Teszt- és kényszerfájl generátor FPGA fejlesztéshez és ennek használata, jelentősége az oktatásban

Borsos Döníz[1], **Fintor József**[2], **Zsáry Gábor**[2]

Óbudai Egyetem Biztonságtudományi Doktori Iskola, Budapest[1]

Óbudai Egyetem Kandó Kálmán Villamosmérnöki Kar, Műszertechnikai és Automatizálási Intézet, Budapest[2]

borsos.doniz@kvk.uni-obuda.hu

fintor.j@stud.uni-obuda.hu

zsarygabor@stud.uni-obuda.hu

Absztrakt: Az FPGA-k iránti keresletben folyamatos növekedés figyelhető meg az elmúlt évek adatait tekintve. Ezt legjobban az mutatja, hogy az FPGA-k piaca 2014 és 2019 között megduplázódott, 5 milliárd USD-ről közel 10 milliárd USD-re nőtt, továbbá ez az érték 2024-re várhatóan eléri a 20 milliárd amerikai dollárt. Ennek következtében a fejlesztéstámogató szoftverek iránti igény is megnőtt. FPGA-ra való fejlesztés során két alapvető feladat a teszt-és kényszerfájl készítése, melyek esetében hatékonyan alkalmazhatók konfiguráló, generáló programok. Az ilyen alkalmazások nem csak fejlesztés során, hanem az oktatásban is segítséget nyújtanak. A tanulmány egy FPGA-khoz fejlesztett teszt- és kényszerfájl generátorral foglalkozik, emellett a program oktatásbeli aspektusait vizsgálja.

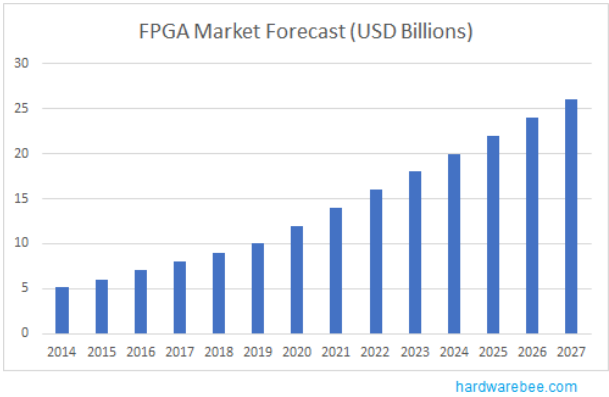
Kulcsszavak: FPGA, tesztfile, kényszerfile, szoftver, oktatás

Abstract: Demand for FPGAs has been growing steadily over the past few years. This is best illustrated by the fact that the FPGA market doubled between 2014 and 2019, growing from $5 billion to nearly $10 billion, and is expected to double again by 2024. As a result, the demand for the use of support software during the development process has also increased. A basic task during development is to create test and constraint files, for which configuration and generation programs can be used effectively. Such applications also help in education. The study deals with a test and constraint file generator developed for FPGAs and involves examining the use of the software in education. Keywords: FPGA, test file, constraint file, software, education

Keywords: FPGA, test file, constraint file, software, education

1 Bevezetés

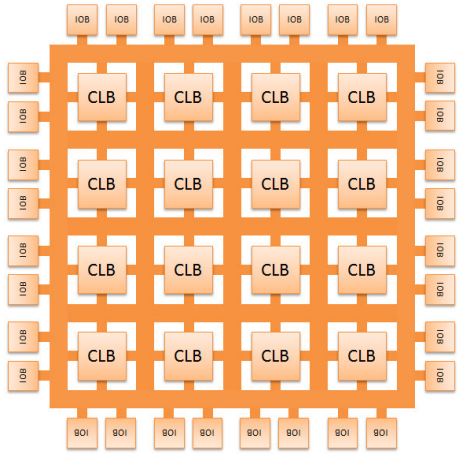
Tapasztalataink szerint a mérnök hallgatók közül meglepően kevesen találkoznak FPGA programozással, pedig az iparban nagy kereslet van FPGA programozókra, hiszen az FPGA-k iránti kereslet folyamatosan növekszik. Az FPGA-k piaca 2014 és 2019 között megduplázódott, 5 milliárd USD-ről közel 10 milliárd USD-re nőtt, továbbá ez az érték 2024-re várhatóan eléri a 20 milliárd amerikai dollárt [1].



Ábra 1
FPGA piac becsült növekedése 2014 és 2027 között[1]

De mi is az az FPGA? Olyan integrált áramkör, amely programozható logikai blokkokból és újra konfigurálható összeköttetésekből áll. Az általános logikai blokkok két fő részből állnak, az úgynevezett LUT-ból, mely segítségével kombinációs hálózatot tudunk létrehozni és Flip-Flopokból, mely szekvenciális hálózat realizálásában vesz részt. Az FPGA többfajta dedikált blokkokat is tartalmazhat, például input-, output blokkokat a külvilággal való kommunikációhoz, memória blokkokat adatok tárolásához, hardveres szorzó blokkokat, memóriavezérlőt külső memória illesztéséhez és analóg digitális átalakítókat is. Ezeket a blokkokat kapcsolja össze az újrakonfigurálható huzalozás [2].

[1]Forrás: https://hardwarebee.com/fpga-market-forecast-2014-2027/
Letöltés dátuma: 2020.11.15.



Ábra 2
FPGA elvi felépítése[2]

A konfigurálást úgynevezett hardver leíró nyelvel (HDL) lehet elvégezni. Akkor használunk ilyen megoldást, ha a nagysebességű párhuzamos műveletvégzés különösen fontos és gyorsan változó valós idejű jelekkel dolgozunk. Ilyen például, a digitális jelfeldolgozás, képfeldolgozás, mesterséges neurális hálózatok, hardvertervezés.

Az Óbudai Egyetem Kandó Kálmán Villamosmérnöki Karán a hallgatók megismerhetik az egyik vezető FPGA gyártó, a Xilinx, fejlesztői környezetét a Vivado Design Suite-et a Nexys 4 fejlesztői board használatával.

A Vivado fejlesztői környezet számos segítséget nyújt az egyszerű és gyors fejlesztéshez. A megírt programunk működését szimulálni tudjuk még mielőtt a hardveren kipróbálnánk. Ehhez szükséges egy úgynevezett tesztfájl készítése, melyben leírhatjuk a várt bemeneteket. Ha alaposan leteszteltük a programunkat a szimulátorban, akkor jöhet a valós környezetben való munka. A programunk be- és kimeneti vektorjainak hozzárendelése az FPGA lábaihoz egy úgynevezett kényszerfájlban történik. Ez a kényszerfájl formátum gyártótól függ. A mi esetünkben a Xilinx saját kényszerfájl formátumát használjuk.

Ez a két fájl nagyrészben ismétlődő és előre meghatározott elemeket tartalmaz. Ezek megírása monoton, hosszadalmas és könnyen hibázhatunk bennük, de automatikus generálásuk könnyen megvalósítható. Számos ilyen programot

[2]Forrás: http://moodle.autolab.uni-pannon.hu/Mecha_tananyag/autoipari_beagyazott_rendszerek/images/10000201000002430000023D37AC7087.png Letöltés dátuma: 2020.11.15.

találhatunk, böngészőből futtatható vagy letölthető formában, de használatuk során gyakran problémákba ütközhetünk.

Körbe kérdeztünk hallgatótársaink között[3] és a megkérdezettek közül, a túlnyomó többség, 93% használ valamilyen kód generáló alkalmazást, legyen az webes felületen használható vagy letöltött programként futtatható. Jelentős részüknek, 43%-uknak volt valamilyen problémája a generált kóddal, például hibás a kód vagy nem támogat bizonyos funkciókat. A kódgenerátorok elérhetőségéről is sokan panaszkodtak, a megkérdezettek 33%-a által használt szoftver nem érhető el többé.

2 FPGA_Tools fejlesztése és bemutatása

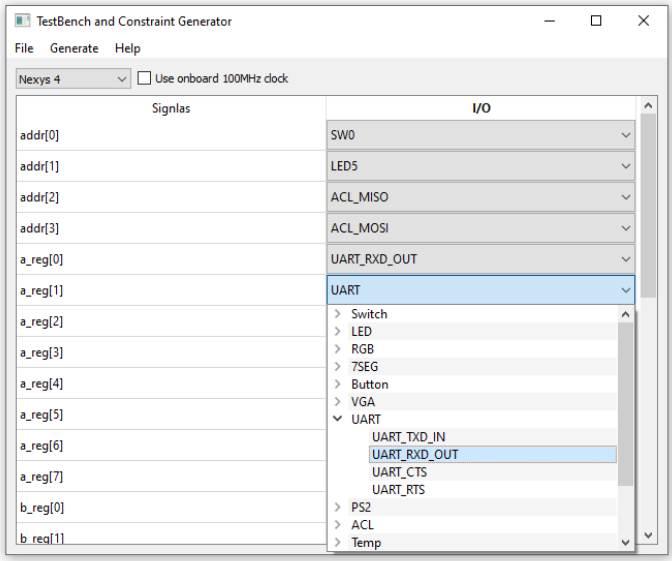
2.1 Python nyelv

Napjainkban a Python az egyik legkedveltebb nyelvnek mondható. Két független programnyelv-népszerűség elemző weboldal mérései is alátámasztják ezt [3] [4]. A nyelv népszerűsége főként abban rejlik, hogy könnyen átlátható, egyszerű megtanulni és szinte minden területen alkalmazható, legyen az akár beágyazott környezet, mesterséges intelligencia, szervezeti folyamat automatizálás, vagy big data elemzés. Ennek következtében, számtalan közösség által készített könyvtár és oktatóanyag található meg az interneten, melyek segítségével bárki könnyedén megismerkedhet a programnyelvvel. A profession.hu szoftverfejlesztői álláshirdetései nagy részében szükséges ismeretként tüntetik fel a Python nyelvet. A nagy igény miatt az egyetemeken is oktatási anyag részét képezi a nyelv, köztük a karunkon is nagy népszerűségnek örvendenek a nyelvvel kapcsolatos tárgyak. Az említett okok miatt, erre a nyelvre esett a választás.

2.2 Az alkalmazás működése és funkciói

A szoftver tervezése és fejlesztése során arra törekedtünk, hogy egyszerű és könnyen átlátható legyen a felhasználói felülete. Mivel elsősorban oktatási célból készült a program, ezért az internetkapcsolat nélküli számítógépes alkalmazást választottuk, mert az egyetemi órákon számonkérés során nincs lehetőség internet használatára és így webszervert sem kell fenntartani az üzemeltetéséhez.

[3] A hallgatókat online kérdőívben kérdeztük meg.



Ábra 3

Az FPGA_Tools alkalmazás működés közben

A szoftver egyik fő funkciója a tesztpad generálás, amelyhez először be kell tallózni a VHDL forrásfájlt. Ezt a File menüpont alatt, a Browse gombra kattintva az aktuális operációs rendszer fájlkezelőjének segítségével lehet megtenni. A választott fájl tallózása után, megjelenik a szoftverben az összes jel az adott modul portlistájából (3. ábra), melynek csak a kényszer generálásakor lesz szerepe. A generálást a Generate menüpont alatti TestBench gombbal lehet elvégezni és menteni a kívánt helyre. A generátor importálja a szükséges könyvtárakat, elkészíti az architektúrát, amelyben felsorolja a forrásfájlból kinyert portokat, létrehozza a lokális jeleket és elvégzi a portok, belső jelekhez való hozzárendelését, amelyek együttesen adják a tesztpad vázát.

A másik funkciója pedig, a kényszerfájl generálás. Lehetőség van a forrásfájl minden egyes jeléhez egy funkciót választani az adott fejlesztői panelhez tartozó funkciólistából, amelyek csoportba rendezve jelennek meg, segítve a felhasználót a navigálásban. A fejlesztőpanelek között a szoftver bal felső sarkában van lehetőség a váltásra, az ábrán a Nexys 4-es panel van kiválasztva. Egyelőre a szoftver csak az órán használt fejlesztőpaneleket támogatja, de lehetőség van saját funkció lista és lábkiosztás hozzáadására JSON file formájában, a szoftver projekt GitHub oldalán[4] leírt formátumban. Ha megfelelően lett hozzáadva a lista, akkor

[4] https://github.com/Fint0r/FPGA_Tools#custom-port-list-format

megjelenik a panelválasztóban. A megfelelő funkciók kiválasztása után a Generate menüpont alatt a Constraint gombbal lehet generálni és menteni a kényszerfájlt, a Vivado fejlesztői környezet által támogatott XDC fájlként.

2.3 Folyamatos fejlesztés, elérhetőség biztosítása

Manapság nagyon népszerű a nyílt forráskódú szoftvercsomagok készítése az ilyen jellegű fejlesztéstámogató programok körében, mert költséghatékony, könnyű terjeszteni és a közösség tagjai is bevonhatók a fejlesztésbe. Az egyik legismertebb forráskódmegosztó weboldal a GitHub, amely, a szoftverfejlesztésben leggyakrabban használt verziókezelőhöz, a Git-hez készült. A célunk elsősorban az volt, hogy bárki számára elérhető szoftvert készítsünk, ezért választottuk a legnépszerűbb programkód megosztó weboldalt, melynek segítségével bonyolítottuk le a probléma bejelentések kezelését, illetve a fejlesztők közötti együttműködést.

A szoftver elérhető a PyPI weboldalán[5] is, mint könyvtár, erre azért volt szükség, mert a program MIT licenccel van ellátva, mint nyílt forráskódú szoftver, ezért futtatható állomány generálása esetén az operációs rendszer letilthatja, mint nem biztonságos forrásból származó alkalmazás. Ezért, ha PyPI weboldaláról (pip-en keresztül) történik a letöltés, akkor natívan futtatható python-nal, így azok is tudják használni, akiknek az operációs rendszere letiltotta a futtatást.

3 FPGA_Tools szerepe az oktatásban

3.1 FPGA oktatás

A Kandó Kálmán Villamosmérnöki Karon a Műszertechnikai és Automatizálási Intézetben 2006. óta folyik FPGA oktatás az Automatizált gyártórendszerek II. laboratórium és az Információs rendszerek előadás keretein belül[6]. A hallgatók az Automatizált gyártórendszerek laboratóriumi gyakorlatok alatt megismerhetnek a hardverleírás alapjaival, a fejlesztés lépéseivel. Az Automatizált gyártórendszerek II. labor folytatásaként, a hallgatók az Információs rendszerek tárgy keretein belül mélyíthetik el ismereteiket az FPGA-k világában. Ezek a tantárgyak 6. és 7. félévben kerülnek meghirdetésre specializálódott hallgatók részére. Az oktatás során a fejlesztés VHDL nyelven folyik Xilinx Vivado környezetet és Nexys 4 Artix-7 FPGA Trainer Board-ot használva (4. ábra).

[5] https://pypi.org/project/fpgatools/

[6] A hallgatók az említett két tantárgy mellett még Digitális technikából, Beágyazott rendszerekből és Jelfeldolgozásból is találkozhatnak FPGA alapokkal.

Ábra 4
Nexys 4 Artix-7 FPGA Trainer Board[7]

Az oktatás gyakorlat-orientált, az egyes témakörök alkalmazássokkal, példákkal kerülnek ismertetésre. A tantárgyakon tárgyalt főbb témakörök a következők:

- FPGA felépítése, alkalmazási területei
- Korszerű FPGA architektúrák
- Bevezetés a hardverleírásba
- VHDL nyelvi elemek, HDL modellezési szintek
- Xilinx Vivado fejlesztői környezet megismerése
- Kombinációs és logikai hálózatok megvalósítása
- FPGA programozása
- XDC file szerkesztése
- Tesztfile készítése
- ChipScope Pro virtuális műszerek használata
- IP-k felhasználása, készítése
- SoC, PSoC megoldások
- MicroBlaze alapú rendszer tervezése és fejlesztése

[7] Forrás: https://th.bing.com/th/id/OIP.OBsoSyB_d4oqK6QUHWdraAHaGA?pid=Api&rs=1 Letöltés dátuma: 2020.11.08.

3.2 Hallgatók által gyakran elkövetett hibák FPGA fejlesztés során

A gyakorlat-orientált oktatás célja, hogy a hallgatók az oktató által ismertetett példákat megértsék és kipróbálják, de emellett kiemelten hangsúlyos az önálló munkavégzés. Mind a mintapéldák másolása, mind az önálló kódolás magában hordozza a hibázás lehetőségét. Ezek a hibák főként figyelmetlenségből és/vagy gépelésből adódnak; feltételezve, hogy a kellő ismeretekkel rendelkezik a hallgató. Ennek tükrében, elkülöníthetők a kellő ismeret hiányából adódó hibák is.

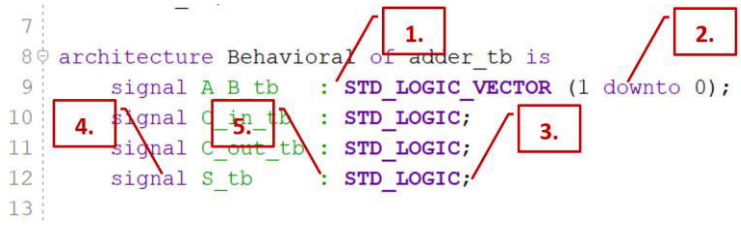
Ha a hallgató nem rendelkezik elég gyakorlattal a hibakeresésben, akkor azokat nagy valószínűséggel nem fogja megtalálni és nem tudja önállóan kijavítani. Ilyenkor két eset fordul elő tipikusan. Az egyik eset az, mikor a hallgató ezt nem jelzi és emiatt lemaradt, nem tud tovább haladni, ez a rosszabb változat. A másik eset, amikor szól, az oktató segítségét kéri. Ha tanórán túl sok hallgatónak kell segíteni, akkor az óra folytonossága sérülhet.

A továbbiakban az FPGA fejlesztés során, tesztfile és kényszerfile készítésekor, elkövetett figyelmetlenségből/elgépelésből adódó hibák kerülnek ismertetésre az oktatási tapasztalatok alapján. A példák az oktatás során használt környezetre vonatkoznak.

3.2.1 Tesztfileok készítése során elkövetett gyakori hibák

Az elkészített kódok tesztelésére általában nem a fizikai eszközön kerül sor, hanem tesztfájlokat készítenek a hallgatók. A tesztfájlok segítségével és a fejlesztői környezet szimulációs eszközét használva lehetőség nyílik az elkészített modulok viselkedési szimulációjára. A hallgatók által elkövetett gyakori hibák ismertetése szintaktikailag helyes kódrészleteken kerülnek bemutatásra.

A hallgatók által tipikusan elkövetett hibák első csoportja a tesztfájlban használt *objektumok deklarációjához* [5] köthető. Az 5. ábrán látható egy példa jel objektumok deklarációjára, vörös számokkal jelölve a kritikus részeket.

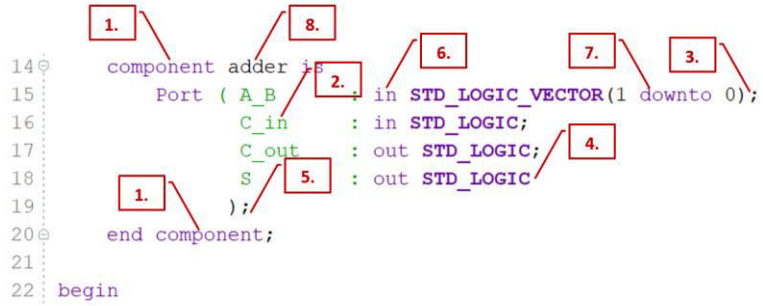


Ábra 5
Gyakori hibák az objektumok deklarációjában

Az objektumok deklarációja esetén először az objektumot kell megadni, ezt követően egy azonosítót hozzá, majd kettőspontot követően a típust, végezetül opcionálisan egy kifejezés, kezdőérték is rendelhető hozzá, de tesztfájl esetén ezt

ritkábban alkalmazzuk. Az *1.* számmal jelölt hiba arra utal, hogy a hallgatók a kettőspont után gyakran megadnak egy speciális módosítót, azaz az irányát a jelnek (in, out, inout, buffer, linkage), melyre itt nem, hogy nincs szükség, de szintaktikailag sem helyes. A *2.* jelölt rész azt mutatja, hogy abban az esetben, ha vektorokkal, bitfüzérekkel dolgozunk, gyakran elmarad a bitszám megadása vagy a méret nem egyezik. A *3.* jelölés a sorok végét lezáró pontos vessző elhagyására utal. A *4.* tipikus hibafaktor az objektum nevének lehagyása. Az *5.* jelölés pedig arra utal, hogy a kötelező kettőspont is sokszor lemaradt a deklarációból. Összeségében elmondható a jelek deklarációjáról az is, hogy az architektúra deklarációs részében kell megadni, azaz a *begin* kulcsszó elé kell kerülnie, gyakran szokott ezzel is probléma lenni, hogy nem elé, hanem mögé írják a hallgatók.

A hallgatók által elkövetett tipikus hibák második csoportja a tesztelni kívánt *komponens deklarációjához* [5] köthető. A komponens (*component*) egy entitás-architektúra párt reprezentál. Egy komponenst a példányosítás előtt deklarálni kell. Az előzőekben ismertetettekhez hasonlóan az architektúra begin előtti részében. Az ezzel kapcsolatos tipikus hibák és helyeik a 6. ábrán láthatók, szintén vörös számokkal jelölve.



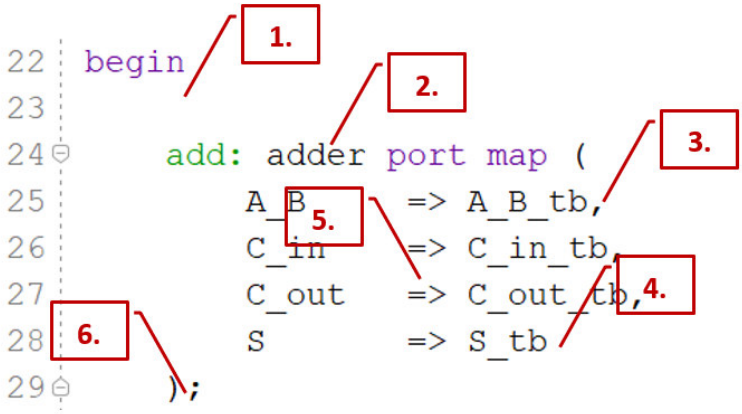
Ábra 6

Gyakori hibák a komponens deklarációjában

A hallgatók a komponens deklarálásakor a component és az end component (*1.*) részt gyakran lehagyják, vagy valamilyen más kulcsszót írnak helyette. A komponens deklarációja során meg kell adni a komponens generic listáját, ha van, és a port listáját. Gyakran előfordul, hogy a két listában szereplő elemek közül valamelyik kimarad a felsorolásból vagy az azonosítója elírásra kerül, ezt *2.*-vel került jelzésre. A *3.* jelölés mutatja, hogy a listaelemek pontosvesszővel kerülnek elválasztásra egymástól, de a *4.* helynél látszik, hogy az utolsó elem után már nincs pontosvessző. Itt tipikusan vagy lehagynak pontosvesszőt a hallgatók, vagy helyette vesszőt írnak, de előfordul, hogy az utolsó listaelem után is kiteszik azt. Ezek mindegyike szintén szintaktikailag helytelen. Az gömbölyű zárójel között megadott listákat is pontosvesszővel kell zárni, mely elhagyása szintén tipikus hibának tekinthető, ezt jelzi az *5.* sorszám. Az objektumok deklarációjával

ellentétben, a port listában szükség van a jelek módosítóinak (*6.*) helyes megadására. Ezek elhagyása vagy elírása szintaktikai hibát okoz. A *7.* hiba pedig hasonlatos az objektum deklarációnál elkövetett hibáknál ismertetett *2.* számmal jelölttel. Kiemelhető még a *8.* eset, amikor a komponens nevének helytelen megadása történik.

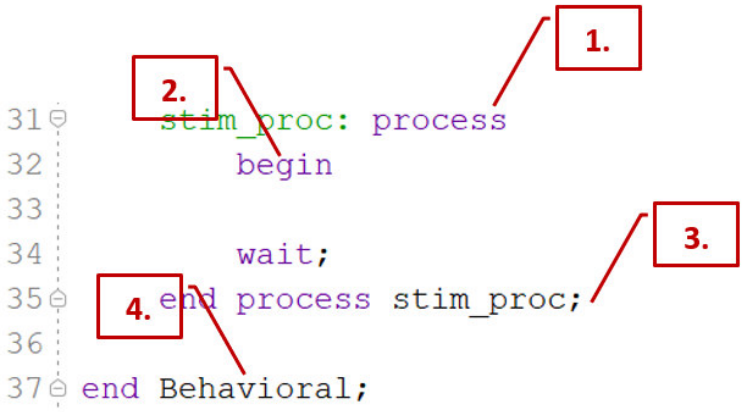
A deklarációs rész után, már lehetséges a komponens *példányosítása* [5]. A példányosítás során elkövetett tipikus hibákat a 7. ábra mutatja. A példányosítást az architektúra definíciós részében kell megtenni, azaz a begin (*1.*) utáni részben.



Ábra 7
Gyakori hibák a komponens példányosítása során

Itt ugyanúgy előfordul, hogy a komponens nevét (*2.*) elírja a hallgató. A *3., 4.* és *6.* jelölések vesző hibákra utalnak. A port map-ben minden sor végén vesszőnek (*3.*) kell lennie, kivéve sz utolsó soránál (*4.*). A port map gömbölyű zárójelek között kerül megadásra és a legvégét pontosvessző (*6.*) zárja le, mely szintén el szokott maradni. Az *5.* jelölés arra utal, hogy a megfeleltetés során a => kitétel esetén vagy lehagyják a hallgatók a =, > valamelyikét, vagy hibásan <=-t használnak. Természetesen itt előfordulhat még az is, hogy rossz jelekre végzik a megfeleltetést vagy kimarad valami a sorból.

A negyedik kiemelt rész a tesztfájlból a *stimulációs process* [5], mellyel a szimulációs eseteket szokás előállítani, ez a 8. ábrán látható.



Ábra 8
Gyakori hibák a stimulációs process-ben

Tipikus hiba a process (*1.*) kezdeti sorának végére felesleges vessző betételen, ugyanakkor gyakran elmarad vagy hibás a lezáró sor (*3.*). Kifejtve, a pontosvessző, az end process rész le szokott maradni vagy a végén a címke elírásra kerül. Előfordul az is, hogy az architektúra lezáró sora (*4.*) elé kerül véletlenül a process, mely szintén szintaktikai hibát eredményez.

Az előbbiekben ismertetett tipikus esetekből és példákból látszik, hogy még egy ilyen egyszerű tesztfile készítése során is rengeteg hibát el tudnak követni a hallgatók figyelmetlenségből, rossz gépelésből, másolásból adódóan.

3.2.2 Kényszerfileok készítése során elkövetett gyakori hibák

A kényszerfájlok elkészítésére a FPGA eszközök fizikai programozásához van szükségünk. A 9. ábrán a Nexys4 FPGA fejlesztői panelhez hozzáférhető kényszerfile sablon részlete látható. Ezt szerkesztve tudjuk az elkészített hálózathoz tartozó kényszereket megadni. Az ábrán látható, hogy egy 721 soros szöveges file-ról beszélünk. Ennek a szerkesztése a nagy méretéből adódóan is hordoz magában hibalehetőséget.

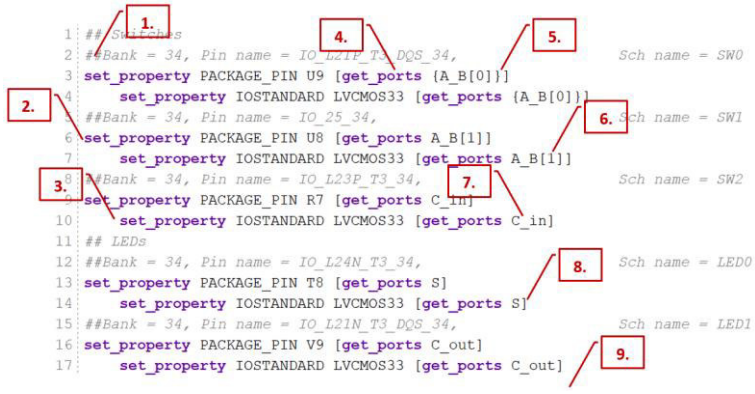
```
## Clock signal
##Bank = 35, Pin name = IO_L12P_T1_MRCC_35,                Sch name = CLK100MHZ
#set_property PACKAGE_PIN E3 [get_ports clk]
    #set_property IOSTANDARD LVCMOS33 [get_ports clk]
    #create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]

## Switches
##Bank = 34, Pin name = IO_L21P_T3_DQS_34,                 Sch name = SW0
#set_property PACKAGE_PIN U9 [get_ports {sw[0]}]
    #set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
##Bank = 34, Pin name = IO_25_34,                          Sch name = SW1
#set_property PACKAGE_PIN U8 [get_ports {sw[1]}]
    #set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]
##Bank = 34, Pin name = IO_L23P_T3_34,                     Sch name = SW2
#set_property PACKAGE_PIN R7 [get_ports {sw[2]}]
    #set_property IOSTANDARD LVCMOS33 [get_ports {sw[2]}]
##Bank = 34, Pin name = IO_L19P_T3_34,                     Sch name = SW3
#set_property PACKAGE_PIN R6 [get_ports {sw[3]}]
    #set property IOSTANDARD LVCMOS33 [get ports {sw[3]}]
...
##Bank = 14, Pin name = IO_L10P_T1_D14_14,                 Sch name = CRAM_A21
#set_property PACKAGE_PIN M16 [get_ports {MemAdr[21]}]
    #set_property IOSTANDARD LVCMOS33 [get_ports {MemAdr[21]}]
##Bank = 14, Pin name = IO_L23N_T3_A02_D18_14,             Sch name = CRAM_A22
#set_property PACKAGE_PIN U13 [get_ports {MemAdr[22]}]
    #set_property IOSTANDARD LVCMOS33 [get_ports {MemAdr[22]}]
```

Ábra 9

Nexys4 kényszerfájl sablon

A 10. ábrán láthatók a *kényszerfájlokhoz* [6] tartozó tipikus hibák jelölései egy helyesen elkészített kódrészleten. Egy ilyen kényszerfájlhoz úgy lehet hozzáférni, hogy egyik sora sem aktív, mindegyik kommentként szerepel. A szerkesztéshez a megfelelő sorok elől ki kell venni a #-et, majd az elnevezéseket kell testre szabni.



Ábra 10

Gyakori hibák a kényszerfájlban

Tipikus hibák között szerepel, ha a hallgató nem megfelelő sor elől (*1.*) veszi ki a #-et vagy nem veszi azt ki (*2., 3.*). További hibák, a get_ports (*4.*) kitörlése, a {} pár (*5.*) egyik felének a törlése és a [] pár (*8.*) egyik vagy mindkettő felének az eltávolítása. Gyakran előforduló hiba még a jelek nevének elírása (*7.*) vagy nem azonos megadása (itt számít a kis- és nagybetű), illetve vektorok, bitfüzérek (*6.*)

esetén a megfelelő elem jelölése. Mindezek mellett, előfordulnak még olyan esetek, amikor a hallgató nem használt PIN-eket is aktívvá tesz, vagy korábbi felhasználásból adódóan nem veszi azt észre. Sajnos előfordulnak olyan problémák is, mikor összekeverednek a PIN-ek megadásai és például egy kapcsolóra szánt jel egy LED PIN-re kerül, amely a működésben okozhat rendellenességeket.

Látszik, hogy itt, a kényszerfile-ok esetén is rengeteg lehetőség van hibákat elkövetni. Még így is, hogy egy előre elkészített sablont kell szerkeszteni. Abban az esetben, ha a hallgatóknak a sablon nélkül kéne dolgozniuk és önállóan megírni a kényszerfile tartalmát, még több hibalehetőség kerülne elő.

3.3 FPGA_Tools használata oktatói szemmel

A generátor programok és szoftverek használata nem egy újkeletű dolog fejlesztések során és ez igaz az oktatásra is. Emellett, az ipari tapasztalatok azt mutatják, hogy egyre gyakoribb a felhasználásuk. Oktatás folyamán, mikrokontroller programozással kapcsolatos tantárgyak esetén előfordul a kód-konfigurátor és generátor szoftverek használata, mi is alkalmazzuk.

Ilyen szoftverek esetén gyakran felmerül a kérdés, hogy mikor és milyen célzattal alkalmazzuk ezeket. Három esetet emelnénk ki. Az egyik eset az, amikor a hallgató nem rendelkezik kellő tudással vagy megfelelő gyakorlattal az adott feladat területén. Ekkor demonstrációs célzattal és a gyors sikerélmény elérése miatt hasznos lehet ilyen generátor szoftverek használata. A másik eset, amikor a hallgató birtokában van a feladat elkészítésihez szükséges ismeretanyagnak és kellő rutinnal is rendelkezik benne. Ilyenkor, egy generátor szoftver alkalmazása nagyban gyorsítja és hatékonyabbá teszi a munkát. A harmadik eset az, amikor ezeket a szoftvereket a hallgató ellenőrzésre tudja használni.

Oktatói szemmel, mindhárom esetben hasznos lehet egy ilyen szoftver. Hozzá kell tenni azt is, hogy a korábban felsorolt, a hallgatók által gyakran elkövetett hibák az FPGA fejlesztés során az FPGA_Tools szoftver használatával megszüntethetők, de legalábbis minimálisra csökkenthetők. Minél kevesebb hallgatók által elkövetett hibát kell az oktatónak megtalálnia és kijavítania, annál gördülékenyebben, hatékonyabban lehet tartani egy órát. Emellett meg kell jegyezni azt is, hogy ez nem azt jelenti, hogy a hallgatóknak nem kell képesnek lenniük önállóan, generátor program használata nélkül, tesztfájlt és kényszerfájlt készíteniük.

Konklúzió

Az FPGA_Tools program az Információs rendszerek előadás, hetedik féléves tantárgy, keretén belül alkalmazásra került, teszt- és kényszerfájl generálására is. A hallgatói visszajelzések és az oktatói tapasztalatok alapján bebizonyosodott, hogy a szoftver használata nagymértékben lecsökkenti az elkövetett hibák számát, és az ilyen fájlok létrehozásához szükséges időt. Mindezeket figyelembe véve, az alkalmazást tervezzük az elkövetkezendő félévek során az FPGA oktatás részeként bevezetni. Az FPGA_Tools fejlesztése folyamatosan zajlik, melyek során az egyetemről és az egyetemen kívülről érkező javaslatokat, észrevételeket is igyekszünk figyelembe venni.

Referenciák

[1] FPGA Market Forecast 2014-2027 https://hardwarebee.com/fpga-market-forecast-2014-2027/ (2020.09.15.)

[2] Karen Parnell, Nick Mehta: Programmable Logic Design Quick Start Handbook. Xilinx, Inc.: 2004

[3] PopularitY of Programming Language https://pypl.github.io/PYPL.html (2020.11.17.)

[4] TIOBE Index for December 2020 https://www.tiobe.com/tiobe-index/ (2020.12.01.)

[5] IEEE Std 1076 IEEE Standard VHDL Language Reference Manual 2000 https://edg.uchicago.edu/~tang/VHDLref.pdf (2020.10.15.)

[6] Xilinx: Vivado Design Suite - User Guide Using Constraints 2018 https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug903-vivado-using-constraints.pdf (2020.11.17.)