© Kiskapu Kft. Minden jog fenntartva

MyHDL: egy Python alapú hardverleíró nyelv

Az alkatrésztervezés végre belépett a XXI. századba. Ez az új eszköz a Python olvasható kódját ötvözi az extrém programozás tudományágával az alkatrész-projektjeinkben.

A digitális alkatrészterveket általában speciális leíró nyelven, az úgynevezett ***alkatrészleíró nyelven (hardware description language, HDL)*** készítik. Ezt a megközelítés azon a feltételezésen alapul, hogy az alkatrésztervezés különleges követelményeket támaszt. Manapság a legkedveltebb ***HDL*** nyelv a ***Verilog*** és a ***VHDL***.
A ***MyHDL*** projekt szakít a hagyománnyal és egy magas szintű, általános célú nyelvet a ***Python***-t teszi alkalmassá az alkatrésztervezésre. Ez a megközelítés lehetővé teszi az alkatrésztervezőknek, hogy kihasználják egy jól megtervezett, széles körben használt nyelv és a mögötte meghúzódó nyílt forrású modell előnyeit.

Fogalmak

A ***HDL*** használ néhány fogalmat amelyek az alkatrészek természetét írják le. A legtöbb leíró fogalom az erős párhuzamosítás modelljére vonatkozik. A ***HDL*** leírása rengeteg apró szálat tartalmaz, amelyek szoros kapcsolatban állnak egymással. Ez a körülmény megköveteli, hogy a szálásítás a lehető legegyszerűbb legyen. A ***HDL*** szerkezeteink egy kifejezetten erre a célra kialakított környezetben, a szimulátorban futnak.
A ***MyHDL*** tervezésekor a ***Python*** szellemiségére oly annyira jellemző egyszerűségre törekedtem, ami persze egyébként is hasznos dolog. A ***MyHDL*** egyik fontos része a ***Python*** felhasználási modellje. A másik rész a ***myhdl Python*** csomag, amely a ***HDL*** fogalmak objektumait tartalmazza. A következő ***Python*** kód néhány ***MyHDL*** objektumot importál, amelyeket hamarosan használni is fogunk:

```
from myhdl import Signal, Simulation, delay, now
```

A ***MyHDL*** a ***Python*** nyelvbe nemrég bevezetett `generator` típusú függvényekkel (lásd a hálózati forrásokat) szimulálja a párhuzamosságot. Ezek nagyon hasonlóak a hagyományos függvényekhez, azzal az eltéréssel, hogy van nem végzetes visszatérési értékük is. Amikor generátorfüggvényt hívunk meg, egy generátort ad vissza, azaz a minket érdeklő objektumot. A generátorok folytathatók és állapotukat az egyes meghívások között is megtartják, így kiválóan felhasználhatjuk őket szuper-könnyűsúlyú szálakként.

Következő példánk egy óragenerátort modellező generátorfüggvényt mutat be:

```
def clkgen(clk):
        “”” Clock generator.
        clk – clock signal
        “””
        while 1:
            yield delay(10)
            clk.next = not clk
```

A függvény nagyon hasonlít a hagyományos Python függvényekhez. Figyeljük meg, hogy a kód a while 1 ciklussal kezdődik; ezzel a pórias megoldással tartjuk örökre életben a generátorunkat. A hagyományos és a generátor függvény között a yield utasítás jelenti a lényegi különbséget. Nagyon hasonlóan működik a return utasításhoz, attól eltekintve, hogy a generátor a yield után tovább él és folytathatja futását erről a pontról. Továbbá a `yield` utasítás késleltető objektumot ad vissza. ***MyHDL*** alatt ezzel a mechanizmussal adjuk át a vezérlési információkat a szimulátornak. Jelen esetben a szimulátort arról értesítettük, hogy tíz időegység múltán kell visszatérnie a végrehajtáshoz.
A `clk` paraméter jelképezi az órajelet. ***MyHDL*** alatt a generátorok közötti kommunikáció ilyen jelzésekkel történik.
A jelzés fogalmát a ***VHDL***-től kölcsönöztük. A jelzés objektum két értékkel rendelkezik: a csak olvasható aktuális értékkel valamint a következő (next) értékkel, amely a `.next` attribútumhoz rendelve állíthatunk be. Példánkban az órajel úgy váltakozik, hogy a következő értéket az aktuális érték inverzére állítjuk be.
Az órajelgenerátor modellezéséhez először is előállítjuk az óra jelzést:

```
clk = Signal(bool(0))
```

A `clk` jelzéshez a logikai `zero` kezdeti értéket rendeljük. Most már a generátorfüggvény meghívásával létrehozhatjuk az órajel-generátorunkat:

```
clkgen_inst = clkgen(clk)
```

1. lista MyHDL SPI szolga modell

```
from myhdl import Signal, posedge, negedge, intbv

ACTIVE_n, INACTIVE_n = bool(0), bool(1)
IDLE, TRANSFER = bool(0), bool(1)

def toggle(sig):
    sig.next = not sig

def SPISlave(miso, mosi, sclk, ss_n,
             txdata, txrdy, rxdata, rxrdy,
             rst_n, n=8):
    """ SPI Slave model.

    miso – „master in, slave out" soros kimenet
    mosi – „master out, slave in" soros kimenet
    sclk –  eltolási óra bemenet (shift clock
    ➥input)
    ss_n – aktív alacsony szolga választó bemenet
    txdata – n-bites bemenet a küldendő adattal
    txrdy – ha új txdata fogadható, megváltozik
    rxdata – n-bites kimenet a fogadott adattal
    rxrdy – megváltozik, ha van elérhető rxdata
    rst_n - aktív alacsony reset bemenet
    n – paraméteres adat

    """

    cnt = Signal(intbv(0, min=0, max=n))

    def RX():
        sreg = intbv(0)[n:]
        while 1:
            yield negedge(sclk)
            if ss_n == ACTIVE_n:
                sreg[n:1] = sreg[n-1:]
                sreg[0] = mosi
                if cnt == n-1:
                    rxdata.next = sreg
                    toggle(rxrdy)

    def TX():
        sreg = intbv(0)[n:]
        state = IDLE
        while 1:
           yield posedge(sclk), negedge(rst_n)
            if rst_n == ACTIVE_n:
                state = IDLE
                cnt.next = 0
            else:
                if state == IDLE:
                    if ss_n == ACTIVE_n:
                        sreg[:] = txdata
                        toggle(txrdy)
                        state = TRANSFER
                        cnt.next = 0
                else: # TRANSFER
                    sreg[n:1] = sreg[n-1:]
                    if cnt == n-2:
                        state = IDLE
                    cnt.next = (cnt + 1) % n
                miso.next = sreg[n-1]

    return RX(), TX()
```

A legegyszerűbb de még használható szimuláció létrehozásához készítsünk egy másik generátort, amely megfigyeli, és kiírja az óra jelzéseinek változását:

```
def monitor():
    print "time: clk"
    while 1:
        print "%4d: %s" % (now(), int(clk))
        yield clk
```

A `yield clk` utasításból láthatjuk, miképpen vár a generátor a jelzés értékének megváltozására.
MyHDL-ben a szimulátort a `Simulation` objektum konstruktorral készíthetjük el amely tetszőleges számú generátort vár paraméterként:

```
sim = Simulation(clkGen_inst, monitor())
```

a szimulátor elindításához meghívjuk a `run` metódust:

```
sim.run(50)
```

Ezzel 50 időegységen keresztül futtatjuk a szimulátort. A kimenet a következő lesz:

```
$ python clkgen.py
time: clk
   0: 0
  10: 1
  20: 0
  30: 1
  40: 0
  50: 1
```

Most már tudjuk, hogyan működik a szimulátor. A szimulátor algoritmusát a ***VHDL*** ihlette, amely ugyan kevésbé népszerű ***HDL*** mint a ***Verilog***, viszont jobb követendő példa. Az összes generátor párhuzamos végrehajtását a szimulátor esemény vezérelt algorit mussal oldja meg. A generátor `yield` utasításában megadott objektum határozza meg, azt az eseményt amelyre a következő meghívásáig várakozni szeretne. Tegyük fel az egyik szimulációs lépésnél néhány generátor

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva

2. lista Teszkörnyezet az SPI Slave Module-hoz

```
import unittest
from random import randrange

from myhdl import Signal, intbv, traceSignals

from SPISlave import SPISlave, ACTIVE_n,
➥INACTIVE_n

def TestBench(SPITester, n):

    miso = Signal(bool(0))
    mosi = Signal(bool(0))
    sclk = Signal(bool(0))
    ss_n = Signal(INACTIVE_n)
    txrdy = Signal(bool(0))
    rxrdy = Signal(bool(0))
    rst_n = Signal(INACTIVE_n)
    txdata = Signal(intbv(0)[n:])
    rxdata = Signal(intbv(0)[n:])

    SPISlave_inst = traceSignals(SPISlave,
        miso, mosi, sclk, ss_n, txdata,
        txrdy, rxdata, rxrdy, rst_n, n=n)

    SPITester_inst = SPITester(
        miso, mosi, sclk, ss_n, txdata,
        txrdy, rxdata, rxrdy, rst_n, n=n)

    return SPISlave_inst, SPITester_inst
```

elindul, ugyanis bekövetkezik az esemény amire vártak. A szimulációs fázisban az összes generátor lefut az aktuális jeleket használva, közben pedig beállítják a következő jelet (next signal). A második fázisban az aktuális jelértékeket lecseréljük a következő jelben lévőkkel. A jelértékek változása következtében néhány generátor ismét aktívvá válik és a szimulációs ciklus megismétlődik. A mechanizmus garantáltan determinisztikus, ugyanis az aktív generátorok futtatási sorrendje a modell viselkedési szempontjából lényegtelen.

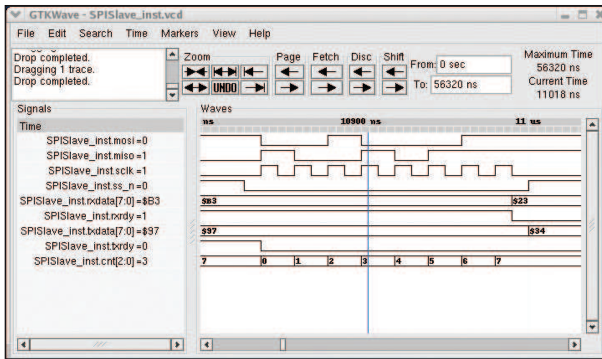
Valós tervezési példa

Az elvek bemutatása után készen állunk rá, hogy egy valódi ***MyHDL*** tervezési példával is megismerkedjünk. A soros külső csatolófelület (SPI) szolga alkatrész modulját választottam. Az ***SPI*** népszerű szinkron soros vezérlőfelület amelyet eredetileg a ***Motorola*** tervezett.
Egyetlen ***SPI*** mester több szolgát is irányíthat. Három általános I/O kaput használhatunk: a ***mosi***, azaz a mester ki, szolga be (master-out slave-in) soros vonalat, a miso, azaz a mester-be, szolga ki (master-in slave-out) soros vonalat, valamint az ***sclk***-t azaz a mester által vezérelt soros órát. Ezen kívül minden szolga rendelkezésére áll a szolga-választó (***ss_n***) vonal. Az ***SPI*** kommunikáció mindig egyszerre

3. lista Teszt eset SPI-on keresztül történő adatfogadáshoz

```
import unittest
from random import randrange

from myhdl import Simulation, join, delay,
➥intbv, downrange

from SPISlave import SPISlave, ACTIVE_n,
➥INACTIVE_n
from SPISlaveTestBench import TestBench

n = 8
NR_TESTS = 100

class TestSPISlave(unittest.TestCase):

    def RXTester(self, miso, mosi, sclk, ss_n,
                ➥txdata, txrdy, rxdata, rxrdy,
                ➥rst_n, n):

        def stimulus(data):
            yield delay(50)
            ss_n.next = ACTIVE_n
            yield delay(10)
            for i in downrange(n):
                sclk.next = 1
                mosi.next = data[i]
                yield delay(10)
                sclk.next = 0
                yield delay(10)
            ss_n.next = INACTIVE_n

        def check(data):
            yield rxrdy
            self.assertEqual(rxdata, data)

        for i in range(NR_TESTS):
            data = intbv(randrange(2**n))
            yield join(stimulus(data),
            ➥check(data))

    def testRX(self):
        """ Test RX path of SPI Slave """
        sim = Simulation(TestBench
           ➥(self.RXTester, n))
        sim.run(quiet=1)

if __name__ == ´__main__´:
    unittest.main()
```

zajlik mindkét irányban. Az adatváltozást kiváltó óra határt általában szabadon beállíthatjuk. Ebben a példában felfutó élet használunk.
Az ***SPI*** szolga ***MyHDL*** kódját a az *1. listában* mutatjuk be. Az ***SPISlave*** nevű klasszikus ***Python*** függvény segítségével



1. kép A gtkwave segítségével a tesztkészlet futása során megjeleníthetjük az összes jelet

modellezzük az alkatrész modult. A függvény az összes jelzést paraméterként kapja meg és két generátort ad vissza. A kód jól mutatja hogyan modellezzük ***MyHDL*** alatt a hierarchiát: a magasabb szintű függvény meghívja az alacsonyabb szintűt és a visszaadott generátorokat beépíti saját visszatérési értékébe.

A modul csatolófelület néhány további jelzést és paramétert is tartalmaz. A `txdata` az átadandó bemeneti adat szó, a `txrdy` pedig akkor változik ha új adatot fogadhatunk. Hasonlóképpen, az `rxdata` tartalmazza a fogadott adatszót, és az `rxrdy` változik ha új szót fogadtunk. Végül találunk egy bemeneti `reset` jelet (`rst_n`), és az bemeneti szószélességet meghatározó n paramétert.

Az ***SPI*** szolga modul belsejében létrehozzuk a `cnt` jelzést amiben a soros ciklusszámot tartjuk nyilván. Ez kezdeti értékként az `intbv` objektumot használja fel. Az `intbv` alkatrész orientált osztály amely bitszintű képességekkel rendelkező egészként viselkedik. A ***Python*** indexelő és szeletelő képességével elérhetjük az egyes biteket és szeleteket. Továbbá az `intbv` objektumnak lehet minimum és maximum értéke.

Az `RX` generátor függvények a fogadási viselkedést írják le. Valahányszor az `ss_n` szolga választó vonal aktív alacsony, a `mosi` bemenetet az `sreg` eltolási regiszterbe toljuk. A `yield negedge(sclk)` utasítás azt jelenti, hogy a művelet az ereszkedő órajel ciklusban hajtódik végre. Az utolsó soros ciklusban az eltolási regiszter az `rxdata` kimenetre kerül az `rxrdy` pedig átvált.

A `TX` generátor függvény valamivel bonyolultabb, mivel egy apró állapotgépre van szüksége a protokoll irányításához. A `yield` utasítás ebben az esetben két esemény jelöl, így a generátor az elsőként bekövetkező esemény hatására folytatja futását. Amikor a `reset` bemenet aktív alacsony, a `cnt` és az állapot alaphelyzetbe kerül. A másik esetben a művelet az állapottól függ. Az `IDLE` állapotban megvárjuk amíg a `select` vonal aktív alacsony és csak akkor fogadjuk el a szót átvitelre és lépünk be a `TRANSFER` állapotba. A `TRANSFER` állapotban a `shift` regisztert sorosan kitoljuk. Az állapotgép fenntartja a helyes soros ciklust és az utolsó kitolás után visszatér az `IDLE` állapotba.

© Kiskapu Kft. Minden jog fenntartva

Kapu a Linux világába
• cikkek
• hírek
• fórum
• címtár
Több mint 1000 ingyenesen letölthető cikk!
www.linuxvilag.hu

© Kiskapu Kft. Minden jog fenntartva

Ellenőrzés

Az *SPI* szolga modult a megvalósításhoz nagyon közeli szinten modelleztük. Ez jó lehetőséget adott nekünk a *MyHDL* fogalmainak bemutatására. Ugyanakkor a *MyHDL*-t ilyen célra használva nem jutunk túl sok előnyhöz a hagyományos *HDL*-ekhez képest. A *MyHDL* igazi ereje abban rejlik, hogy a teljes *Python* képességkészletet elérhetővé teszi az alkatrésztervezők számára. A *Python* kifejezőereje, rugalmassága és kiterjedt könyvtára már jóval túlmutat a hagyományos *HDL*-ek képességein.
Az egyik terület, ahol a *Python*-szerű képességek nagyon jól jönnek: az ellenőrzés. Akárcsak a programoknál az alkatrésztervezésnél is az ellenőrzés a keményebb dió. Általános vélemény, hogy a hagyományos *HDL*-ek nem igazán alkalmasak erre a feladatra. Következésképpen egy újabb nyelv jelent meg a színen, az ***alkatrész ellenőrző nyelv (hardware verification language avagy HVL)***. A *MyHDL* ezzel a megközelítéssel szemben a *Python* támaszkodik.
Az alkatrész ellenőrző környezet kialakításakor először is létrehozzuk a tesztasztalt. Ez az az alkatrész modul, amely a ***tesztelés alatt álló tervünk (design under test, azaz DUT)*** , valamint az adatgenerátorok és ellenőrzők összességét megjeleníti számunkra. A *2. lista* az *SPI* szolga modul tesztasztalát mutatja be. Itt az *SPI* szolga modult, és az *SPI* tesztelő modult láthatjuk, amely a csatolófelület tűit vezérli. Hogy egy időben több, a tervet különböző szempontok alapján vizsgáló *SPI* tesztelőt is alkalmazni tudjunk, az *SPI* teszt modul a tesztasztalunk paramétere lesz.
Magukhoz a tesztekhez tesztelő keretrendszert használunk. Az egységtesztelés a ***extrém programozás*** (*XP*) sarokköve. Ez a modern programfejlesztési módszer a józan ítélőképesség és radikálisan új elképzelések érdekes keveréke. Az eredeti *XP* megközelítés szerint először a tesztet kell kifejleszteni, még a megvalósítás előtt. Bár az *XP* igen hasznos módszertan, az alkatrészfejlesztők közössége általában mégis figyelmen kívül hagyja tanulságait. *MyHDL* alatt a *Python* egységtesztelő keretrendszere a ***unittest***, amely jó szolgálatot tesz nekünk a teszt-vezérelt alkatrészfejlesztésben.
A *3. lista* az *SPI* szolga modul teszt kódját mutatja be. A teszteket a `unittest.TestCase` osztály alosztályaiként adjuk meg. Az összes test előtaggal jelölt metódus a tényleges tesztre vonatkozik, a többi a teszt támogatására szolgál. Egy tipikus tesztben számos tesztet és tesztesetet találunk, itt azonban mintaképpen csak egytelen tesztet mutatunk be.
Az `RXTester` generátor függvény metódus célja az *SPI* szolga fogadási oldalának tesztelése. Tartalmaz egy stimulus nevű helyi generátor függvényt, amely mesterként továbbítja az adatszót az *SPI* buszon. Másik helyi generátor függvénye a `check`, amely ellenőrzi, hogy a szolga helyesen fojgadta-e az adatszót. A teljes teszt adott számú véletlen szó átviteléből áll. Minden adatszó esetében létrehozunk egy stimulus és egy `check` generátort. A *MyHDL* lehetővé teszi, hogy a `yield` utasításba helyezzük őket, így megvárhatjuk a befejezésüket. A helyes szinkronizáció érdekében csak akkor akarunk továbblépni ha már mindkét függvény kilépett. Ezt a feladatot a `join` függvény látja el.
A tesztprogram futtatásakor a kimenetben láthatjuk melyik teszt lett hibás melyik ponton. Amennyiben minden működik, apró példánk kimenete a következő lesz:

```
$ python test_SPISlave.py -v
Test RX path of SPI Slave ... ok
-----------------------
Ran 1 test in 0.559s
```

Hullámforma nézet

A *MyHDL* támogatja az alkatrész viselkedés népszerű vizsgálati módszerét, a hullámforma nézetet is. A 2. listában az *SPI* szolga modul létrehozását a `traceSignals` függvényhívásba csomagoltuk. Mellékhatásként a szimuláció alatt történt jelváltozások szabványos formátumú fájlba íródnak. Az *1. ábrán* a minta-hullámformánkat látjuk a nyílt forráskódú `gtkwave` hullámforma-nézegető megjelenítésében.

Kapcsolat más HDL rendszerekkel

A *MyHDL* praktikus megoldás, amely más *HDL* rendszerekkel is képes a kapcsolattartásra. A *MyHDL* támogatja a segéd-szimulációt más *HDL* szimulátorokkal, amennyiben azok rendelkeznek csatolófelülettel az operációs rendszerhez. Minden szimulátorhoz létre kell hozni egy hidat. Ez a nyílt forrású *Icarus* és *cver Verilog* szimulátorokhoz már el is készült.
Ezen túl, a *MyHDL* megvalósítás-orientált kódrészei automatikusan konvertálhatók *Verilog* alá. Ezt a `toVerilog` függvény segítségével érhetjük el, amelyet ugyanúgy használunk, mint a korábban bemutatott `traceSignals` függvényt. Az eredményül kapott kódot felhasználhatjuk a szokásos tervezési folyamatban például automatikusan szintetizálhatjuk megvalósításunkba.

Epilógus

Tim Peters, a híres *Python* guru, a következő paradox kijelentéssel jellemzi szeretett *Python* nyelvét: ***„a Python kódot könnyű kidobni.”*** Hasonló szellemben, a *MyHDL* célja olyan kedvelt alkatrésztervező nyelvvé válni, amelyben egyszerűen próbálhatunk ki új ötleteket.

Linux Journal 2004. november, 127. szám

Jan Decaluwe 18 évig dolgozott ASIC tervezőmérnökként és vállalkozóként. Jelenleg elektronikus tervezési és automatizálási tanácsadó. A jan@jandecaluwe.com címen érhető el.

KAPCSOLÓDÓ CÍMEK

- www.jandecaluwe.com/Tools/MyHDL/Overview.html
- sourceforge.net/projects/myhdl
- news.gmane.org/gmane.comp.python.myhdl
- www.python.org/peps/pep-0255.html
- www-106.ibm.com/developerworks/linux/library/l-pythrd.html
- www.embedded.com/story/OEG20020124S0116
- www.mct.net/faq/spi.html