még négy 8 bites I/O interfésszel. Reset hatására az akkumulátor, a PC, a B regiszter a PSW regiszter nulla kezdőértéket kap, a SP kezdeti értéke 7.



Forrás: www.cs.ucr.edu/~dalton/i8051/i8051syn/

13. *ábra*: A modell blokkdiagramja

3. Kidolgozott eljárás

Ebben a pontban először ismertetem a munkám során felhasznált eszközöket, majd bemutatok egy újonnan fejlesztett VHDL modult, amelynek a későbbiekben tárgyalt, a 8051-es mikrokontrollerhez történő illesztésével egy olyan univerzális feladatokra felhasználható, IP core-nak tekinthető VHDL modellt sikerült kifejleszteni, amelyet akár egy FPGA-n realizálva fel tudunk programozni C nyelven írt algoritmusokkal. Mind az új VHDL modulnak, mind az új modul a 8051-essel történő együttes működésének helyes működését szimulációkkal igazolom, majd szintén szimulációk segítségével bemutatom, hogy az újonnan kifejlesztett modellen hogyan működik egy C nyelven írt program.

3.1. Felhasznált eszközök

Kutatásaim során célom az volt, hogy egy megfelelő mikroprocesszor VHDL [1, 2] modelljét úgy alakítsam át, hogy azt közvetlenül tudjam programozni C nyelven. Ennek eredményeként az Intel 8051-es [3] mikrokontroller egy szintetizálható VHDL modellje [4] került továbbfejlesztésre. A VHDL modell, valamint a fejlesztések helyességét a Mentor Graphics honlapjáról elérhető ModelSim Student Edition 10-es verziójával szimuláltam. A C nyelven írt tesztprogramok működését először egy VMWare virtuális gépre telepített 11.04-es verziószámú Ubuntu Linux disztribúció alatt futó GCC C compilerrel ellenőriztem. Miután a GCC fordító Intel x86-os architektúrára fordít, így szükséges volt egy olyan fordító használata, amely az Intel 8051-es utasításkészletére fordít, így felhasználásra került a nyílt forráskódú SDCC (Small Device C Compiler), valamint a Keil egy 8051-hez kifejlesztett C fordítója.

3.2. A VHDL modell programozása gépi kódban

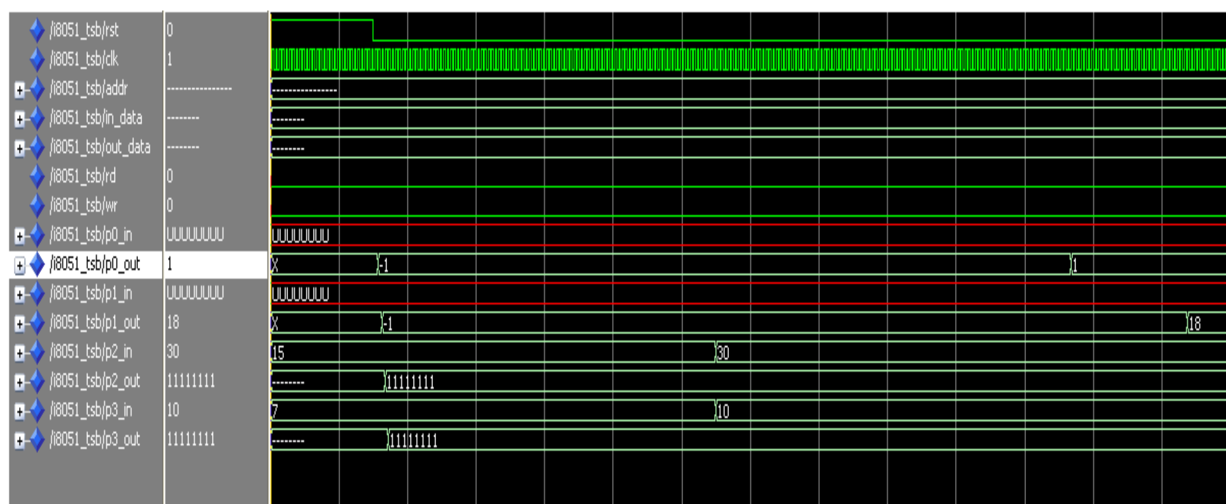
A mikrokontroller VHDL modelljének [4] tanulmányozása során felmerülhet a kérdés, hogy hogyan lehet ezt az eszközt újraprogramozni, ha egy másik programot akarunk rajta futtatni? Ehhez a ROM tartalmát kell átírni, ugyanis mindig onnan olvassa ki a végrehajtandó programot. Ez a mikrokontroller ezen VHDL leírásának gyenge pontja, ugyanis az eszköz által végrehajtandó programot csak egy új VHDL ROM modul írásával tudjuk megváltoztatni, vagyis ahányszor más programot akarunk futtatni a mikrokontrolleren, annyiszor kell belenyúlni a ROM modul viselkedési leírásába. Ebből következik, hogy a VHDL kód szintetizálása után, ill. az eszköz egy FPGA-n történő megvalósítása során a mikrokontroller által végrehajtandó program nem változtatható. Ez nehézkessé teszi a különböző programok tesztelését. Másik probléma, hogy ilyen módon a kontroller csak gépi kóddal programozható fel, ami ugyancsak hosszadalmassá teheti új programok implementálását. A fent leírtak megfigyelhetők a ROM modul eredeti viselkedési leírásában, ami alább látható:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use WORK.I8051_LIB.all;

entity I8051_ROM is
    port(rst      : in  STD_LOGIC;
         clk      : in  STD_LOGIC;
         addr     : in  UNSIGNED (11 downto 0);
         data     : out UNSIGNED (7 downto 0);
         rd       : in  STD_LOGIC);
end I8051_ROM;

architecture BHV of I8051_ROM is

    type ROM_TYPE is array (0 to 27) of UNSIGNED (7 downto 0);

   constant PROGRAM : ROM_TYPE := (

    "11010010", --SETB bit
    "10100000", --set P2 as input
    "11100101", --MOV A, direct
    "10100000", --A0h  /P2
    "11010010", --SETB bit
    "10110000", --set P3 as input
    "00100101", --ADD A, direct
    "10110000", --B0h /P3
    "11000000", --PUSH
    "11100000", --E0h /ACC
    "11100101", --MOV A, direct
```

```
    "10100000", --A0h  /P2
    "00100101", --ADD A, direct
    "10110000", --B0h /P3
    "11000000", --PUSH
    "11100000", --E0h /ACC
    "11010000", --POP
    "11100000", --E0h /ACC
    "11010000", --POP
    "11110000", --F0 /B
    "10000100", --DIV A/B
    --kvociens kiirasa
    "11110101", --MOV direct, A
    "10000000", --80h /P0
    --maradek kiirasa
    "11100101", --MOV, A, direct, B-bol az A-ba irjuk, hogy irathassuk
    "11110000", --F0h /B
    "11110101", --MOV direct, A
    "10010000", --90h /P1
    "00000000");-- NOP

begin
    process(rst, clk)
    begin
        if( rst = '1' ) then

            data <= CD_8;
        elsif( clk'event and clk = '1' ) then

            if( rd = '1' ) then
                data <= PROGRAM(conv_integer(addr));
            else
                data <= CD_8;
            end if;
        end if;
    end process;

end BHV;
```

A modul egyedbejelentésében deklaráljuk a modul határfelületét a környezetével, vagyis a portjait. Láthatjuk, hogy 4 bemeneti és 1 kimeneti portja van. Ezen 5 port közül 3-mal a vezérléssel tart fent kapcsolatot (13. ábra). Az *addr* egy unsigned típusú 12 bites bemenet, amivel a vezérlő modul megadja a következő utasítás címét a memóriában, az *rd* egy std_logic típusú bemenet, amivel a vezérlő modul engedélyezi a memóriaolvasást, míg a data egy szintén unsigned típusú 8 bites kimenet a vezérlő számára, ami tartalmazza a kérvényezett memóriacímen lévő utasítást vagy adatot. A modul építményében először is definiálunk egy ROM_TYPE típust, ami egy olyan tömb, ami olyan hosszú ahány bájtos a program és 8 bit széles unsigned. Ezek után deklarálunk egy állandót, ami ezen ROM_TYPE típusú, majd feltöltjük ezen

konstansot a programunk gépi kódjával. Ezzel tulajdonképpen deklaráltuk a mikrokontroller által elvégzendő programot. A modul fő részében egy folyamatot definiálunk, miszerint az *rst* érvényrejutása esetén a modul *data* kimenetén a lib modulban definiált „-“ don’t care értékeket vegyen fel. *Rst* = „0“ esetén viszont felfutó órajelélre az *addr* bemeneten kért címen lévő 8 bites információt adja ki a data kimenetre, feltéve, hogy *rd* is „1-es“ állapotban van.



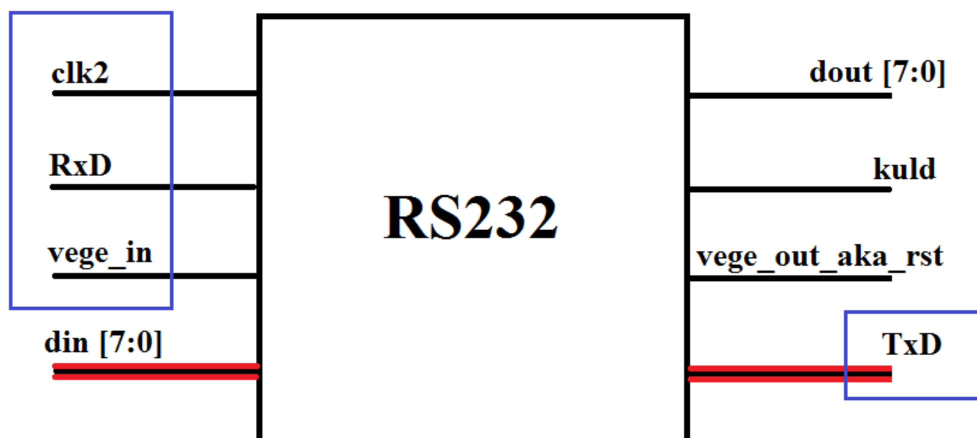
14. *ábra:* Az eredeti VHDL modell szimulációjának eredménye

A ROM modulban lévő program azt az egyszerű feladatot hajtja végre, hogy a 8051-es 2 bemenetére egymás után rákapuzott 2 számot páronként összeadja, majd elosztja egymással, és az osztás eredményét, valamint a maradékot kiírja egy-egy kimenetére. A mikrokontroller működését leszimulálva a 14. ábrán látható eredményt kapjuk a ROM modulban lévő programot végrehajtva.

3.3. Az RS232 VHDL modul

A processzor fel- és újraprogramozása a fenti megoldással láthatóan nehézkes és lassú, hiszen az új programot gépi kódban kell megadni, továbbá minden programváltoztatás után újra kell szintetizálni a mikrokontroller teljes VHDL modelljét. Ezen nehézségek kiküszöbölése érdekében kifejleszetésre került egy RS232 interfészt megvalósító VHDL modul azzal a céllal, hogy azt a mikrokontrollerhez illesztve, ennek segítségével képesek legyünk a mikrokontroller által végrehajtandó memóriát elérni és igény szerint újraprogramozni.

Az RS232 modul részletes tárgyalása előtt figyeljük először meg a modul ki- és bemeneti portjait. A modulnak egy fekete dobozát láthatjuk a 15. ábrán.



15. *ábra:* Az RS232 modul fekete doboza

Mint ahogyan az a képen is látható, a modulnak összesen 8 darab I/O interfésze van, 4 darab kimenet és 4 darab bemenet. A ki- és bemenetek 1-1 kivételtől eltekintve 1 bitesek, a 2 kivétel 8 bit széles. A pirossal jelölt portok az RS232-es modelljében implementálva vannak későbbi fejlesztésekhez, azonban azokat a mikrokontrollerhez illesztve jelenleg nem fogjuk felhasználni. Továbbá megemlítendő, hogy a kékkel bekarikázott portokon keresztül tartja a modul a kapcsolatot a „felhasználóval", míg a többi interfészen keresztül a mikrokontroller moduljaival fog kommunikálni.

A modul fejlesztése közben mindig első számú szempont volt, hogy a modul a lehető legegyszerűbben tudjon illeszkedni a 8051-es mikrokontroller modelljéhez. Miután - mint azt később látni fogjuk - az RS232-es interfész működése szempontjából az órajelfrekvencia alakulása kifejezetten kritikus szempont, és a szükéges órajelfrekvencia nagyságrendekkel különbözik a mikrokontroller órafelfrekvenciájától, így szükségszerű volt egy, a mikrokontroller órajelétől független órajel használata, amely a modulhoz annak *clk2* bemenetén keresztül kapcsolódik. A következő – *RxD* nevű – bemeneten bitenként fogadja a modul a számára küldendő adatokat, méghozzá az RS232-es szabványban leírtak szerint. Esetünkben tehát 1 byte adatot megfelelő időzítések betartása mellett egy „0“ startbittel kezdődő, és egy „1“-es stopbittel végződő 10 bites keretben tudunk átvinni. Sorban a következő bemenetünk a *vege_in*, amellyel jelezni tudjuk a modullal, hogy az adatátvitelt befejeztük, vagyis az utolsó byte adatunkat is átküldtük. A pirossal jelölt 8 bites *din* bemenetet most nem fogjuk használni, de a port ettől függetlenül működőképes és a modullal küldendő byte-ot kell neki értékül adni, annak érdekében, hogy azt a modul a *TxD* kimenetén az RS232 szabványban meghatározott időzítéssel és formátumban bitenként elküldje.

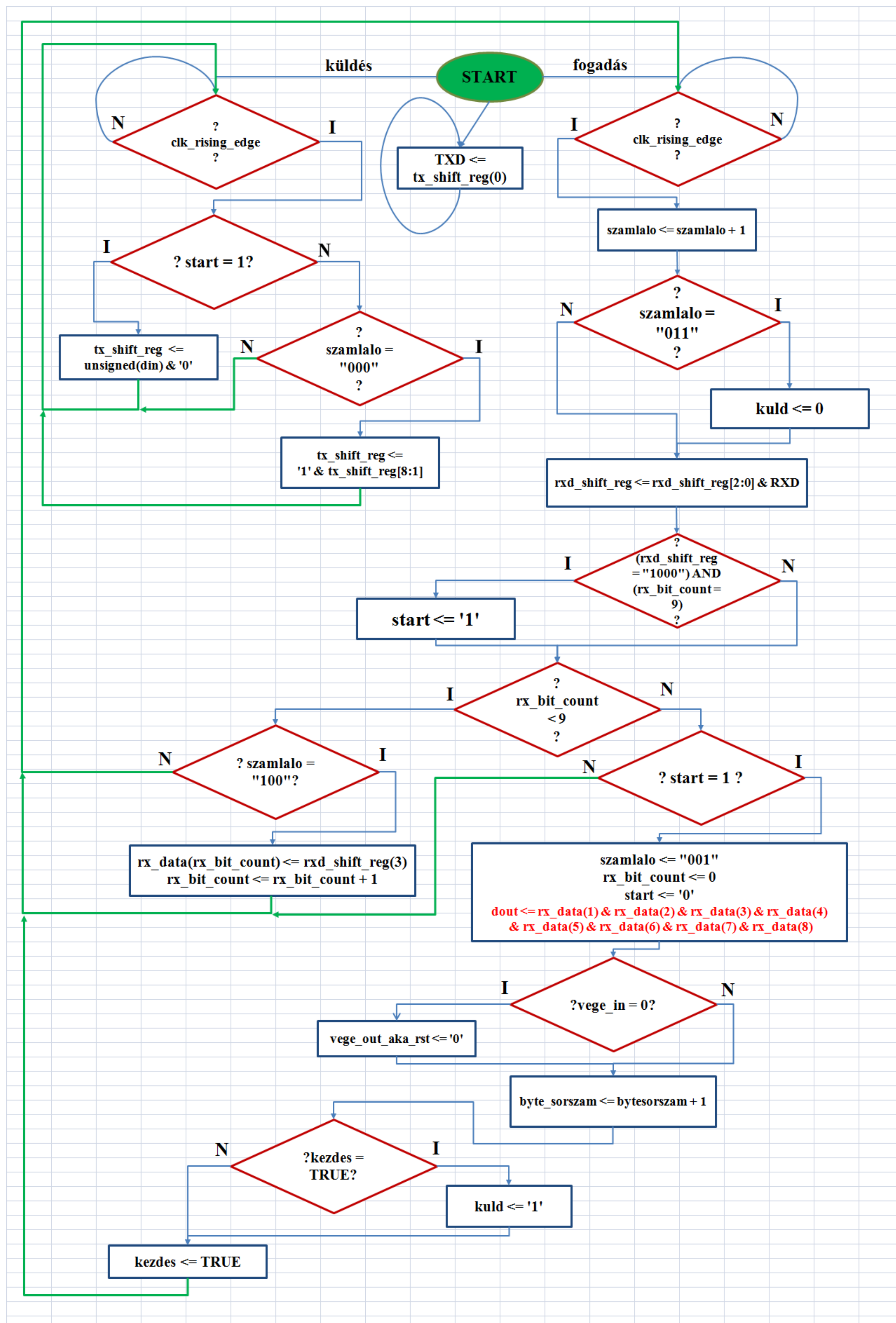
A kimeneti interfészek közül a modul a *dout* 8 bit széles portján át adja ki az *RxD* portján keresztül a modulnak bitenként küldött adatokat. A *kuld* kimenet azt a célt szolgálja, hogy elősegítsük annak a modulnak a működését, amely az RS232-es *dout* kimenetéről olvassa az egymás után érkező byte-okat, ugyanis ezen a kimeneten egy rövid ideig tartó impulzusszerű jellel tudjuk jelezni az *dout*-ot olvasó modulnak, hogy egy új, kiolvasandó byte érkezett a *dout*-ra. A kicsit hosszú nevű *vege_out_aka_rst* kimeneti jelnek majd a modul a mikrokontrollerhez történő illesztésénél lesz jelentősége, ugyanis ez a jel lesz az mikrokontrollert alkotó összes modul reset (*rst*) jele, méghozzá a külső (felhasználói) reset helyett. Végül a modul a *TxD* jelen keresztül bitenként továbbítja a külvilág felé a *din* bemeneten lévő byte-ot. A modul határfelületeinek áttekintése után az alábbiakban láthatjuk az RS232 VHDL kódját.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity RS232 is
    Port ( clk2 : in  STD_LOGIC;
           TXD  : out STD_LOGIC := '1';
           RXD  : in  STD_LOGIC;
           din  : in  STD_LOGIC_VECTOR (7 downto 0);
           dout  : out STD_LOGIC_VECTOR (7 downto 0) :=
"00000000";
           kuld              : out STD_LOGIC := '0';
           vege_out_aka_rst : out STD_LOGIC := '1';
           vege_in           : in STD_LOGIC);
end RS232;

architecture viselkedesi of RS232 is

signal szamlalo      : UNSIGNED(2 downto 0) := "000";
signal tx_shift_reg   : UNSIGNED(8 downto 0) := "111111111";
signal rxd_shift_reg  : STD_LOGIC_VECTOR (3 downto 0) := "1111";
signal start          : STD_LOGIC := '0';
signal rx_data        : STD_LOGIC_VECTOR (8 downto 0) :=
"000000000";
signal rx_bit_count   : integer range 0 to 9 := 9;
signal byte_sorszam   : integer := 0;

begin

kuldes: process

   begin
      wait until rising_edge(clk2);
      if (start = '1') then
         tx_shift_reg <= unsigned(din) & '0';
      elsif (szamlalo="000") then
         tx_shift_reg <= '1' & tx_shift_reg(8 downto 1);
      end if;
   end process;
   TXD <= tx_shift_reg(0);

fogadas: process

   variable kezdes : boolean := FALSE;

    begin
      wait until rising_edge(clk2);
      szamlalo <= szamlalo+1;

      if (szamlalo = 3) then
        kuld <= '0';
      end if;
```

```
      rxd_shift_reg <= rxd_shift_reg(2 downto 0) & RXD;

      if (rxd_shift_reg = "1000" and rx_bit_count=9) then
         start <= '1';
      end if;

      if (rx_bit_count<9) then
         if (szamlalo="100") then
            rx_data(rx_bit_count) <= rxd_shift_reg(3);
            rx_bit_count <= rx_bit_count + 1;
         end if;
      else
         if (start = '1') then
            szamlalo <= "001";
            rx_bit_count  <= 0;
            start     <= '0';

            dout <= rx_data(1) & rx_data(2) & rx_data(3) &
rx_data(4) & rx_data(5) & rx_data(6) & rx_data(7) & rx_data(8);

            if (vege_in = '0') then
                vege_out_aka_rst <= '0';
            end if;

            byte_sorszam <= byte_sorszam + 1;

            if (kezdes = TRUE) then
                kuld <= '1';
            end if;
            kezdes := TRUE;

         end if;
      end if;
   end process;
end viselkedesi;
```

A jobb átláthatóság és érthetőség érdekében a kód működését a 16. ábrán látható folyamatábrán mutatjuk be. Az ábrán látható bemeneti jelek kezdeti értékei az alábbiak.

szamlalo[2:0]	*:= „000“*
tx_shift_reg[8:0]	*:= „111111111“*
rxd_shift_reg[3:0]	*:= „1111“*
start	*:= ‚0‘*
rx_data[8:0]	*:= „000000000“*
rx_bit_count	*:= 9*
byte_sorszam	*:= 0*



16. *ábra*: Az RS232 modul működésének folyamatábrája

A modul elindítása után, alapvetően három folyamat fut párhuzamosan végtelen ciklusban, ebből egy a küldés, egy pedig a fogadás viselkedési leírását tartalmazó process. A 3. párhuzamos folyamat a *tx_shift_reg* 9 bites unsigned típusú jel 0. bitjének értékét rendeli hozzá minden időpillanatban a modul TxD kimeneti portjához. A modul felépítéséből kifolyólag egyszerűbbnek tűnik, ha a fogadást megvalósító folyamatot tekintjük át elsőként.

A modul fogadás process-e az elindulása után tulajdonképpen egy folyamatosan futó végtelen ciklusnak tekinthető, amely minden egyes felfutó órajelélre lefut, vagyis a process minden egyes iterációjának a triggereseménye a felfutó órajel. Ha ez bekövetkezett, akkor a *szamlalo* jel értékének inkrementálása után – amennyiben a *szamlalo* jel eléri a „011”=3 értéket –, a *kuld* jel kinullázásra kerül. A *kuld* jel az első iterációban természetesen egyébként is 0, ám a későbbi iterációk során beállított 1 értéke azt hivatott jelezni a modul kimenetén keresztül, hogy a modul *dout* kimenetén egy új byte áll rendelkezésre. Miután a jelzés megtörtént, még 3 órajelciklusig fent tartjuk a *kuld* jelet 1-ben, majd ezek után kinullázzuk egészen a következő új *dout* előálltáig (a *kuld* jel 1-be történő állításáról később lesz szó). A következő lépésben az *rxd_shift _reg* nevű shift_register legalsó bitjére beírjuk az RXD aktuális értékét és ezt minden órajel felfutó élre megtesszük. Amennyiben az RXD bemeneten folyamatosan stopbitek vannak, vagyis a bemenet logikai 1 értéket vesz föl, addig az *rxd_shift_reg* meg fog egyezni az inicializálási értékével, vagyis végig „1111” értéket vesz föl, hiszen a 4 bit minden egyes balra történő eltolásakor az RXD értékét felvevő legalsó bithelyiérték is 1 lesz.

Ha el akarjuk kezdeni az adást a modulnak (vagyis azt akarjuk, hogy a modul fogadjon), akkor nagyon fontos, hogy az elküldendő biteket 104 µs-os időközönként küldjük, ugyanis ez éppen 8 órajelciklusnak felel meg, és a modul működése ezt az időzítést feltételezi. A továbbítandó byte előtt egy startbitet kell küldenünk, amit az RXD 104 µs-ig tartó logikai 0-ba történő állításával tudjuk elérni.

Ezt a startbitet kell érzékelnünk, annak érdekében, hogy a modul tudja, hogy vennie fog kelleni egy neki küldött adást. Abban a pillanatban, hogy az RXD 0 lesz, az innentől számított 3. órajelciklusra az *rxd_shift_reg* felveszi az „1000” értéket, hiszen három órajel alatt 3 ’0’ „érkezett be” jobbról. Miután az *rxbitcount* kezdőértéke 9 volt ezért teljesülni fog annak az összetett feltétele, hogy a *start* jel 1 legyen. Amennyiben a *start*

jel a startbitnek köszönhetően bebillent 1-be, tudjuk, hogy most 8 „fontos” bit következik, és felkészülünk ezek fogadására. Ehhez a *szamlalo* jelet „001”-be állítjuk, az *rx_bit_count*-ot (amely azt számolja, hogy a byte-nak éppen melyik bitjét fogadjuk), és a *startot* kinullázzuk, a *dout* kimenetre pedig kiadjuk az előzőleg fogadott byte eredményét, amelyet az *rx_data* felső 8 bitje tartalmaz. Annak érdekében, hogy az 1. byte beolvasásakor a start jel hatására ne küldjük el a *dout*-on keresztül az *rx_data* első olvasás előtti (nyilvánvalóan hamis) értékét, ezért a *kuld* kimenetet csak a *start* jel második 1-be kerülése után állítjuk majd 1-be. A továbbiakban a *kuld* kimeneten keresztül jelezzük az RS232 *dout* portjára csatlakozó modulnak, hogy új adat érhető el a *dout* kimeneten. Amennyiben a *vege_in* bemenetről azt a jelet kapjuk, hogy az összes adat átküldésre került (ekkor *vege_in* = 0), akkor szintén a *start* = 1 értékére kinullázzuk, a *vege_out_aka_rst* kimenetet, amely jel segítségével majd a modul a mikrokontroller mellett történő implementálása során el tudjuk indítani a kontrollert. Most tehát, hogy a *szamlalo* = 1 és az *rx_bit_count* = 0, a szamlalo minden órajelperiódusra megnöveli eggyel a *szamlalo* értékét, valamint minden órajelperiódusban eltolja eggyel balra az *rxd_shift_reg*-et és a legalacsonyabb helyiértékére beírja az RXD aktuális értékét. Mindez 3 órajelciklus hosszan ismétlődik, mígnem a *szamlalo* eleri a 4, vagyis az „100” értéket, ahol értéket ad az *rx_data*-nak, amelybe bithelyesen beolvassuk az *rxd_shift_reg* legfelsőbb bitjének értékét, vagyis az RXD azon értékét, amelyet pont 3 órajelperiódussal azelőtt mentettünk el az *rxd_shift_reg* legalsó helyiértékén és mostanra a legfelsőbb helyiértékre tolódott. Tehát a 0. bitet (a startbitet) elmentettük a 9 bit széles *rx_data* 0. bitjén, majd megnöveltük 1-gyel az *rx_bit_count* értékét. Minden felfutó órajelre növeljük a *szamlalo* értékét és az *rxd_shift_reg* balratolása után a legalacsonyabb helyiértékén mentjük el az RXD aktuális értékét. Mindezt addig, amíg a számláló át nem fordul és el nem éri újra az „100”-t, amikor a byte következő bitjének 8 órajelperiódusig kitartott értékének közepén mintavételezett értékét be nem olvassuk hasonló módon. Ha az *rx_bit_count* elérte a 9-et, akkor addigra a startbitet és az utána következő byte-ot elmentettük bithelyesen az *rx_data*-ban. Az *rx_data* értékének a dout kimenetnek történő átadásakor figyelembe kell venni, hogy az *rx_data*-ban az helyiértékek megfordultak, és az *rx_data*[1]-ben van legmagasabb helyiérték az *rx_data*[8]-ban a legkisebb. Mindezek

után a következő startbit hatására a modul *dout* kimenetén megjelenik helyesen a teljes byte.

Rátérve a küldő modul működésére, azt is egy a fogadó process-szel párhuzamosan futó processben definiáltuk. Ahogyan a fogadó processnek, ennek is minden iterációja egy felfutó órajellel kezdődik. A modul egy a *din* nevű bemenetére kapuzott 8 bites adatot küld el a TXD kimenetén keresztül. A teljes modul full duplex módon működik, és ahhoz, hogy adni tudjon a *kuldes* process, a modulnak az RXD bemenetére egy startbitet kell adni, annak érdekében, hogy a *start* jel 1-be váltson. Ekkor ugyanis a *din*-re kapuzott adatot beírjuk a 9 bit széles tx_shift_reg legfelső 8 pozíciójára és konkatenáljuk egy 0-val a 0. bit helyén. Miután ez a 0. biten lévő 0 bit azonnal megjelenik a TXD kimeneten, ezt tekintjük a *kuldes* process által küldött startbitnek. A *szamlalo*t itt is időzítésre használjuk fel, mert az pontosan 8 órajelciklus alatt fordul körbe, így pont annyi idő alatt, ameddig a TXD kimeneten érvényben kell tartanunk egy bitet. Természetesen ez az idő itt is 104 μs. Így a szamlalo minden egyes nullátmeneténél a tx_shift_reg-et eltoljuk jobbra, ezzel annak 0. bitjére tolva az aktuálisan átviendő bitet, ami mint tudjuk azonnal megjelenik a TXD kimeneten. Így minden 8 órajelciklusonként shiftelünk egyet jobbra, egészen addig, míg az egész, *din*-en beolvasott elküldendő byte-ot ki nem toltuk jobbra. A *tx_shift_reg* 9. bitje – ami biztosan 1-es – lesz a stopbit. A küldés végén a *tx_shift_reg* csupa egyessel lesz feltöltve. A következő byte küldéséhez pedig megint meg kell várnunk a start jel 1 értékét.

3.4. Az RS232-es modul testbench-e

A fejlesztett modul működésének tesztelését egy erre a célra írt testbench-csel fogjuk végrehajtani, amely egy Intel hex formátumú file-ból fogja kiolvasni az RS232-nek (a későbbiekben az RS232-n keresztül a mikrokontrollernek) küldendő adatokat. Miután azonban a hex file-ban tárolt adatok nem sorrendben vannak, ezért az adatokat még mielőtt el tudnánk küldeni őket, egy sorrendi bináris formátumra kell hoznunk. Ezt az átalakítást magában a testbenchben fogjuk elvégezni, mégpedig úgy, hogy a testbenchben definiálunk egy 4 Kb méretű memóriát (ez a mikrokontroller programmemóriájával megegyező méret), és abban sorrendbe rendezzük a hex file-ban található adatokat.

Ehhez azonban először tekintsük meg az Intel hex file formátumot. Egy olyan, bináris adatok tárolására szolgáló file formátumról beszélünk, amely ASCII-formátumban áll elő, és a benne lekódolt bináris byte-okat pedig 2 hexadecimális szám ábrázolja. Egy ilyen Intel hex file példája:

```
:100003008B128A1389148D1574011200FBAB12AA8B
:1000130013A914F58275830012010D751602AD162E
:10002300ED3395E0FCC3ED95157480F86C98503E64
...
```

Értelmezzük a fenti sorokat: a file rekordok egymásutánjából áll, minden rekord egy kettősponttal kezdődik. A kettőspont utáni két szám (az első byte), a rekord hosszáról ad információt. Ezek szerint, ha az első sort nézzük, példánk első rekordja 10_{hex} (16_{dec}) byte hosszú. A 3., 4., 5. és 6. pozícóban lévő számok adják a rekord kezdőcímét. Esetünkben a rekordban található első adat 0003_{hex} címen található. A 7. és a 8. pozícióból a rekord típusát olvashatjuk ki. A vizsgált sorban ez 00, vagyis a rekord data típusú. A 9. pozícióban lévő számtól egészen a 40. pozícióig bezárólag a rekordban tárolt kereken 16 byte-nyi adatot láthatjuk. A rekord utolsó byte-ja egy ellenörzőösszeg. A modulunk testbench-ének a kódját az alábbiakban mutatjuk be.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY tb_RS232 IS
END tb_RS232;

ARCHITECTURE stimulus OF tb_RS232 IS

--  (unit under test)
     COMPONENT RS232
     PORT(
          clk2              : IN std_logic;
          RXD               : IN std_logic;
          din               : IN std_logic_vector(7 downto 0);
          TXD               : OUT std_logic;
          dout              : OUT std_logic_vector(7 downto 0);
          kuld              : OUT std_logic;
          vege_out_aka_rst  : OUT std_logic;
          vege_in           : IN std_logic
          );
     END COMPONENT;

     --bemenet
     SIGNAL iclk2               :  std_logic := '0';
     SIGNAL iRXD                :  std_logic := '1';
     SIGNAL idin                :  std_logic_vector(7 downto 0)
:= (others=>'0');
     SIGNAL ivege_in            : std_logic := '1';

     --kimenet
     SIGNAL iTXD                : std_logic;
     SIGNAL idout               : std_logic_vector(7 downto 0);
     SIGNAL ikuld               : std_logic;
     SIGNAL ivege_out_aka_rst : std_logic;

type ROM_TYPE is array (0 to 4095) of UNSIGNED (7 downto 0);

function String_To_Integer(Some_Char: character) return integer
     is variable Return_Value: integer := 0;
     begin
       case Some_Char is
          when '0' => Return_Value := 0;
          when '1' => Return_Value := 1;
          when '2' => Return_Value := 2;
          when '3' => Return_Value := 3;
```

```
            when '4' => Return_Value := 4;
            when '5' => Return_Value := 5;
            when '6' => Return_Value := 6;
            when '7' => Return_Value := 7;
            when '8' => Return_Value := 8;
            when '9' => Return_Value := 9;
            when 'A' => Return_Value := 10;
            when 'B' => Return_Value := 11;
            when 'C' => Return_Value := 12;
            when 'D' => Return_Value := 13;
            when 'E' => Return_Value := 14;
            when 'F' => Return_Value := 15;
            when others => null;
         end case;
      return(Return_Value);
      end function String_To_Integer;


function String_To_Unsigned(Some_Char: character) return
unsigned is variable Return_Value: unsigned  (3 downto 0);
      begin
        case Some_Char is
            when '0' => Return_Value := "0000";
            when '1' => Return_Value := "0001";
            when '2' => Return_Value := "0010";
            when '3' => Return_Value := "0011";
            when '4' => Return_Value := "0100";
            when '5' => Return_Value := "0101";
            when '6' => Return_Value := "0110";
            when '7' => Return_Value := "0111";
            when '8' => Return_Value := "1000";
            when '9' => Return_Value := "1001";
            when 'A' => Return_Value := "1010";
            when 'B' => Return_Value := "1011";
            when 'C' => Return_Value := "1100";
            when 'D' => Return_Value := "1101";
            when 'E' => Return_Value := "1110";
            when 'F' => Return_Value := "1111";
            when others => null;
         end case;
      return(Return_Value);
      end function String_To_Unsigned;

      BEGIN

       uut: RS232 PORT MAP(
```

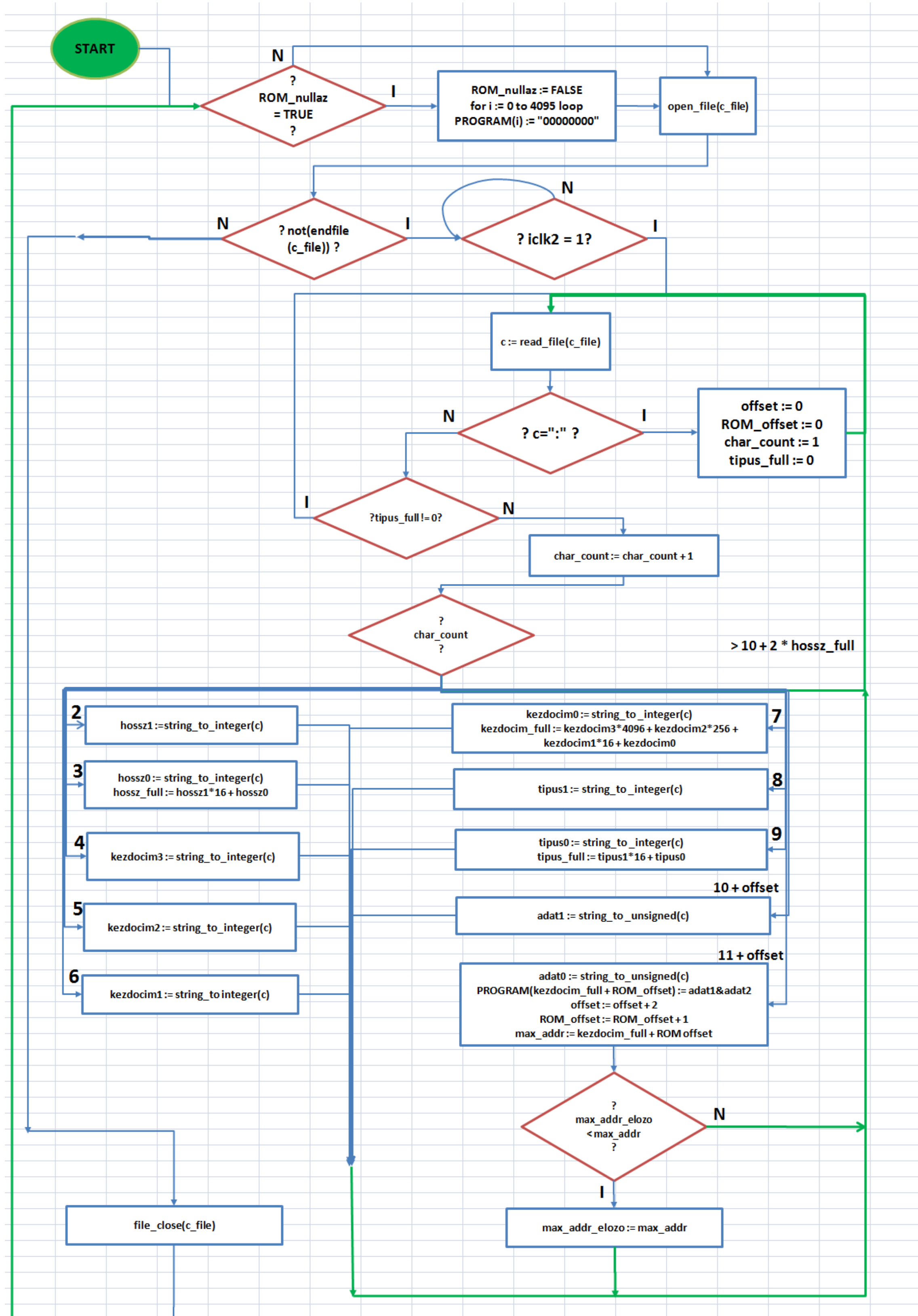
```
                clk2                => iclk2,
                TXD                => iTXD,
                RXD                => iRXD,
                din                => idin,
                dout               => idout,
                kuld               => ikuld,
                vege_out_aka_rst   => ivege_out_aka_rst,
                vege_in            => ivege_in
          );

iclk2 <= not iclk2 after 6500 ns; -- f = Baudrate * 8 = 13 us

        idin <= "11001100" after 100 ms;

       process

             type characterFile is file of character;
             file        cFile: characterFile;

             variable  c             : character;
             variable  Fstatus       : FILE_OPEN_STATUS;
             variable  char_count    : integer := 0;
             variable  hossz0        : integer := 0;
             variable  hossz1        : integer := 0;
             variable  hossz_full    : integer := 0;
             variable  kezdocim0     : integer := 0;
             variable  kezdocim1     : integer := 0;
             variable  kezdocim2     : integer := 0;
             variable  kezdocim3     : integer := 0;
             variable  kezdocim_full : integer := 0;
             variable  tipus0        : integer := 0;
             variable  tipus1        : integer := 0;
             variable  tipus_full    : integer := 0;
             variable  adat0         : UNSIGNED (3 downto 0);
             variable  adat1         : UNSIGNED (3 downto 0);
             variable  offset        : integer := 0;
             variable  ROM_offset    : integer := 0;
             variable  ROM_nullaz    : boolean := true;
             variable  send_byte     : unsigned(7 downto 0);
             variable  decode_vege   : boolean := TRUE;
             variable  PROGRAM       : ROM_TYPE;
             variable  max_addr      :integer:=0;
             variable  max_addr_elozo:integer:=0;
             variable  file_read     : boolean := TRUE;

     BEGIN --process beginje
```

```
            if (ROM_nullaz = true) then
                  ROM_nullaz := false;
                  for i in 0 to 4095 loop-- kinullázzuk a ROM-ot
                      PROGRAM(i) := "00000000";
                  end loop;
            end if;

file_open(Fstatus, cFile,
"C:\Users\Balint\Desktop\VHDL_8051_kieg\fib.hex", read_mode);

            while (not endfile(cfile)) loop

                  wait until iclk2 = '1';
                  read(cfile, c);

                  if (c = ':') then
                        offset := 0;
                        ROM_offset := 0;
                        char_count:= 1;
                        tipus_full := 0;
                        next;
                  end if;

                  if (tipus_full /= 0) then
                        next;
                  end if;

                  char_count := char_count + 1;

                  if (char_count = 2) then
                          hossz1 := string_to_integer(c);
                  end if;

                  if (char_count = 3) then
                          hossz0 := string_to_integer(c);
                          hossz_full := hossz1*16 + hossz0;
                  end if;

                  if (char_count = 4) then
                          kezdocim3 := string_to_integer(c);
                  end if;

                  if (char_count = 5) then
                          kezdocim2 := string_to_integer(c);
                  end if;
```

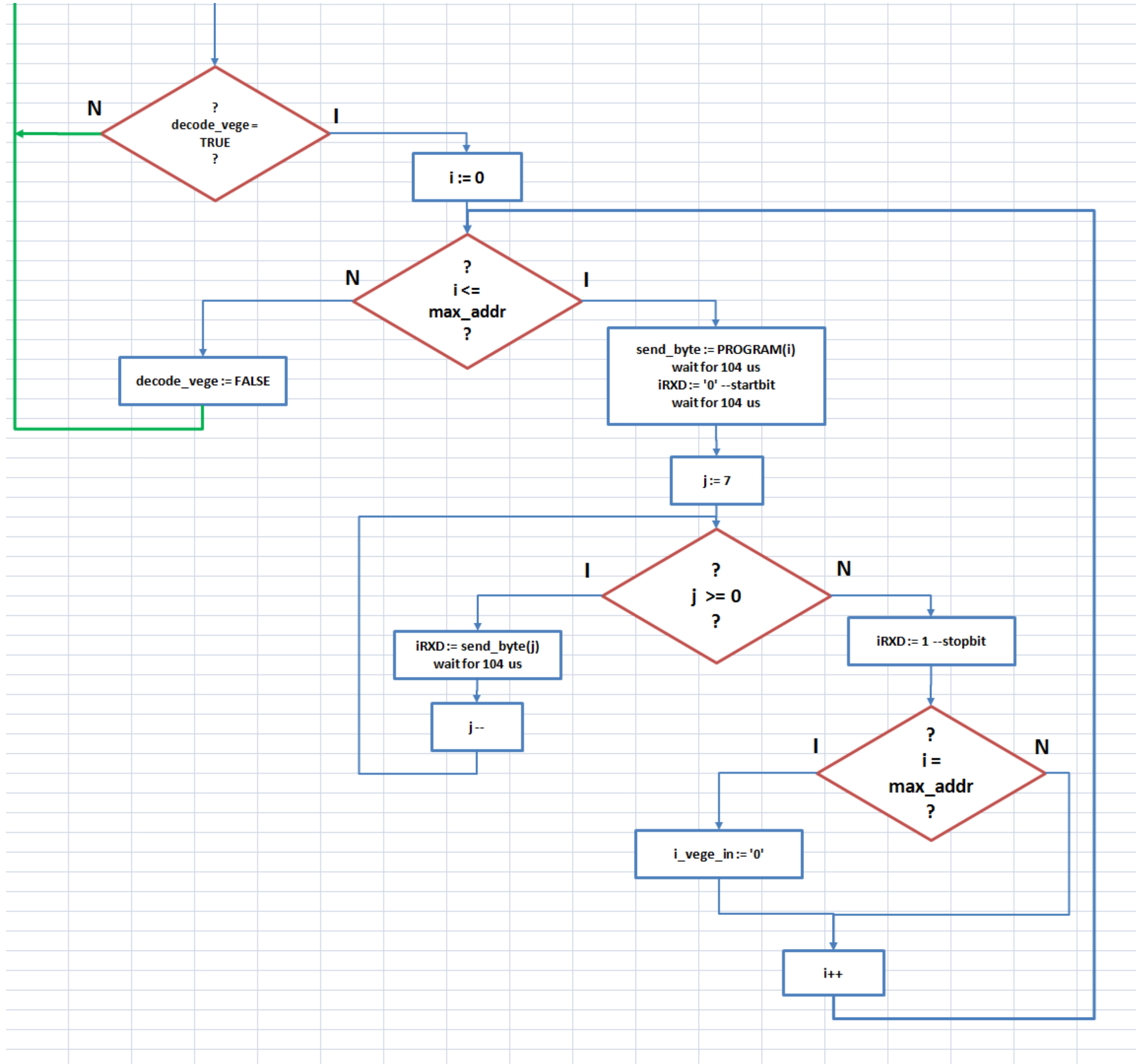
```
            if (char_count = 6) then
                    kezdocim1 := string_to_integer(c);
            end if;

            if (char_count = 7) then
                    kezdocim0 := string_to_integer(c);
                    kezdocim_full := kezdocim3*4096 +
kezdocim2*256 + kezdocim1*16 + kezdocim0;
            end if;

            if (char_count = 8) then
                    tipus1 := string_to_integer(c);
            end if;

            if (char_count = 9) then
                    tipus0 := string_to_integer(c);
                    tipus_full := tipus1*16 + tipus0;
            end if;

            if (char_count >= 10 + 2*hossz_full) then
              next;
            end if;

            if (char_count = 10 + offset) then
                    adat1 := string_to_unsigned(c);
            end if;

            if (char_count = 11 + offset) then
                    adat0 := string_to_unsigned(c);
                    PROGRAM(kezdocim_full + ROM_offset) :=
adat1&adat0;
                    offset := offset + 2;
                    ROM_offset := ROM_offset + 1;
                    max_addr:=kezdocim_full + ROM_offset;

                    if (max_addr_elozo < max_addr) then
                      max_addr_elozo := max_addr;
                    end if;

            end if;

      end loop;

      file_close(cfile);
      if (decode_vege = TRUE) then
          for i in 0 to max_addr loop
```

```
                    send_byte := PROGRAM(i);
                    wait for 104 us;
                        iRXD <= '0'; --startbit
                    wait for 104 us;
                        for j in 7 downto 0 loop
                            iRXD <= send_byte(j);
                            wait for 104 us;
                        end loop;
                        iRXD <= '1'; --stopbit;

                        if (i = max_addr) then
                            ivege_in <= '0';
                        end if;

                    end loop;

                    decode_vege := FALSE;
                  end if;
                end process;
END;
```

A testbenchben (amelynek folyamatábráját a 17. a és b ábrán láthatjuk) az RS232 modul példányosítása, valamint a példányosított modulhoz a testbenchben kapcsolódó jelek deklarálása után két függvényt definiálunk, amelyek közül az egyik egy string-integer, míg a másik egy string-unsigned típuskonverziót hajt végre. Az egész testbench egy processből áll, ami alapvetően 3 részre osztható. Az elején egy előzőleg definiált PROGRAM nevű memória tartalmát nullázzuk ki egy ciklussal, majd szintén egy ciklus segítségével olvassuk ki egy Intel hex file tartalmát, és byte-onként a hex file-ban meghatározott címük szerint, 0-tól kezdve sorrendben feltöltjük az előbb kinullázott PROGRAM nevű memóriát. Ezen ciklus bejezésekor a PROGRAM nevű változó tartalmazza a hex file tartalmát, ami nem más, mint az illesztés után majd a mikrokontrolleren futtatandó kód. A következő és egyben utolsó ciklus az előzőleg feltöltött PROGRAM nevű memória tartalamát olvassa ki úgy, hogy minden byte-ot felbont 8 bitre, valamint minden bájtot ellát egy start- és egy stopbittel. Így minden byte átviteléhez egy 10 bites keretet alkot, amelynek byte-jait megfelelő időzítéssel küldi el az RXD bemenetre.



17. *a. ábra*: Az RS232-es modul testbench-e (1. rész)



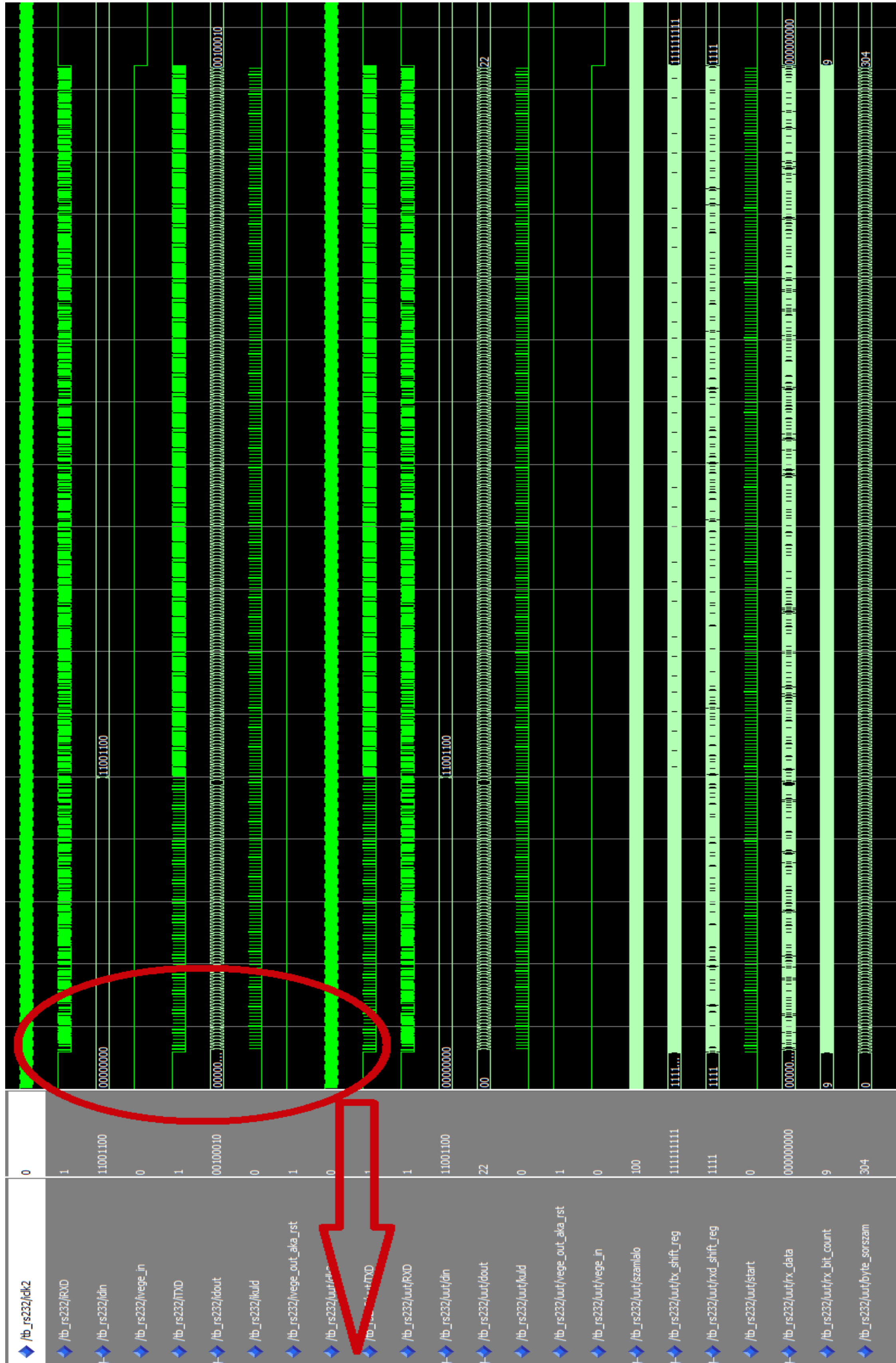
17.b. *ábra*: Az RS232-es modul testbench-e (2. rész)

Az időzítés ebben az utolsó ciklusban kritikus, ugyanis az RS232-es modul úgy lett megtervezve, hogy 9600 baud sebességgel működjön, amely azt jelenti, hogy az RXD bemenetre 104 µs-onként kell új értéket adni. Ez a késleltetési idő az alábbi összefüggésből számítható: az órajel periódusideje 13 µs, az RS232 modul pedig 8 órajelperiódusig feltételez az RXD bemenetén egy jelet állandónak: ebből következik, hogy 13µs * 8 = 104µs, valamint 1/104e-6 ~ 9600 baud.

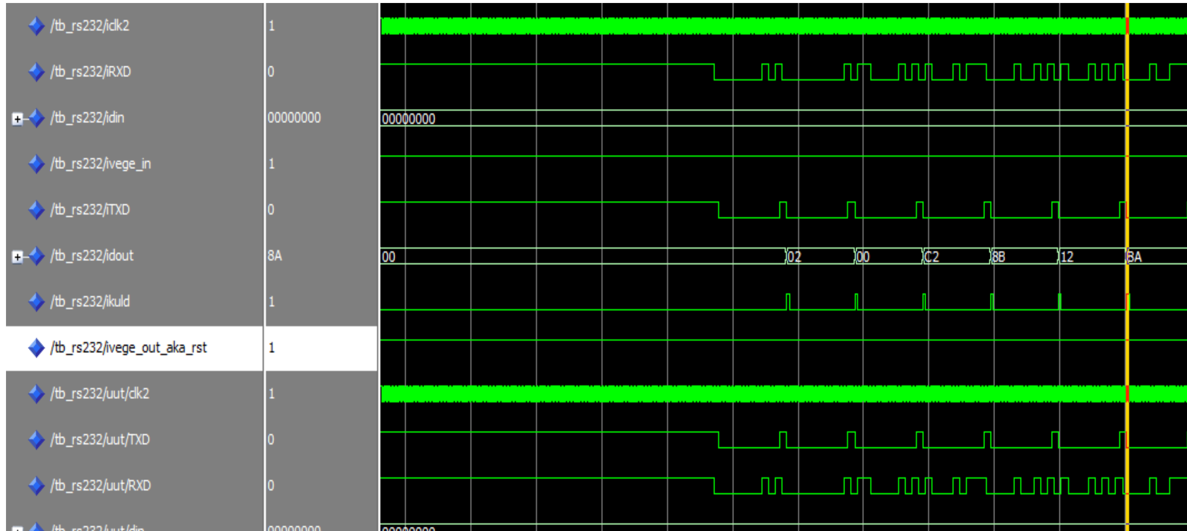
Mindezek után 18. és 19. ábrán láthatjuk az RS232-es modul szimulációs eredményét a fent tárgyalt tesztbench-csel meghajtva amely a alábbi fib.hex nevű file-t használta fel.

```
:100003008B128A1389148D1574011200FBAB12AA8B
:1000130013A914F58275830012010D751602AD162E
:10002300ED3395E0FCC3ED95157480F86C98503E64
:10003300AF16EF3395E0FEEF24FEFDEE34FFAB1277
:10004300AA13A9148D82F5831200CEFDEF24FFFFBE
:10005300EE34FF8F82F5831200CE2DFFE516FD33BC
:0F00630095E08D82F583EF12010D051680B02216
:100072008B128A1389148D15E4F516AD16ED33959E
:10008200E0FCC3ED95157480F86C985019AB12AA78
:1000920013A914AF16EF3395E08F82F5831200CEC9
:0700A200F580051680D52250
:1000A9007B007A0079087D0A1200037B007A0079C7
:0900B900087D0A12007280FE228B
:030000000200C239
:0C00C200787FE4F6D8FD7581160200A9D5
:1000CE00BB010CE58229F582E5833AF583E02250E7
:1000DE0006E92582F8E622BBFE06E92582F8E22231
:0D00EE00E58229F582E5833AF583E493224B
:1000FB00BB010689828A83F0225002F722BBFE01E4
:02010B00F322DD
:10010D00F8BB010DE58229F582E5833AF583E8F028
:10011D00225006E92582C8F622BBFE05E92582C8D4
:02012D00F222BC
:00000001FF
```

A 18. ábrán a teljes hosszúságú, 360 ms-os szimulációs látható, valamint az, hogy a modul a file beolvasását követően az *(i)vege_in* nevű portját lehúzza 0-ba amelyet majd a mikrokontrollert alkotó modulok számára fogunk tudni rst jelként használni. A 19-es ábrán ugyanaz a szimulációs eredmény látható kinagyítva,így arról is meggyőződhetünk, hogy a fenti hex file-t helyesen olvasta ki. A fenti hex file alapján az első 5 memóricímen az alábbi hexadecimális értékeket várjuk: 02, 00, C2, 8B, 12, és a 19. ábrán látható dout kimeneten ugyanezeket az eredményeket is kaptuk.



18. *ábra*: Az RS232-es modul működésének szimulációs eredménye

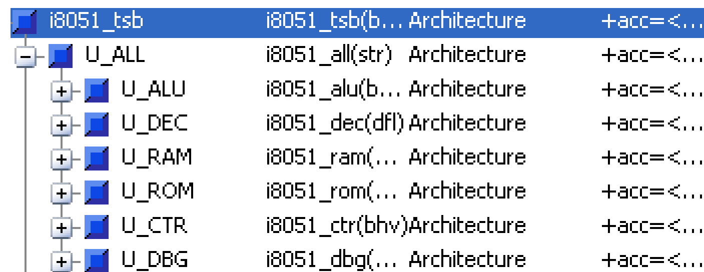


19. *ábra*: Az RS232-es modul szimulációjának zoomolt eredménye

Összefoglalva az eddigieket, kifejlesztésre került egy olyan RS232 nevű modul, valamint egy hozzá tartozó testbench, amelyekkel képesek vagyunk szimulálni egy Intel hex típusú file-ból történő olvasást, majd ezen hex file tartalmát címük szerint sorba tudjuk rendezni a testbench modulunk segítségével. Mindezt annak érdekében tettük, hogy a következő lépésben e rendezett file-tartalmat bitenként és helyes időzítésekkel az RS232-es modulunk RXD bemenetére tudjuk küldeni, hogy a modul majd a testbenchből küldött „bitstream"-ből vissza tudja állítani az eredeti hex file tartalmát és azt egy, az RS232 mellé implemetált memóriában el tudja menteni. Miután megbizonyosodtunk arról, hogy ezen modulunk helyesen működik, végső lépésként hozzá kell illesztenünk a modult és a testbenchet a 8051-es mikrokontroller VHDL modelljét alkotó moduljaihoz, valamint azok testbench moduljához.

3.5. Az RS232 modul illesztése a mikrokontroller VHDL modelljéhez

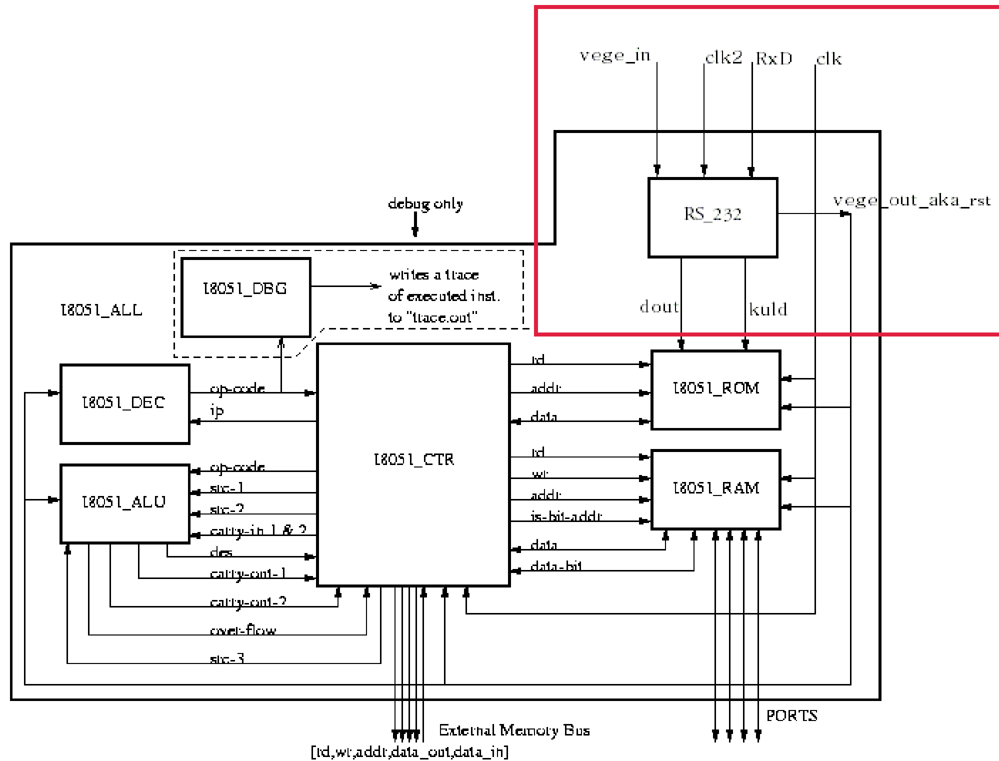
A 8051-es mikrokontroller VHDL modelljének rövid bemutatására már sor került a 2.3.2.-es pontban. Ugyanúgy látható a 13. ábrán a 8051-es mikrokontroller blokkdiagramja. A mikrokontroller modell hierarchikus felépítését a 20. ábra mutatja.



20. *ábra*: a mikrokontroller modell hierarchikus felépítése

Itt jól látható, hogy a mikrokontroller modell egyes moduljait egy U_ALL nevű top module kapcsolja össze, ahol a többi modul azonos hierarchia szinten van. Terveink szerint az illesztést úgy valósítjuk meg, hogy az RS232-es modult az „almodulok” mellé illesztjük és az új modul a többi modullal történő összehuzalozását at U_ALL top module módosításával hajtjuk végre. A 13. ábrával összehasonlítva jól mutatja célunkat a 21. ábra. A képen a mikrokontroller blokkdiagramja látható, kiegészítve a pirossal bekeretezett résszel, amely az újonnan a mikrokontrollerhez illesztett RS232-es modult ábrázolja.

Ahogyan már az előző fejezetben is szó volt róla az RS232-es modul 6 db interfészét használjuk az illesztés folyamán. Ennek fele-fele ki- és bemenet. A bemenetek mind kívülről vezérelhetők, a testbenchen keresztül adunk nekik értéket. Az RS232 3 kimenete alapvetően a ROM modulhoz csatlakozik, a *dout* kimeneten kapja a ROM byte-onként a végrehajtandó programot, vagyis a testbench által beolvasott hex file tartalmát. A *kuld* interfészen keresztül „értesíti” az RS232-es modul a ROM modult, hogy egy új byte került a *dout* portra. Végül az egyik legfontosabb a *vege_out_aka_rst* jel, amely akkor lesz 0, ha a vege_in jel 0, és ez a jel helyettesíti majd a mikrokontroller összes moduljának reset-jét vagyis ezen jel eltűnésének hatására kezd el működni a kontroller.



21. *ábra*: A mikrokontroller blokkdiagramja az RS232 modul illesztése után

A mikrokontroller és az illesztett modul együttes modelljének működésének stratégiája az, hogy a teljes modell bekapcsolása után a mikrokontroller összes modulja - a ROM modul bizonyos áramköreit kivéve - resetben van, vagyis arra vár, hogy „elindulhassan", az RS232-es viszont már a kezdetektől fogva dolgozik, és tölti föl a *dout* kimenetén keresztül a 8051-es ROM modulban deklarált memóriát a későbbiekben végrehajtandó programmal. Amint ez a file-feltöltési folyamat befejeződött, az RS232, a *vege_out_aka_rst* portjának kimenetét 0-ba állítja, aminek köszönhetően a mikrokontoller elkezd dolgozni, vagyis elkezdi végrehajtani az előbb az RS232 által a ROM-jába betöltött programot. (A ROM megnevezés itt félreérthető lehet, hiszen ezt a memóriát az RS232-vel pont az előbb írtuk, ám az a mikrokontroller szempontjából továbbra is csak olvasható memóriaterület, így helytálló ezen a ponton a ROM megnevezés.)

Az illesztés érdekében a mikrokontroller modell moduljai közül a ROM és az ALL modulokon kellett fejlesztéseket végeznem, továbbá az egész, újonnan kidolgozott modell szimulálhatóságának érdekében a mikrokontroller testbench moduljában is. Az alábbiakban látható a legtöbb fejlesztésen átesett ROM modell:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use WORK.I8051_LIB.all;

entity I8051_ROM is
    port(clk        : in  STD_LOGIC;
         addr       : in  UNSIGNED (11 downto 0);
         data       : out UNSIGNED (7 downto 0);
         rd         : in  STD_LOGIC;
         RS232_dout : in STD_LOGIC_VECTOR (7 downto 0);
         RS232_kuld : in STD_LOGIC;
         RS232_vege_out_aka_rst : in STD_LOGIC
         );
end I8051_ROM;

architecture BHV of I8051_ROM is

type ROM_TYPE is array (0 to 4095) of UNSIGNED (7 downto 0);
signal cim   : integer := 0;

begin
    process(RS232_vege_out_aka_rst, clk, RS232_kuld)

    variable PROGRAM : ROM_TYPE;
    variable ROM_nullaz: boolean := TRUE;

    begin

      if (ROM_nullaz = TRUE) then
            ROM_nullaz := FALSE;
            for i in 0 to 4095 loop         -- kinullázzuk a
ROM-ot
                PROGRAM(i) := "00000000";
            end loop;
      end if;

      if ((RS232_kuld = '1' ) and rising_edge(RS232_kuld)) then
          PROGRAM(cim) := UNSIGNED(RS232_dout);
```

```
          cim <= cim + 1;
     end if;

       if( RS232_vege_out_aka_rst = '1' ) then
          data <= CD_8;
        elsif( clk'event and clk = '1' ) then

              if( rd = '1' ) then

                data <= PROGRAM(conv_integer(addr));
            else
                data <= CD_8;
            end if;
          end if;
       end process;
     end BHV;
```

A mikrokontroller ROM nevű moduljában, amelyben eddig a program egy konstansként volt definiálva, létrehoztunk egy új, az eddigi memóriával megegyező méretű változót, amelybe az RS232-es modulon keresztül kapott új pogramot elmentjük. Hogy ezt meg tudjuk tenni a modul 2 porttal bővült, valamint az eddigi *rst* bemenetnek a helyét átvette az *RS232_vege_out_aka_rst* nevű jel, amelynek funkciója azonban a ROM modul szempontjából ugyanaz maradt. Mindkét újonnan implementált bemenet az RS232-höz csatlakoztatja a modult, az *RS232_dout*-on keresztül fogadja az RS232 által dekódolt és 8 bites formátumba hozott, a mikrokontroller indulása után végrehajtandó programot. Minden újonnan érkező byte elérhetőségét, vagyis, hogy az ki lett adva az *RS232_dout* bemenetre az *RS232 _kuld* bemeneti jellel ütemezzük.

A mikrokontroller modell másik modulja – amit módosítani kellett – az ALL nevű top module.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use WORK.I8051_LIB.all;

entity I8051_ALL is
    port(clk          : in  STD_LOGIC;
         xrm_addr     : out UNSIGNED (15 downto 0);
         xrm_out_data : out UNSIGNED (7 downto 0);
         xrm_in_data  : in  UNSIGNED (7 downto 0);
         xrm_rd       : out STD_LOGIC;
```

```
        xrm_wr        : out STD_LOGIC;
        p0_in         : in  UNSIGNED (7 downto 0);
        p0_out        : out UNSIGNED (7 downto 0);
        p1_in         : in  UNSIGNED (7 downto 0);
        p1_out        : out UNSIGNED (7 downto 0);
        p2_in         : in  UNSIGNED (7 downto 0);
        p2_out        : out UNSIGNED (7 downto 0);
        p3_in         : in  UNSIGNED (7 downto 0);
        p3_out        : out UNSIGNED (7 downto 0);
        rs232_clk2    : in  STD_LOGIC;
        rs232_RXD     : in  STD_LOGIC;
        rs232_vege_in: in  STD_LOGIC);

end I8051_ALL;

architecture STR of I8051_ALL is

    component I8051_ALU
        port(rst     : in  STD_LOGIC;
             op_code : in  UNSIGNED (3 downto 0);
             src_1   : in  UNSIGNED (7 downto 0);
             src_2   : in  UNSIGNED (7 downto 0);
             src_3   : in  UNSIGNED (7 downto 0);
             src_cy  : in  STD_LOGIC;
             src_ac  : in  STD_LOGIC;
             des_1   : out UNSIGNED (7 downto 0);
             des_2   : out UNSIGNED (7 downto 0);
             des_cy  : out STD_LOGIC;
             des_ac  : out STD_LOGIC;
             des_ov  : out STD_LOGIC);
    end component;

    component I8051_DEC
        port(rst     : in  STD_LOGIC;
             op_in   : in  UNSIGNED (7 downto 0);
             op_out  : out UNSIGNED (8 downto 0));
    end component;

    component I8051_RAM
        port(rst          : in  STD_LOGIC;
             clk          : in  STD_LOGIC;
             addr         : in  UNSIGNED (7 downto 0);
             in_data      : in  UNSIGNED (7 downto 0);
             out_data     : out UNSIGNED (7 downto 0);
             in_bit_data  : in  STD_LOGIC;
             out_bit_data : out STD_LOGIC;
```

```
            rd           : in  STD_LOGIC;
            wr           : in  STD_LOGIC;
            is_bit_addr  : in  STD_LOGIC;
            p0_in        : in  UNSIGNED (7 downto 0);
            p0_out       : out UNSIGNED (7 downto 0);
            p1_in        : in  UNSIGNED (7 downto 0);
            p1_out       : out UNSIGNED (7 downto 0);
            p2_in        : in  UNSIGNED (7 downto 0);
            p2_out       : out UNSIGNED (7 downto 0);
            p3_in        : in  UNSIGNED (7 downto 0);
            p3_out       : out UNSIGNED (7 downto 0));
    end component;

    component I8051_ROM
        port(clk      : in  STD_LOGIC;
             addr     : in  UNSIGNED (11 downto 0);
             data     : out UNSIGNED (7 downto 0);
             rd       : in  STD_LOGIC;
             RS232_dout : in STD_LOGIC_VECTOR (7 downto
0);
             RS232_kuld : in STD_LOGIC;
             RS232_vege_out_aka_clr : in std_logic
             );
    end component;

  -- RS232 COMPONENT --
    component RS232IO
        port ( clk2  : in  STD_LOGIC;
               TXD  : out STD_LOGIC ;
               RXD  : in  STD_LOGIC;
               din  : in  STD_LOGIC_VECTOR (7 downto 0);
               dout : out STD_LOGIC_VECTOR (7 downto 0) :=
"00000000";
               kuld : out std_logic := '0';
               vege_out_aka_clr : out std_logic := '1';
               vege_in          : in std_logic);
    end component;
  ----

    component I8051_CTR
        port(rst              : in  STD_LOGIC;
             clk              : in  STD_LOGIC;
             rom_addr         : out UNSIGNED (11 downto
0);
             rom_data         : in  UNSIGNED (7 downto 0);
             rom_rd           : out STD_LOGIC;
```

```
            ram_addr          : out UNSIGNED (7 downto 0);
            ram_out_data      : out UNSIGNED (7 downto 0);
            ram_in_data       : in  UNSIGNED (7 downto 0);
            ram_out_bit_data : out STD_LOGIC;
            ram_in_bit_data  : in  STD_LOGIC;
            ram_rd            : out STD_LOGIC;
            ram_wr            : out STD_LOGIC;
            ram_is_bit_addr  : out STD_LOGIC;
            xrm_addr          : out UNSIGNED (15 downto
0);
            xrm_out_data      : out UNSIGNED (7 downto 0);
            xrm_in_data       : in  UNSIGNED (7 downto 0);
            xrm_rd            : out STD_LOGIC;
            xrm_wr            : out STD_LOGIC;
            dec_op_out        : out UNSIGNED (7 downto 0);
            dec_op_in         : in  UNSIGNED (8 downto 0);
            alu_op_code       : out UNSIGNED (3 downto 0);
            alu_src_1         : out UNSIGNED (7 downto 0);
            alu_src_2         : out UNSIGNED (7 downto 0);
            alu_src_3         : out UNSIGNED (7 downto 0);
            alu_src_cy        : out STD_LOGIC;
            alu_src_ac        : out STD_LOGIC;
            alu_des_1         : in  UNSIGNED (7 downto 0);
            alu_des_2         : in  UNSIGNED (7 downto 0);
            alu_des_cy        : in  STD_LOGIC;
            alu_des_ac        : in  STD_LOGIC;
            alu_des_ov        : in  STD_LOGIC);
    end component;

    -- synopsys_synthesis off
    component I8051_DBG
        port(op_in    : in  UNSIGNED (8 downto 0));
    end component;
    -- synopsys_synthesis on

    signal rom_addr         : UNSIGNED (11 downto 0);
    signal rom_data         : UNSIGNED (7 downto 0);
    signal rom_rd           : STD_LOGIC;
    signal ram_addr         : UNSIGNED (7 downto 0);
    signal ram_out_data     : UNSIGNED (7 downto 0);
    signal ram_in_data      : UNSIGNED (7 downto 0);
    signal ram_out_bit_data : STD_LOGIC;
    signal ram_in_bit_data  : STD_LOGIC;
    signal ram_rd           : STD_LOGIC;
    signal ram_wr           : STD_LOGIC;
    signal ram_is_bit_addr  : STD_LOGIC;
```

```
    signal dec_op_out       : UNSIGNED (7 downto 0);
    signal dec_op_in        : UNSIGNED (8 downto 0);
    signal alu_op_code      : UNSIGNED (3 downto 0);
    signal alu_src_1        : UNSIGNED (7 downto 0);
    signal alu_src_2        : UNSIGNED (7 downto 0);
    signal alu_src_3        : UNSIGNED (7 downto 0);
    signal alu_src_cy       : STD_LOGIC;
    signal alu_src_ac       : STD_LOGIC;
    signal alu_des_1        : UNSIGNED (7 downto 0);
    signal alu_des_2        : UNSIGNED (7 downto 0);
    signal alu_des_cy       : STD_LOGIC;
    signal alu_des_ac       : STD_LOGIC;
    signal alu_des_ov       : STD_LOGIC;

-- Send parameter
    signal RS232_TXD        : STD_LOGIC;   -- Send output
    signal RS232_din        : STD_LOGIC_VECTOR (7 downto
0):= "00000000"; -- Send input
-- Receive parameter
    signal RS232_dout       : STD_LOGIC_VECTOR (7 downto
0);
    signal RS232_kuld       : STD_LOGIC;
    signal RS232_vege_out_aka_clr : std_logic;

begin

    U_RS232 : RS232IO port map (RS232_clk2,
                                RS232_TXD,
                                RS232_RXD,
                                RS232_din,
                                RS232_dout,
                                RS232_kuld,

                                RS232_vege_out_aka_clr,
                                RS232_vege_in);

    U_ALU : I8051_ALU port map(RS232_vege_out_aka_clr,
                               alu_op_code,
                               alu_src_1,
                               alu_src_2,
                               alu_src_3,
                               alu_src_cy,
                               alu_src_ac,
                               alu_des_1,
                               alu_des_2,
                               alu_des_cy,
```

```
                                alu_des_ac,
                                alu_des_ov);

U_DEC : I8051_DEC port map(RS232_vege_out_aka_clr,
                                dec_op_out,
                                dec_op_in);

U_RAM : I8051_RAM port map(RS232_vege_out_aka_clr,
                                clk,
                                ram_addr,
                                ram_in_data,
                                ram_out_data,
                                ram_in_bit_data,
                                ram_out_bit_data,
                                ram_rd,
                                ram_wr,
                                ram_is_bit_addr,
                                p0_in,
                                p0_out,
                                p1_in,
                                p1_out,
                                p2_in,
                                p2_out,
                                p3_in,
                                p3_out);

U_ROM : I8051_ROM port map(clk,
                                rom_addr,
                                rom_data,
                                rom_rd,
                                RS232_dout,
                                RS232_kuld,

                                RS232_vege_out_aka_clr
                                );

U_CTR : I8051_CTR port map(RS232_vege_out_aka_clr,
                                clk,
                                rom_addr,
                                rom_data,
                                rom_rd,
                                ram_addr,
                                ram_in_data,
                                ram_out_data,
                                ram_in_bit_data,
                                ram_out_bit_data,
```

```
                                             ram_rd,
                                             ram_wr,
                                             ram_is_bit_addr,
                                             xrm_addr,
                                             xrm_out_data,
                                             xrm_in_data,
                                             xrm_rd,
                                             xrm_wr,
                                             dec_op_out,
                                             dec_op_in,
                                             alu_op_code,
                                             alu_src_1,
                                             alu_src_2,
                                             alu_src_3,
                                             alu_src_cy,
                                             alu_src_ac,
                                             alu_des_1,
                                             alu_des_2,
                                             alu_des_cy,
                                             alu_des_ac,
                                             alu_des_ov);

         -- synopsys_synthesis off
         U_DBG : I8051_DBG port map(dec_op_in);
         -- synopsys_synthesis on
end STR;
```

Itt a többi modul példányosítása mellett példányosítani kellett az RS232-es modult is. Ezenkívül az eddigi, külső *rst* bemenet helyett a *vege_out_aka_rst* belső *rst* jelet kell a modulok bemenetére kötni.

Végül az új modell egészét együtt szimuláló testbenchet mutatom be.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;

entity I8051_TSB is
end I8051_TSB;

architecture BHV of I8051_TSB is

    component I8051_ALL
        port(clk          : in  STD_LOGIC;
             xrm_addr     : out UNSIGNED (15 downto 0);
```

```
            xrm_out_data : out UNSIGNED (7 downto 0);
            xrm_in_data  : in  UNSIGNED (7 downto 0);
            xrm_rd       : out STD_LOGIC;
            xrm_wr       : out STD_LOGIC;
            p0_in        : in  UNSIGNED (7 downto 0);
            p0_out       : out UNSIGNED (7 downto 0);
            p1_in        : in  UNSIGNED (7 downto 0);
            p1_out       : out UNSIGNED (7 downto 0);
            p2_in        : in  UNSIGNED (7 downto 0);
            p2_out       : out UNSIGNED (7 downto 0);
            p3_in        : in  UNSIGNED (7 downto 0);
            p3_out       : out UNSIGNED (7 downto 0);
            RS232_clk2   : in STD_LOGIC;
            RS232_RXD    : in STD_LOGIC;
            RS232_vege_in: in STD_LOGIC);
    end component;

 type ROM_TYPE is array (0 to 4095) of UNSIGNED (7 downto 0);

     signal RS232_clk2 : std_logic := '0';
     signal RS232_RXD  : std_logic := '1';
     signal RS232_vege_in : std_logic := '1';
     signal clk      : STD_LOGIC := '0';
     signal addr     : UNSIGNED (15 downto 0);
     signal in_data  : UNSIGNED (7 downto 0);
     signal out_data : UNSIGNED (7 downto 0);
     signal rd       : STD_LOGIC;
     signal wr       : STD_LOGIC;
     signal p0_in    : UNSIGNED (7 downto 0);
     signal p0_out   : UNSIGNED (7 downto 0);
     signal p1_in    : UNSIGNED (7 downto 0);
     signal p1_out   : UNSIGNED (7 downto 0);
     signal p2_in    : UNSIGNED (7 downto 0);
     signal p2_out   : UNSIGNED (7 downto 0);
     signal p3_in    : UNSIGNED (7 downto 0);
     signal p3_out   : UNSIGNED (7 downto 0);

function String_To_Integer(Some_Char: character) return integer
     is variable Return_Value: integer := 0;
begin
  case Some_Char is
     when '0' => Return_Value := 0;
     when '1' => Return_Value := 1;
     when '2' => Return_Value := 2;
     when '3' => Return_Value := 3;
     when '4' => Return_Value := 4;
```

```
      when '5' => Return_Value := 5;
      when '6' => Return_Value := 6;
      when '7' => Return_Value := 7;
      when '8' => Return_Value := 8;
      when '9' => Return_Value := 9;
      when 'A' => Return_Value := 10;
      when 'B' => Return_Value := 11;
      when 'C' => Return_Value := 12;
      when 'D' => Return_Value := 13;
      when 'E' => Return_Value := 14;
      when 'F' => Return_Value := 15;
      when others => null;
   end case;
return(Return_Value);
end function String_To_Integer;


function String_To_Unsigned(Some_Char: character) return
unsigned is variable Return_Value: unsigned  (3 downto 0);
      begin
        case Some_Char is
          when '0' => Return_Value := "0000";
          when '1' => Return_Value := "0001";
          when '2' => Return_Value := "0010";
          when '3' => Return_Value := "0011";
          when '4' => Return_Value := "0100";
          when '5' => Return_Value := "0101";
          when '6' => Return_Value := "0110";
          when '7' => Return_Value := "0111";
          when '8' => Return_Value := "1000";
          when '9' => Return_Value := "1001";
          when 'A' => Return_Value := "1010";
          when 'B' => Return_Value := "1011";
          when 'C' => Return_Value := "1100";
          when 'D' => Return_Value := "1101";
          when 'E' => Return_Value := "1110";
          when 'F' => Return_Value := "1111";
          when others => null;
       end case;
return(Return_Value);
end function String_To_Unsigned;

begin

    clk <= not clk after 25 ns;
    RS232_clk2<= not RS232_clk2 after 6500 ns;
```

```
U_ALL : I8051_ALL port map (clk,
                            addr, out_data, in_data, rd, wr,
                            p0_in, p0_out, p1_in, p1_out,
                            p2_in, p2_out, p3_in, p3_out,
                          RS232_clk2, RS232_RXD, RS232_vege_in);
     process

             type characterFile is file of character;
             file                cFile: characterFile;
             variable  c              : character;
             variable  Fstatus        : FILE_OPEN_STATUS;
             variable  char_count     : integer := 0;
             variable  hossz0         : integer := 0;
             variable  hossz1         : integer := 0;
             variable  hossz_full     : integer := 0;
             variable  kezdocim0      : integer := 0;
             variable  kezdocim1      : integer := 0;
             variable  kezdocim2      : integer := 0;
             variable  kezdocim3      : integer := 0;
             variable  kezdocim_full  : integer := 0;
             variable  tipus0         : integer := 0;
             variable  tipus1         : integer := 0;
             variable  tipus_full     : integer := 0;
             variable  adat0          : UNSIGNED (3 downto 0);
             variable  adat1          : UNSIGNED (3 downto 0);
             variable  offset         : integer := 0;
             variable  ROM_offset     : integer := 0;
             variable  send_byte      : unsigned(7 downto 0);
             variable  ROM_nullaz     : boolean := TRUE;
             variable  decode_vege    : boolean := TRUE;
             variable  file_read      : boolean := TRUE;
             variable  PROGRAM        : ROM_TYPE;
             variable max_addr        :integer:=0;
             variable max_addr_e      :integer:=0;

            begin --process beginje


           if (ROM_nullaz = true) then
                 ROM_nullaz := false;
                 for i in 0 to 4095 loop         -- kinullázzuk
     a ROM-ot
                     PROGRAM(i) := "00000000";
                 end loop;
           end if;
```

```
      file_open(Fstatus, cFile,
"C:\Users\Balint\Desktop\VHDL_8051_kieg\fib.hex",
read_mode);        --***

      if file_read then

      while (not endfile(cfile)) loop

            wait until RS232_clk2 = '1';
            read(cfile, c);


            if (c = ':') then
                  offset := 0;
                  ROM_offset := 0;
                  char_count:= 1;
                  tipus_full := 0;
                  next;
            end if;


            if (tipus_full /= 0) then
                  next;
            end if;

            char_count := char_count + 1;

            if (char_count = 2) then
                    hossz1 := string_to_integer(c);
            end if;

            if (char_count = 3) then
                    hossz0 := string_to_integer(c);
                    hossz_full := hossz1*16 + hossz0;
            end if;

            if (char_count = 4) then
                    kezdocim3 := string_to_integer(c);
            end if;

            if (char_count = 5) then
                    kezdocim2 := string_to_integer(c);
            end if;

            if (char_count = 6) then
```

```
                    kezdocim1 := string_to_integer(c);
            end if;

            if (char_count = 7) then
                    kezdocim0 := string_to_integer(c);
                    kezdocim_full := kezdocim3*4096 +
kezdocim2*256 + kezdocim1*16 + kezdocim0;
            end if;

            if (char_count = 8) then
                    tipus1 := string_to_integer(c);
            end if;

            if (char_count = 9) then
                    tipus0 := string_to_integer(c);
                    tipus_full := tipus1*16 + tipus0;
            end if;

            if (char_count >= 10 + 2*hossz_full) then
              next;
            end if;

            if (char_count = 10 + offset) then
                    adat1 := string_to_unsigned(c);
            end if;

            if (char_count = 11 + offset) then
                    adat0 := string_to_unsigned(c);
                    PROGRAM(kezdocim_full + ROM_offset) :=
adat1&adat0;
                    offset := offset + 2;
                    ROM_offset := ROM_offset + 1;
                    max_addr:=kezdocim_full + ROM_offset;

                    if max_addr_e < max_addr
                    then
                      max_addr_e:=max_addr;
                    end if;
            end if;


      end loop;
      file_close(cfile);

      file_read := FALSE;
    end if;
```

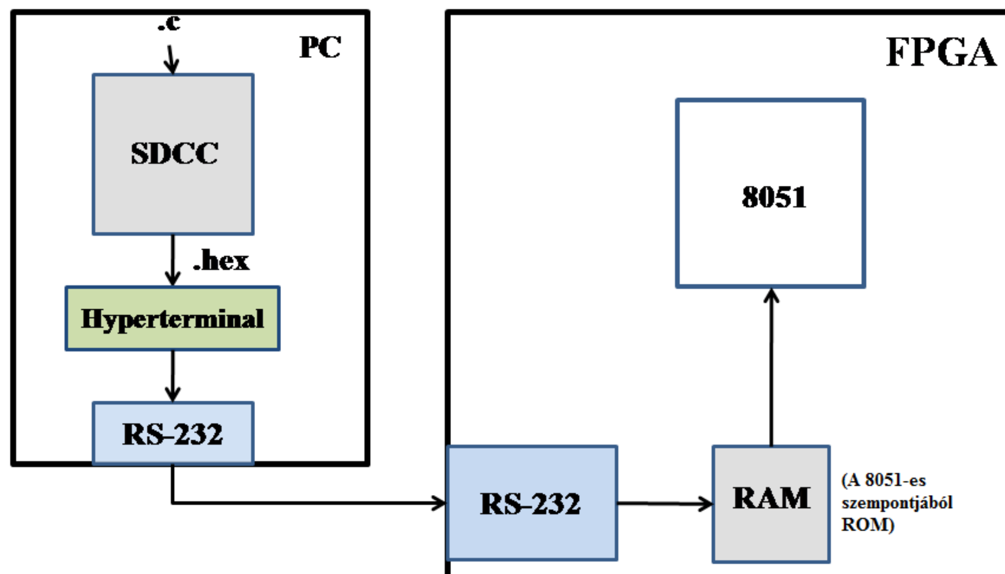
```
        if (decode_vege = TRUE) then
            for i in 0 to max_addr loop
            send_byte := PROGRAM(i);
            wait for 104 us;
                RS232_RXD <= '0'; -- startbit
            wait for 104 us;
                for j in 7 downto 0 loop
                    RS232_RXD <= send_byte(j);
                    wait for 104 us;
                end loop;
                RS232_RXD <= '1'; --stopbit;

                if (i = max_addr) then
                    RS232_vege_in <= '0';
                end if;

            end loop;
          decode_vege := FALSE;
          end if;
          wait for 104 us;
        end process;
  end BHV;

  configuration CFG_I8051_TSB of I8051_TSB is
      for BHV
      end for;
end CFG_I8051_TSB;
```

Ennél a modulnál visszaköszön a különálló, RS232-es modult tesztelő leírás, azzal a különbséggel, hogy az RS232-es mellett a mikrokontroller összes modulja szintén példányosítva van, valamint előállítjuk a kontrollert vezérlő bemeneti jeleket is.

4. Eredmények és értékelésük

Az előző fejezetekben bemutatásra került egy újonnan kifejlesztett RS232-es modul, valamint egy hozzá fejlesztett testbench modul, amelynek szimulációs eredményeit az adott fejezetben tárgyaltuk. A kész és szimulációval igazoltan helyes működésű RS232-es modul a 8051-es mikrokontroller VHDL modelljéhez történő illesztését is részletesen tárgyaltam az ezt megelőző fejezetben.

Az így kifejlesztett új mikrokontroller modellel képesek vagyunk egy C nyelven írt és egy cross-compilerrel a 8051-es mikrokontroller utasításkészletére lefordított programmal a 8051-es mikrokontrollerünket egy a µC mellé implementált RS232-es áramkör segítségével felprogramozni (22. ábra).



22. *ábra*: A 8051-es programozása RS232-es interfészen keresztül

Ezen új modell több fontos előnnyel is jár, az egyik és talán legfontosabb, hogy ilyen módon a mikrokontroller közvetlenül programozhatóvá válik magas szintű programnyelven. Egy másik előnye, hogy az újonnan fejlesztett, vagy egy módosított

program a mikrokontrolleren történő futtatásához nincs szükség az egész mikrokontroller modell újraszintetizálására. Végül, de nem utolsó sorban a mikrokontroller modell (elvben, még nincsen tesztelve) IP-coreként implementálható egy programozható kapuáramkörre, ahol képes lehet a fejlesztések „gyorsan futtathatóságának" köszönhetően C programok 8051-es mikrokontrollerre történő fejlesztését elősegíteni.

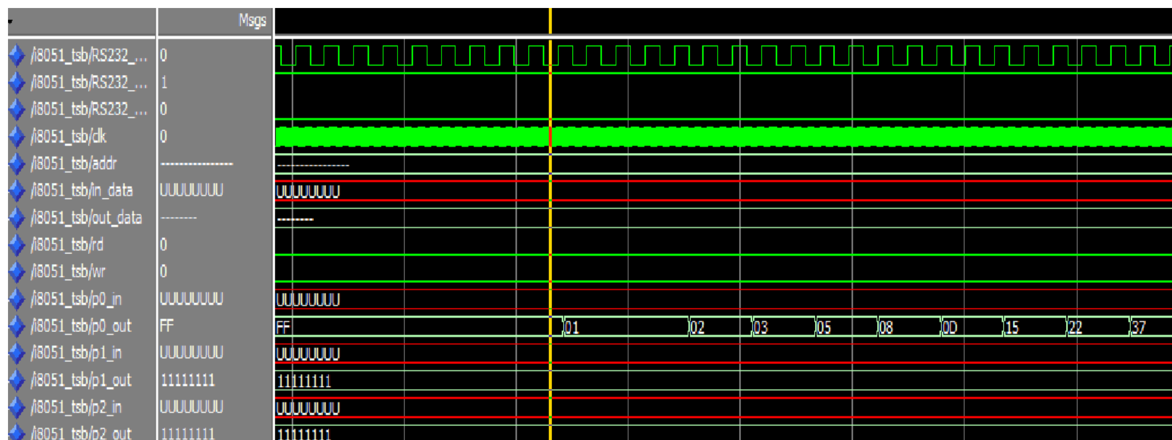
Az alábbiakban bemutatjuk szimulálva egy a 22. ábrán látható fejlesztési folyamatot, kiindulva egy egyszerű C programból. Legyen a rövid C programunk egy Fibonacci számsor 10 elemét előállító C program az alábbi forráskóddal.

```
#include <8051.h>

void fib(unsigned char* buf, unsigned char n) {

    char i;

    buf[0] = 1;
    buf[1] = 1;
    for(i=2; i<n; i++) {

        buf[i] = buf[i-1] + buf[i-2];
    }
}

void print(unsigned char* buf, unsigned char n) {

    char i;

    for(i=0; i<n; i++) {

        P0 = buf[i];
    }
}
void main() {

    unsigned char buf[10];

    fib(buf, 10);
    print(buf, 10);
    while(1);
}
```

Ezt a kódot az SDCC cross-compilerrel lefordítva egy fib.hex nevű Intel hex file-t kapunk. A hex file tartalmát az előző fejezetben a 40. oldalon az RS232 tesztelésekor már láthattuk. A következőkben a 23. és a 24. ábrán látható szimulációval mutatjuk be ennek a programnak az új processzor modell általi beolvasását és végrehajtását. A 23.

ábrán a teljes szimuláció megfigyelhető. Ennek első része a mikrokontroller memóriájának a végrehajtandó programmal történő feltöltése, míg a második része magának a programnak a végrehajtása.

23. *ábra*: A mikrokontroller programmal történő feltöltésének és annak végrehajtásának szimulációja



24. *ábra*: A fib.c program mikrokontrolleren történő futásának szimulált eredménye

Mint látható, a kidolgozott mikrokontroller modellre feltöltött lefordított C program futási eredményeként a kontroller helyesen – a programnak megfelelően – a P0 kimenetére írja ki az első 10 Fibonacci számot.

5. Összefoglalás

A hardver és szoftver együttes tervezésnek, valamint a hardver-compilerek szakirodalmának áttanulmányozására alapulva a kutatás célja egy olyan processzor modell kifejlesztése volt, amellyel képesek vagyunk algoritmusokat egy célhardver számára közvetlenül elérhetővé és értelmezhetővé tenni. Ennek érdekében egyrészt egy a szakirodalomban megtalálható C compilert használtam fel, másrészről pedig kifejlesztettem VHDL nyelven egy olyan modult, amely egy 8051-es mikrokontroller egy adott VHDL modelljéhez illesztve képessé teszi annak közvetlenül magas szintű programnyelven történő programozását. A mikrokontrolleren az újonnan kifejlesztett RS232-es modul hozzá történő illesztése során több, a működést érintő fejlesztés történt, aminek eredményképpen létrejött új mikrokontroller modell egy univerzálisan felhasználható IP core-nak tekinthető és elvben akár ASIC, akár FPGA áramkörön megvalósítható.

A fejlesztésnél végig szem előtt tartottam a modellben rejlő további lehetőséget, így az RS232-ben a küldés funkció a jövőre való tekintettel lett implementálva. Ezt a funkciót használva lehetővé válhatna, hogy a mikrokontroller a számítási eredményeit egy olyan memóriaterületre írja, ahonnan ahhoz az RS232 segítségével kívülről hozzá tudunk férni. Ezen felül egy olyan, további kutatásokat igénylő fejlesztés alapja is lehet az itt bemutatott modell, amelyben a mikrokontroller univerzalitását feláldozzuk egy adott feladatra optimalizált célhardver elkészítésének érdekében, amelyben figyeljük, hogy egy adott feladatot megoldó szoftver a hozzá tartozó hardver mely erőforrásait használja, így ennek ismeretében egy, a korábbinál kisebb erőforrásigényű, optimálisabb, és ugyanakkor gyorsabb működést lehetővé tevő hardver-szoftver együttes modellt tudunk előállítani.

6. Irodalomjegyzék

[1] IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-1993). New York, Institute of Electrical and Electronics Engineers, Inc., 1994.

[2] VHDL alapú rendszertervezés, Dr. Hosszú Gábor – Keresztes Péter, Budapest, 2009.

[3] MCS® 51 Microcontroller Family User's Manual, February 1994, Publication number 121517, Intel Corporation

[4] Dalton Project, University of California, Dept. of Computer Sciences, Riverside, http://www.cs.ucr.edu/~dalton/i8051/i8051syn/

[5] Target Architecture for HW/SW systems, Christian Plessl, Universität Paderborn, 2010.

[6] HW/SW Codesign, Micaela Serra, University of Victoria, Department of Computer Science, 2009.

[7] Anatomy of a Hardware Compiler, Kurt Keutzer, Wayne Wolf, AT&T Bell Laboratories, Murray Hill (NJ), 1988.

[8] Hardware Compilation: Translating Programs into Circuits, Nikolaus Wirth, Swiss Federal Institute of Technology, Zürich, 1998.

[9] Fly – A Modifiable Hardware Compiler, Ho, Leong, Lee, Dept. of Comp. Sc., The Chinese University of Hong Kong; Ludewig, Zipf, Ortiz, Glesner, Inst. of Microelectronic Systems, Darmstadt University of Technology