# HLS alapú hardvertervezés. Ötletek az alapvető feladatok megoldásához.

## Bevezető

A téma keretében a Vivado HLS magas szintű szintetizáló eszközt alkalmazva valósítunk meg néhány egyszerű áramkört: összeadó vagy szorzó áramkör, tömb elemei összegének a kiszámítása. Bevezetünk a HLS módszerhez kapcsolódó alapvető fogalmakat és kényszerlehetőségeket, amelyek alkalmazhatóak az interfészek, portok, memóriák, ciklusok specifikálására. A magas szintű szintetizáló eszközök alkalmazása során meg kell határozni, hogy például C, C++ programkód különböző részei milyen módon valósíthatóak meg hardverben. Egy-egy függvénynek hardverre fordítására, tehát egy függvény alapján egy áramköri elem létrehozására nagyon sok lehetőség közül kell választani. A tervező a szintetizáló programnak recepteket kell megadjon, amely alapján elkészül a hardver.

Annak függvényében, hogy egy adott forráskódra milyen kényszerfeltételeket alkalmazunk, ugyanabból a programkódból különböző felépítésű hardverelem hozható létre.

A HLS eszköz alkalmazása során azt tapasztaltam, hogy a tervezés folyamán a legfontosabb az interfészek és bemeneti portok megértése, hiszen ezek ismeretében átláthatóvá és érthetővé válik az egyes alegységek összekapcsolása, a modulokba az adatok bevitele, valamint az eredmény kinyerése. Annak függvényében, hogy az interfészekre és port jelekre milyen protokollt alkalmazunk, különböző típusú bemeneti és kimeneti vezérlőjelek vannak alkalmazva.

A fejezetben egyszerű példák szemléltetésével elsajátítható a különböző kényszerfeltételek alkalmazása, mint például az interfész és port típusú kényszerfeltételek és a kapcsolódó vezérlőjelek.

A magas szintű szintézis során az egyik legfontosabb résznek a C, C++ forráskódból az RTL fordítás során az interfész kialakítását és megértését tartom. Pontosabban, hogy megértsük, hogy a függvényből, a függvény argumentumaiból és a visszatérített értékből hogyan lesznek hardver szintű port jelek, valamint különböző direktívák alkalmazása mellett milyen vezérlőjelek jönnek létre, amelyek segítik a tervezett modulok szinkronizálását.

Egy RTL implementáció porjeleit a következő elemekből lehet származtatni:

- bármely megadott függvény-szintű protokollból
- függvények argumentumából
- globális változók, amelyeket a legfelső szintű (top level) függvény elér

Az RTL szintézis során azt, hogy hogyan jönnek létre a modul port jelei, az INTERFACE kulcsszóval lehet meghatározni. A kényszerfeltételeket meg lehet adni a forráskódban #pragma kulcsszót követően, vagy egy külön .tcl szkript állományban.

Függvény szintű protokoll, vagy blokk szintű I/O protokoll vezérlő jeleket szolgáltat/illeszt a modul vezérléséhez. Ezen vezérlőjelekkel jelezzük a függvényből származtatott modulnak, amikor elkezdi a műveletek végrehajtását (start) és jelzi, amikor az elkezdett művelet sorozat befejeződik (done), a modul készenléti állapotban van (idle) és készen áll egy új bemenet fogadására (ready). A blokk szintű protokoll lényegében meghatározza, hogy hogyan kezeljük a tervezett alegységet: mikor kezdhet el működni, mikor fejezte be az adott számítási ciklust, valamint mikor áll készen új adatok fogadására. Az illesztett

vezérlőjelek segítenek az egyes modulok összekapcsolásában, egy rendszerbe integrált alegységek szinkronizálásában.

Függvényszintű protokollok típusát a *mode* kulcsszóval határozzuk meg. A Vivado HLS a következő négy függvényszintű protokollt alkalmazza:

- *Ap_ctrl_none:* e protokoll kiválasztása során nem generálódnak függvény szintű vezérlő jelek. Abban az esetben javasolt ennek a protokoll típusnak az alkalmazása, ha az adatfolyam folyamatos és nem kell megszakítani, leállítani a számolást, tehát nem szükséges egyéb vezérlőjelek csatolása.
- *Ap_ctrl_hs:* különböző vezérlő port jeleket alkalmaz a műveletvégzés indítására *(start),* valamint jelzi, hogy a modul készenléti állapotban van *(idle),* befejezte a műveletvégzést *(done)* és készen áll új adat fogadására. Az ap_ctrl_hs az alapértelmezett blokk szintű I/O protokoll.
- *Ap_ctrl_chain:* blokk szintű vezérlő jeleket implementál a modul működésének elkezdésére, a műveletek folytatására valamint jelzi, amikor a modul készenléti állapotban van, befejezte a műveletvégzést és készen áll egy következő adat fogadására.
- *s_axilite* AXI sínrendszerre illeszthető interfész jön létre

Az ap_ctrl_hs protokoll jelei

- *ap_start*: ez a jel vezérli a modul működését. A jelet '1' logikai szintre kell kapcsolni ahhoz, hogy a modul elkezdjen működni, a modul bemenetén található adatokat elkezdje feldolgozni. Ha az ap_start jelet hamarabb visszakapcsoljuk logikai '0' szintre, mint hogy az ap_ready logikai '1' szintre váltana, valószínű a modulnak nem sikerül az összes bemenetet beolvasni. A műveletvégzés emiatt leállhat a következő bemenet beolvasására várva.
- *ap_ready*: a kimenet jelzi, hogy a modul készen áll vagy sem egy új bemeneti adat fogadására. az ap_ready logikai '1' szintre való váltásakor a modulnak sikerült ebben a műveletvégzési ciklusban processzálni az összes beolvasott adatot és készen áll az új adatok fogadására,
- *ap_done:* kimeneti vezérlőjel, jelzi, hogy a modul befejezte az aktuális tranzakció összes műveletét. Mivel ez a tranzakció végét is jelenti, szintén jelzi, hogy az ap_return kimeneti porton érvényes az adat.
- *ap_idle:* kimeneti vezérlőjel jelzi azt, hogy az alegység működik vagy nem végez műveletet (idle). Várakozási állapotban (idle) az ap_idle kimenet magas logikai szinten van. Amikor a modul elkezdi a műveletvégzést, a kimeneti jel logikai '0' szintre vált. Logikai '1' szintre vált, amikor a modul befejezte a műveletvégzést.

Minden egyes függvény argumentumhoz, vagy abban az esetben, ha a függvény visszatérít értéket, a visszatérítési értékhez port szintű I/O interfész protokollt lehet meghatározni, mint például érvényes kézfogás (ap_vld) vagy kézfogás nyugtázása (ap_ack). Abban az esetben, ha az INTERFACE pragma *mode* paraméterét csak felső szintű függvényekre lehet alkalmazni, az alfüggvényekre az INTERFACE pragmának csak a *register* paramétere alkalmazható.

A mode kulcsszóval meghatározható az interfész protokoll típusa a függvények argumentumaira, függvények által alkalmazott globális változókra, vagy a blokk szintű vezérlő protokollra. Az alábbi listában felsorolunk a mode kulcsszóra pár lehetséges paramétert:

- `ap_none`: nem alkalmaz protokollt, az interfész egy egyszerű adatport

- `ap_stable`: nincs alkalmazva protokoll, a Vivado HLS feltételezi, hogy a reset jelet követően az adat a porton mindig stabil, amely lehetővé teszi a terv optimalizálása során a számítási láncban a felesleges regiszterek törlését.
- `ap_vld`: az adat porthoz egy jelző jelet rendel (valid), amely jelzi, hogy az adat készen áll az írásra vagy olvasásra.
- `ap_ack`: az adatporttal együtt implementál egy nyugtázó jelet (`ack`), amelyen nyugtázza, hogy az adaton megtörtént az írás vagy olvasás
- `ap_hs`: az adatport mellet alkalmazza az érvényes adat (valid) valamint a nyugtázó (acknowledge) jelet, egy kétirányú kézfogásos protokollt biztosítva, jelezve, amikor az adat érvényes írásra vagy olvasásra (vld) valamint nyugtázza, hogy az írás vagy olvasás megtörtént (ack).
- `ap_ovld`: a kimeneti adat porthoz egy érvényes jelző bitet csatol (vld) , amelyen jelzi, hogy az adat készen áll az olvasásra.
- `ap_fifo`: a portot egy standard FIFO interfészként valósítja meg, alkalmazza a ki és bemeneti adat portokat kiegészítve az üres (empty) és tele (full) portjelekkel. Az ap_fifo nem biztosítja a kétirányú írást vagy olvasást, vagy csak az argumentumok írását vagy az olvasását teszi lehetővé
- `ap_bus`: egy mutatót sín interfészként valósít meg
- `ap_memory`: tömb típusú argumentumot standard RAM interfészként valósít meg.
- `bram`: a tömb típusú argumentumokat standard RAM interfészként valósítja meg. RTL alapú tervezés esetében a memória interfész egy portos memóriaként jelenik meg.
- `axis`: az összes portot AXI4-Stream interfészként valósítja meg
- s_axilite: az összes portot AXI4-Lite interfészként valósítja meg
- m_axi: az össze portjelet AXI4 interfészként értelmezi. A config_interface paraméterrel egyéb AXI4 típusú paraméterek konfigurálhatóak, mint például 32, vagy 64 bites címzés kiválasztása.

*register:* fakultatívan alkalmazható kulcsszó, amely egy kimeneti/bemeneti jelhez vagy vezérlő jelhez regisztert rendel. Az eredmény, hogy a regiszterrel ellátott jelek mindaddig fennmaradnak, amíg legalább a függvény végrehajtásának utolsó ciklusa meg nem történik.
A regiszter opció a következő interfész módok esetében alkalmazható:

- ap_none
- ap_ack
- ap_vld
- ap_ovld
- ap_hs
- ap_stable
- axis
- s_axilite

A függvény argumentumokra és globális változókra alapértelmezetten a következő protokoll van alkalmazva:

- Csak olvasható (bemenet): ap_none.
- Csak írható (kimenet): ap_vld.
- Írható /olvasható (ki-bemenet): ap_ovld.
- Tömbök: ap_memory.

| Kényszer parancsok megnevezése | Kényszer értelmezése |
|---|---|
| ALLOCATION | Korlátozza az alkalmazható művetek, műveletvégző egységek számát, ezáltal megosztott hardver erőforrás van alkalmazva. |
| ARRAY_MAP | Több kisebb tömböt egy nagy tömbbe egyesít, ezáltal csökkenti a szükséges blokk RAM erőforrások számát. |
| ARRAY_PARTITION | Nagyobb méretű tömböket feloszt kisebb méretű tömbökre vagy regiszterekre, javítva az adatokhoz való hozzáférést és csökkentve a BRAM memóriák alkalmazásából származó szűk keresztmetszetet. |
| ARRAY_RESHAPE | Átalakítja a memória tömböket úgy, hogy egy több elemű tömböt átalakít kisebb elemű, viszont nagyobb szószélességet tároló tömbre (több adatbittet). Javítja az adatokhoz való hozzáférést anélkül, hogy megnövelnénk a használt BRAM memóriák számát. |
| DATA_PACK | Egy struktúra adatmezőit egyetlen szélesebb szószélességgel rendelkező skalárba csomagolja. |
| DATAFLOW | Feladatszintű csővezeték (pipeline) kialakítását engedélyezi, biztosítva függvények és ciklusok párhuzamos működését, csökkentve a műveletvégzéshez szükséges ciklusok számát |
| DEPENDENCE | Az utasítással egyéb kiegészítő információkat szolgáltat a szintézishez a ciklusokból származó függőségek feloldására, lehetővé téve a ciklusokból csővezeték kialakítását. |
| EXPRESSION_BALANCE | Engedélyezi az automata kifejezés kiegyensúlyozás kikapcsolást.<br>C-ben egy feladat egy szekvenciális műveletsorozattal van specifikálva, amely az RTL szintézisben egy hosszú műveletláncot eredményez, növelve a műveletvégzéshez szükséges órajelek számát. A HLS szintetizáló eszköz, kihasználva a műveletek asszociatív és kommutatív tulajdonságát, újrarendezi a műveletek végrehajtási sorrendjét. Az újrarendezés egy kiegyensúlyozott számítási fát eredményez, csökkentve a számítási lánchoz szükséges órajelek számát. |
| FUNCTION_INSTANTIATE | Függvények többszöri példányosítása során lehetővé teszi a lokális optimalizálást.. |
| INLINE | Ezzel a kényszerfeltétellel eltávolítja a függvényhierarchiát. Függvények határain átívelő logikai optimalizálást engedélyez. Csökkenti az óraciklusok számát, kiiktatva a függvényhívások költségeit. |
| INTERFACE | Meghatározza a függvényből és argumentumaiból az RTL modul portjainak származtatását. |
| LATENCY | Ezzel a megkötéssel meghatározható a minimum és maximum óraciklus. |
| LOOP_FLATTEN | Lehetővé teszi egymásba ágyazott ciklusok egyetlen ciklusba való összevonását, csökkentve a művelethez szükséges óraciklusokat. |

| | |
|---|---|
| LOOP_MERGE | Egyesíti az egymást követő ciklusokat, csökkentve az általános késleltetést, növelve a megosztott erőforrás használatot és javítja a logikai optimalizálást (egyszerre az összevont ciklusokra). |
| LOOP_TRIPCOUNT | Változó ciklus iteráció esetében megad egy becsült iteráció számot. Nincs hatása a szintézisre, csak a szintézis eredményét kiértékelő jelentésben. |
| PIPELINE | Ciklusokon és függvényeken csővezetéket hoz létre, csökkenti az inicializálási intervallumot, engedélyezve a párhuzamos végrehajtást |
| PROTOCOL | A programkód egy részének megengedi, hogy protokoll részként kezeljük. A protokoll programrészből interfész protokoll határozható meg (alakítható ki). |
| RESET | Reset jel globális vagy lokális állapotváltozókhoz való hozzáadását vagy eltávolítását teszi lehetővé |
| RESOURCE | Egy változóra, tömbre, aritmetikai műveletre vagy függvényargumentumra meghatározza, hogy milyen könyvtári erőforrással legyen megvalósítva |
| STREAM | Meghatározza, hogy agy adott tömb FIFO vagy memória csatornával legyen megvalósítva az adatfolyam optimalizálása során |
| UNROLL | Kifejti, kitekeri a ciklusokat, több független műveletet végző modult alkalmaz. Teljes cikluskifejtés esetében annyi műveletvégző modult alkalmaz, ahány ciklus iteráció van |

A kurzus nem teszi lehetővé az interfész vagy port típusú kényszerfeltételek teljes körű bemutatását, de egy pár példán keresztül megpróbálja az érdeklődőt rávezetni a fontosabb tervezési ötletek megértésére és alkalmazására. Első lépésben egy egyszerű szorzó áramkört mutatunk be, és különböző kényszerfeltételeket alkalmazva tárgyaljuk a létrejött interfész vezérlő jeleit (adat, illetve vezérlő jelek).

Bemutatjuk a különbséget a létrejött portjelek szempontjából, ha:

- az eredményt a függvény teríti vissza
- az eredményt egy argumentum szolgáltatja.
- a kimenetekre regisztereket illesztünk

## Példa 1. Szorzómodul, az eredmény a függvény visszatérítési értéke

Első esetben a művelet eredményét függvény téríti vissza, a C program a következő részben van bemutatva:.

A header állomány tartalma:

```
#ifndef _FUNC_SIZED_H_
#define _FUNC_SIZED_H_

#include <stdio.h>
#include "ap_cint.h"

typedef int12 din_t;  // típus deklarálás 12  bites előjeles egész szám
typedef int24 dout_t; // típus deklarálás 24  bites előjeles egész szám
```

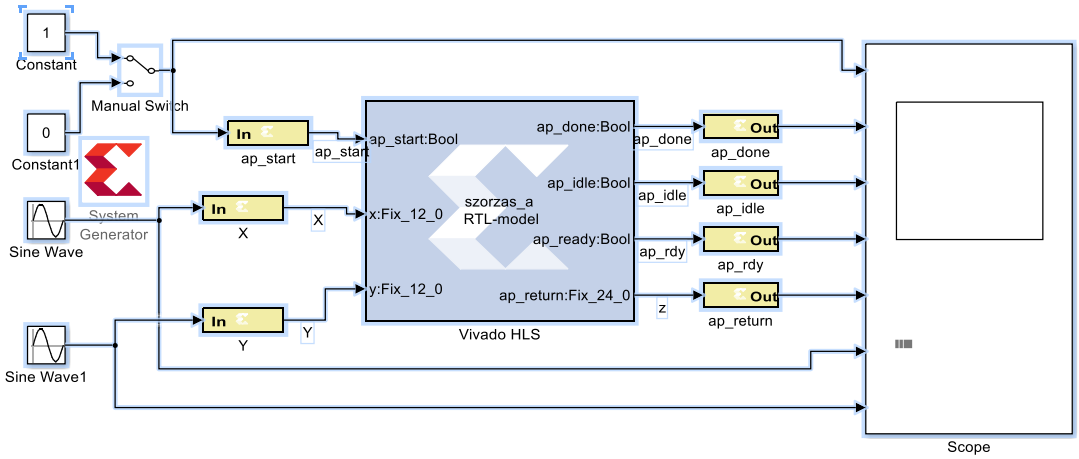
```
dout_t szorzas_a(din_t a,din_t b);

#endif
```

A szorzást megvalósító felsőszintű (top level) függvény:

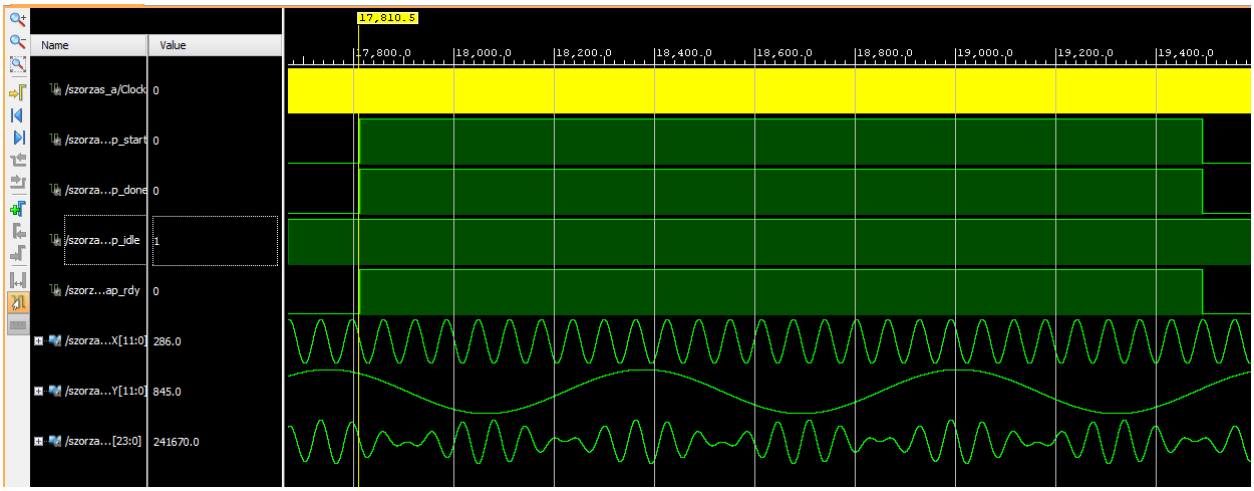
```
#include "szorzas_a.h"

dout_t szorzas_a(din_t x, din_t y) {
  int tmp;

  tmp = (x * y);
  return tmp;
}
```

Ha nem alkalmazunk egyetlen megkötést (direktívát sem), a ki- és bemeneti jelek mellett a következő függvény szintű vezérlő jelek generálódnak: ap_start, ap_done, ap_ready, ap_idle. Tehát az alapértelmezetten alkalmazott függvény szintű protokoll: *ap_ctrl_hs.*

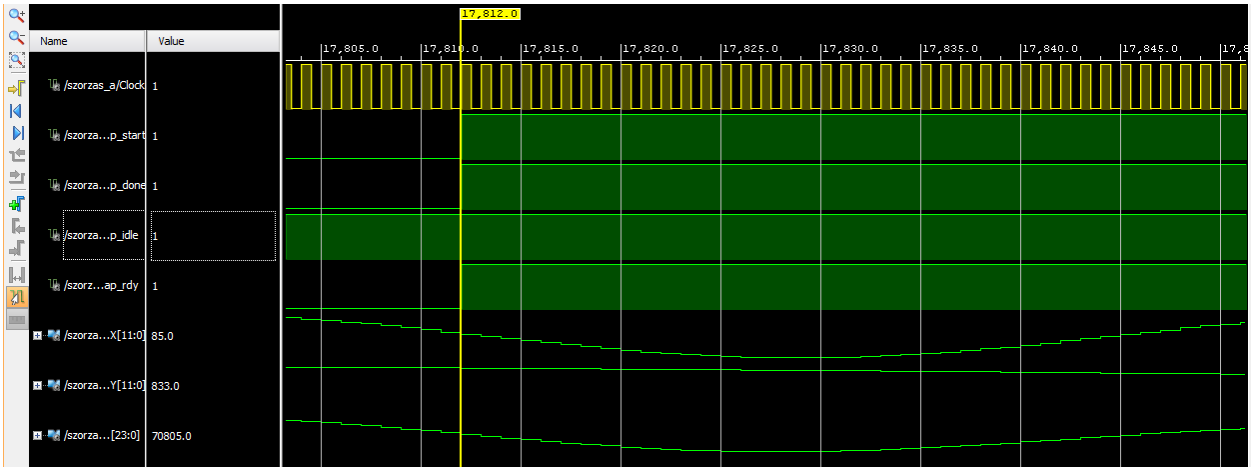
A példa egyszerűbb értelmezése és megértése végett a szintetizált RTL modellt generáltuk és exportáltuk System Generatorba. Hasznosnak találom grafikusan megjeleníteni a hardver modult, ezáltal könnyen integrálható egy célfeladatban (akár a modul tesztelése céljából) és vizuálisan is értelmezhetőek a különböző kényszerfeltételek mellet létrejött vezérlő jelek.




*. Ábra. HLS ben generált szorzó modul System Generator alapú tömb vázlata. Alapértelmezett függvényszintű protokoll: ap_ctrl_hs. A két összeadandó értéket az X és Y bemeneteken kapcsoljuk a szorzó modulra. A két különböző periódusú szinusz jelet két szinusz generátor szolgáltatja.*

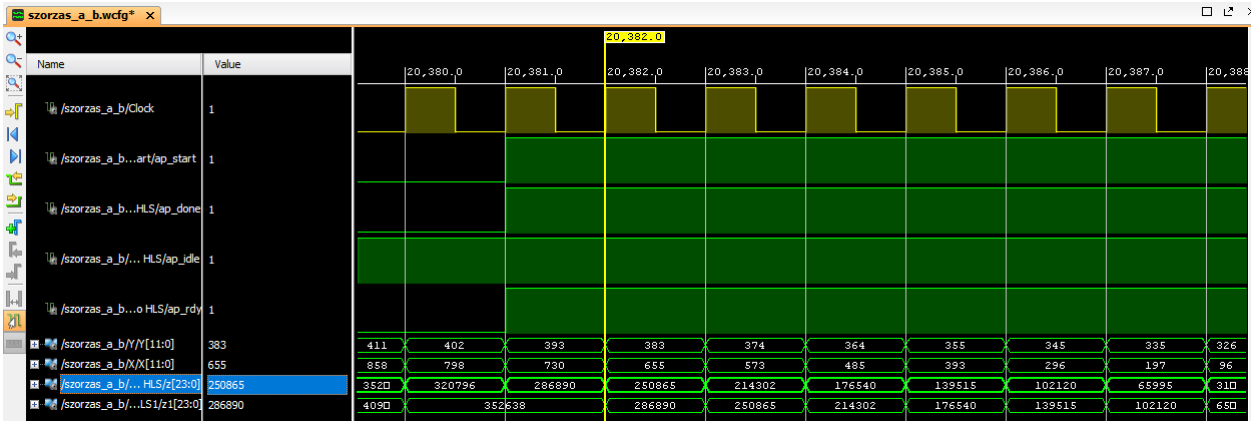



*1. Ábra. A rajzon láthatóak a vezérlőjelek, a két bemeneti szinusz jel és a kimenetként a két szinusz jel szorzata*



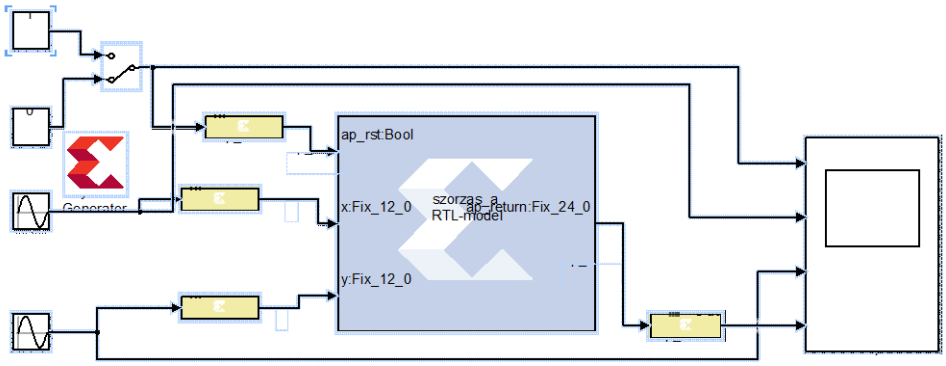

*2. Ábra részlet a szimulációból, órajel szinten értelmezhető a műveletvégzés. Az ap_start logikai '1'-esre való kapcsolását követően az ap_done és ap_ready is logikai magas szintre vált.*

Sem a bemenethez, sem a kimenethez nem volt regiszter illesztve. A szimulációs ábrán megfigyelhető, hogy amelyik pillanatban a bemenet start vezérlőjelet logikai egyesre kapcsoljuk (modul működés engedélyezve), a kimeneten a vezérlő jelek is azonnal váltanak, és a szorzás eredménye is azonnal megjelenik a kimeneten (szorzas_a peldaprogram).




*3. Ábra. Értékszinten is leellenőrizhető a szorzás eredménye. A z kimenet esetében az alapértelmezett ap_ctrl_hs protokoll, míg a z1 kimenet esetében az ap_ctrl_none protokoll van alkalmazva. A szimulációs diagramon csak az első modul vezérlőjelei vannak szemléltetve. A két modul párhuzamosan futott ugyanabban a szimulációban: mindkét modulra ugyanazok a bemeneti adatok vannak rávezetve. A z1 kimenethez regiszter van hozzárendelve, amint látható a szimulációs kimeneten is.*

```
#include "szorzas_a.h"

dout_t szorzas_a(din_t x, din_t y)
{
#pragma HLS INTERFACE ap_ctrl_none register port=return
dout_t tmp=0;
        tmp = (x * y);
  return tmp;
}
```

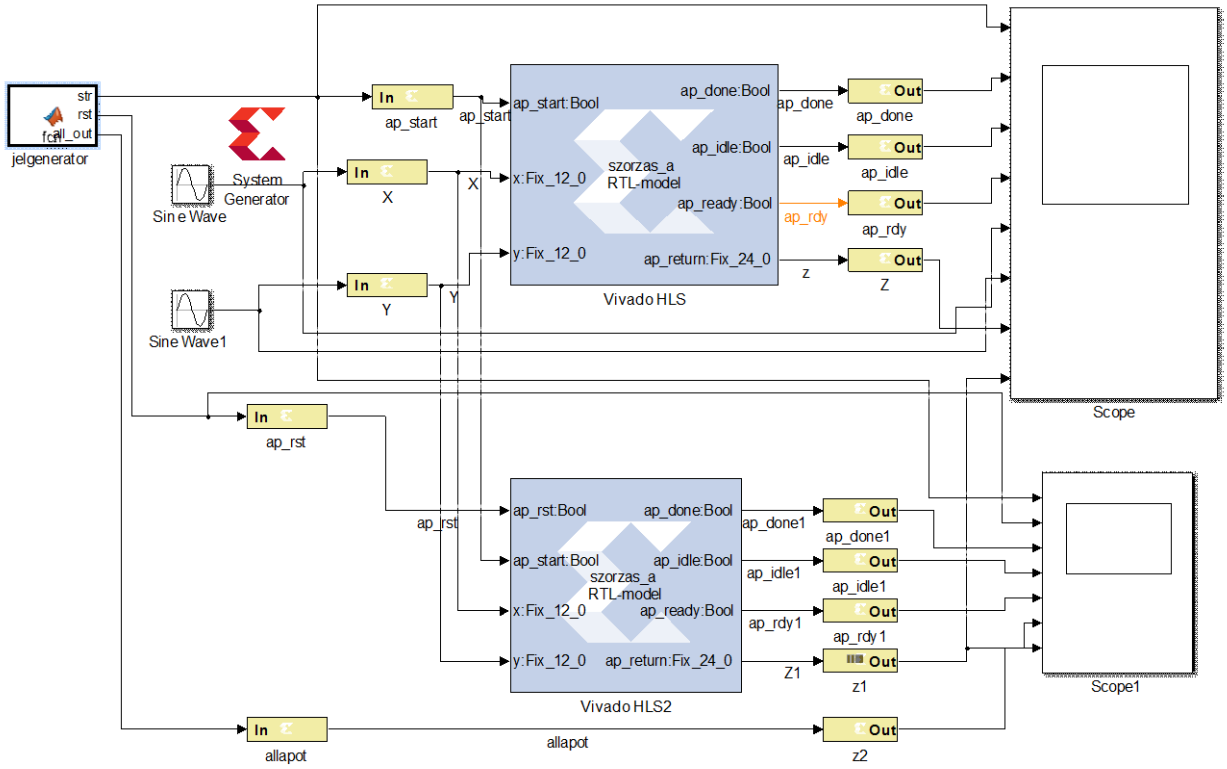



*4. Ábra ap_ctrl_none protokoll a kimenethez csatolt regiszterrel*

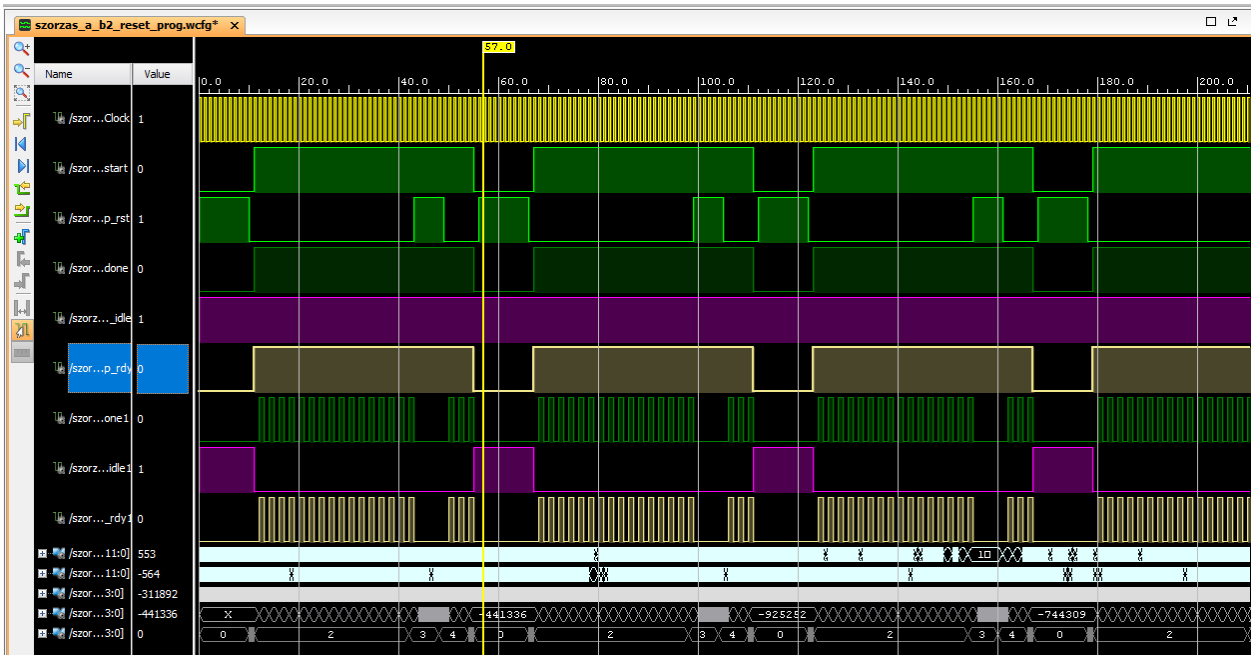
Abban az esetben, ha a függvényszintű protokollként az ap_ctrl_none módot választjuk és a kimenethez regiszter van illesztve, a vezérlőjelek között nem szerepel a bemeneten a start és a kimeneten az ap_done, ap_ready és ap_idle jelzőbitek. Az ap_start, ap_done, ap_ready, ap_idle nem jelennek meg mivel a függvény szintű az ap_ctrl_none interfész protokoll nem alkalmaz vezérlőjeleket. Az ap_rst bemeneti vezérlő jel a kimenethez rendelt regiszter miatt jelenik meg. A szimulációs eredményeket a 4. ábra szemlélteti. A z kimeneten a bemeneti adatokkal megegyező óraciklusban elérhető az eredmény. A z1 kimeneten a regiszter illesztése miatt egy óraciklussal később érhető el az eredmény (Ábra 4.).

A következő példában megpróbáljuk összehasonlítani, mi történik, ha az ap_ctrl_hs függvény szintű protokollra alkalmazunk, illetve nem regisztereket. Az alábbi Simulink modellben a felső modul esetében nincsenek, míg az alsó modul esetében vannak regiszterek alkalmazva.

```
dout_t szorzas_a(din_t x, din_t y)
{
#pragma HLS INTERFACE ap_ctrl_hs register port=return
dout_t tmp=0;
        tmp = (x * y);
  return tmp;
}
```




*5. Ábra A HLS-ben generált két szorzó modul. Mindkét modulra az alapértelmezett ap_ctrl_hs protokoll van alkalmazva. Az alsó modulban a kimenetekhez regiszterek vannak illesztve (szorzas_a_b2_reset_prog.slx).*




*6. Ábra Két szorzó modul működésének, a generált vezérlőjeleknek az összehasonlítása.*

A 6. ábrán bemutatott áramköri modul portjeleinek értelmezése:

| Jel megnevezése | Melyik modulhoz kapcsolódik a jel | Jel típusa |
|---|---|---|
| Clock | közös | órajel (bemenet) |
| ap_start | közös start | bemenet |
| ap_rst | 2 | reset jel (bemenet) |
| ap_done | 1 | a műveletvégzés befejeződött, kimeneti függvényszintű vezérlő jel |
| ap_idle | 1 | a modul készenléti állapotban van, kimeneti függvényszintű vezérlő jel |
| ap_rdy | 1 | az eredmény elérhető, kimeneti függvényszintű vezérlőjel |
| ap_done1 | 2 | a műveletvégzés befejeződött, kimeneti függvényszintű vezérlőjel |
| ap_idle1 | 2 | a modul készenléti állapotban van, kimeneti függvényszintű vezérlőjel |
| ap_rdy1 | 2 | az eredmény elérhető, kimeneti függvényszintű vezérlőjel |
| Y | | bemeneti adat |
| X | | bemeneti adat |
| z | | első modullal számolt eredmény |
| Z1 | | második modullal számolt eredmény |
| allapot | | a szimuláció során a szimuláció állapotát jelzi, egyszerűsítette a különböző tesztfázisok azonosítását |

Követve a szimulációs eredményeket (6. ábra), a következő következtetéseket lehet levonni:

1. Az ap_start jelnek az első modulra nézve nincsen különösebb szerepe. Logikai ‘1’-re való kapcsolásával azonnal, ugyanabban az óraciklusban az ap_done és ap_ready kimenetek is egyesre

váltanak, és folyamatosan a logikai ‘1’ szinten maradnak. Alacsony logikai szintre való kapcsolást követően az ap_done és ap_ready kimenetek is alacsony logikai szintre kapcsolnak. Viszont ha megfigyeljük az ap_idle kimeneti vezérlő bit logikai szintjét, levonható a következtetés, hogy az ap_start bemeneti vezérlőjeltől függetlenül folyamatosan logikai ‘1’ állapotban van. A logikai ‘1’ állapot azt jelenti, hogy a modul működik, logikai ‘0’ pedig azt, hogy készenléti állapotban van.

2. Ha megfigyeljük a második modul vezérlőjeleit, az ap_done1 és ap_ready jelek a műveletvégzés engedélyezését követően (ap_start=1) nincsenek folyamatosan logikai ‘1’ állapotban. A szimuláció szerint csak minden második órajelre történik új adatbeolvasás.
3. Második modul esetében
    a. ha az ap_start=0, ap_rst=1, vagyis a modulra kiadtunk egy reset jelet és a műveletvégzést nem indítottuk
        i. a szimuláció kezdetén a z1 kimenet határozatlan
        ii. az ap_idle1 jel logikai egyesben van, a modul várakozik a művelet végrehajtás elkezdésére
    b. ha az ap_start=0, ap_rst=0, mivel a műveletvégzés még nem kezdődött el, a modul várakozási állapotban van (ap_idle=’1’)
    c. ha az ap_start=’1’, ap_rst=0, az ap_start=’1’ jelezzük, hogy elkezdődhet a műveletvégzés (készen állnak az adatok a bemeneten), az ap_done1 illetve ap_ready1 vezérlő bitek jelzik, hogy a modul befejezett egy műveletvégzést (ap_done1) és készen áll a következő adat fogadására.
    d. ha az ap_rst1 a modul nem végez műveletet, az állapot bitek ap_done1 és ap_read1 logikai ‘0’ szinten maradnak.

### Példa 2. Szorzó modul: a szorzás eredménye argumentumként van visszatérítve

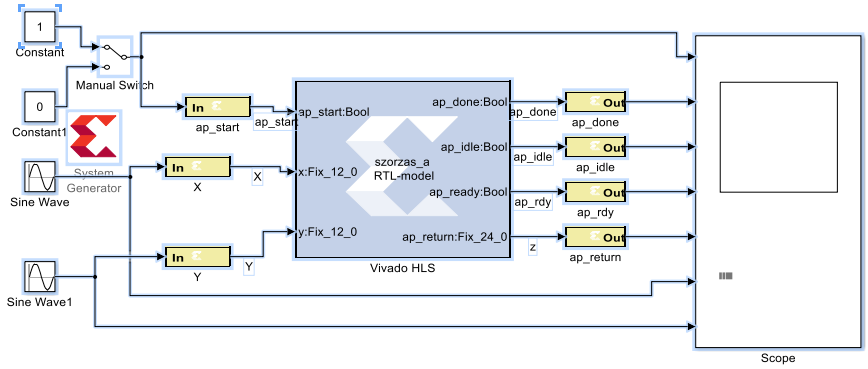
```
#include "szorzas_b.h"

void szorzas_b(din_t x, din_t y, dout_t *z) {
  int tmp;

  tmp = (x * y);
  *z=tmp;
  return ;
}
```

```
#include "szorzas_b.h"

void szorzas_b(din_t x, din_t y, dout_t *z) {
  int tmp;

  tmp = (x * y);
  *z=tmp;
  return ;
}
```

Összehasonlítva a két változatban visszatérített értéket, ugyanazt a függvény szintű protokollt alkalmazva mindkét változatra, megfigyelhető, hogy az argumentumként visszatérített érték esetében

a z kimenet mellett megjelenik z_ap:vld állapotjel, amelynek logikai ’1’ állapota jelzi, hogy érvényes az adat. A függvény argumentumából származtatott kimeneti porthoz alapértelmezetten ap_vld vezérlőjel van csatolva.

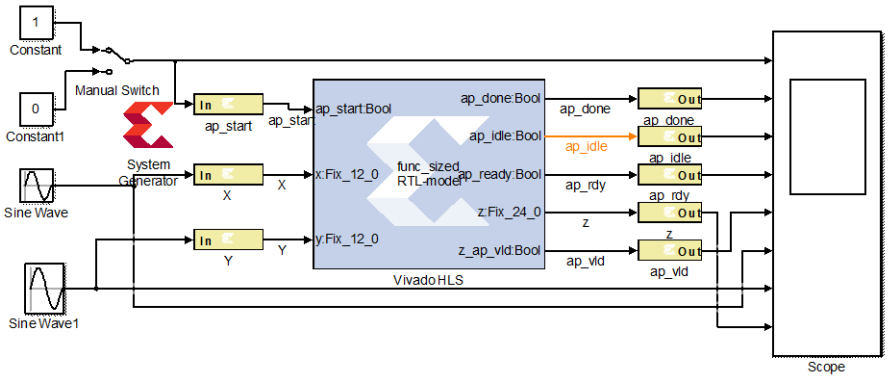
```
dout_t szorzas_a(din_t x, din_t y)
```



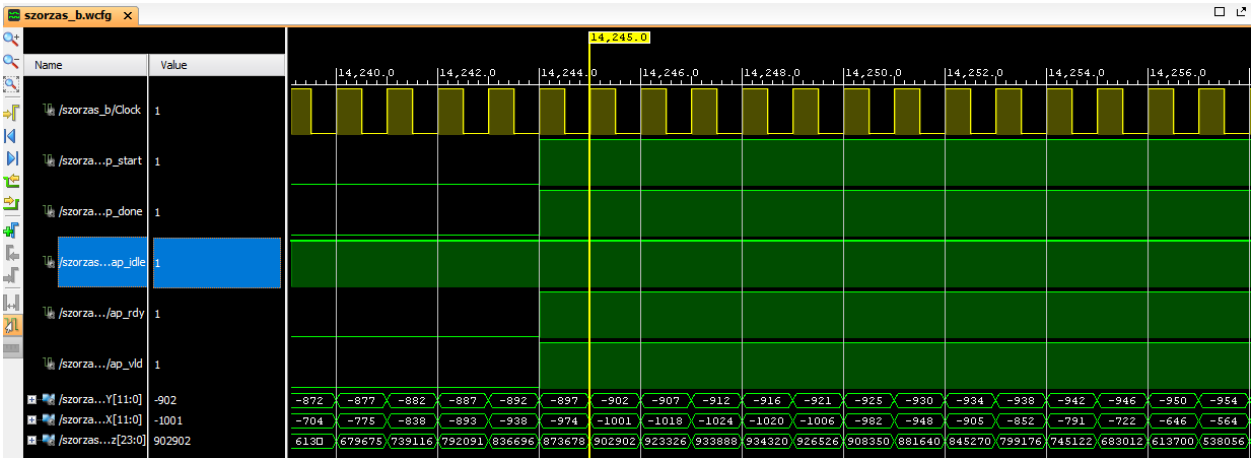

*7. Ábra függvényértékként visszatérített eredmény esetében ap_ctrl_hs protokollra a vezérlőjelek*

)

```
void szorzas_b(din_t x, din_t y, dout_t *z)
```




*8. Ábra függvény argumentumként visszatérített eredmény, az ap_ctrl_hs hez rendelt vezérlőjeleken kívül megjelenik az ap_valid port szintű vezérlőjel*




*9. Ábra*

Mivel nem regiszterelt az ap_ctrl_hs, a kimenet azonnal érvényes. A Simulink szimuláció szerint, amelyik pillanatban jelezzük, hogy van érvényes bemeneti adat, az ap_vld vezérlő jel is logikai ‘1’-re vált.

### Példa 3. Tömb elemeinek összege

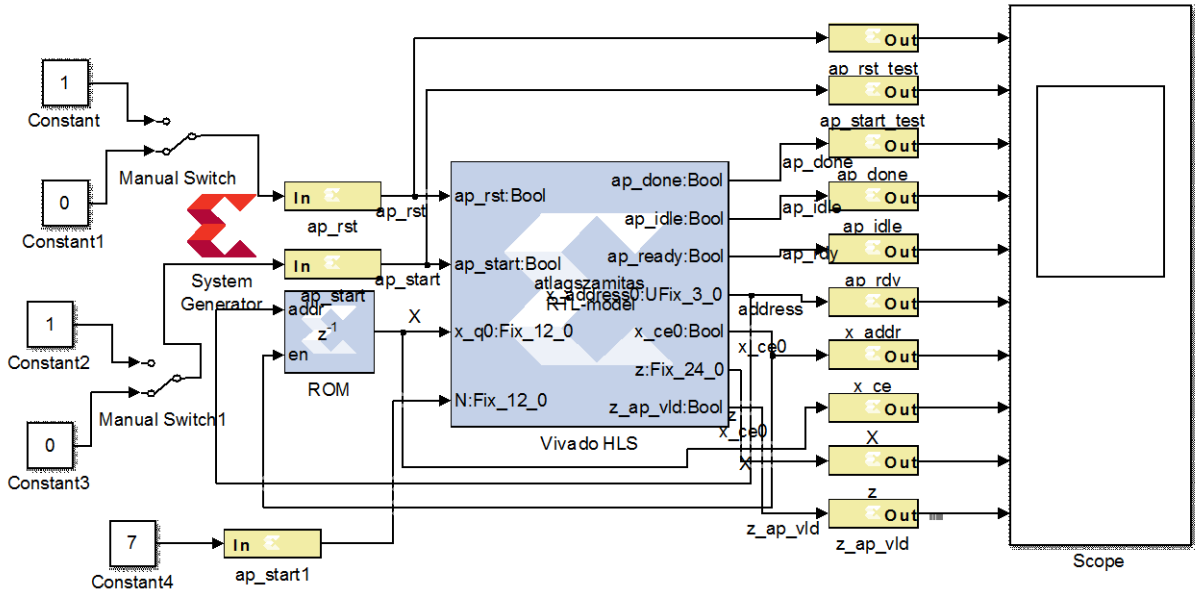
A következő példában HLS-ben tervezett áramkör kiszámolja N szám összegét. A példákban a tömb szintű bemenetekre alkalmazható interfész megoldásokat mutatunk be.

Az alábbi változat szerint a z kimenethez nincsen regiszter hozzárendelve, és az alapértelmezett függvény és port szintű protokoll van alkalmazva.

```
#include "osszegszamitas.h"

void osszegszamitas(din_t x[M], din_t N, dout_t *z)
{
  int tmp;
  int i;
  tmp=0;
  for (i=0;i<N;i++)
      tmp =tmp+x[i];
    *z=tmp;
  return ;
}
```

A következő megoldásban szintén az alapértelmezett protokoll van alkalmazva, viszont a z kimenethez regiszter van illesztve. Mindkét esetben az eredményt függvény argumentumként térítjük vissza. Szimuláció során összehasonlítjuk az szimuláció eredményét, ha a z kimenethez regisztert csatolunk. A direktívát a programkódba illesztettük.

```
#include "osszegszamitas.h"

void osszegszamitas(din_t x[M], din_t N, dout_t *z)
{
#pragma HLS INTERFACE register port=z
  int tmp;
  int i;
  tmp=0;
  for (i=0;i<N;i++)
      tmp =tmp+x[i];
    *z=tmp;
  return ;
}
```



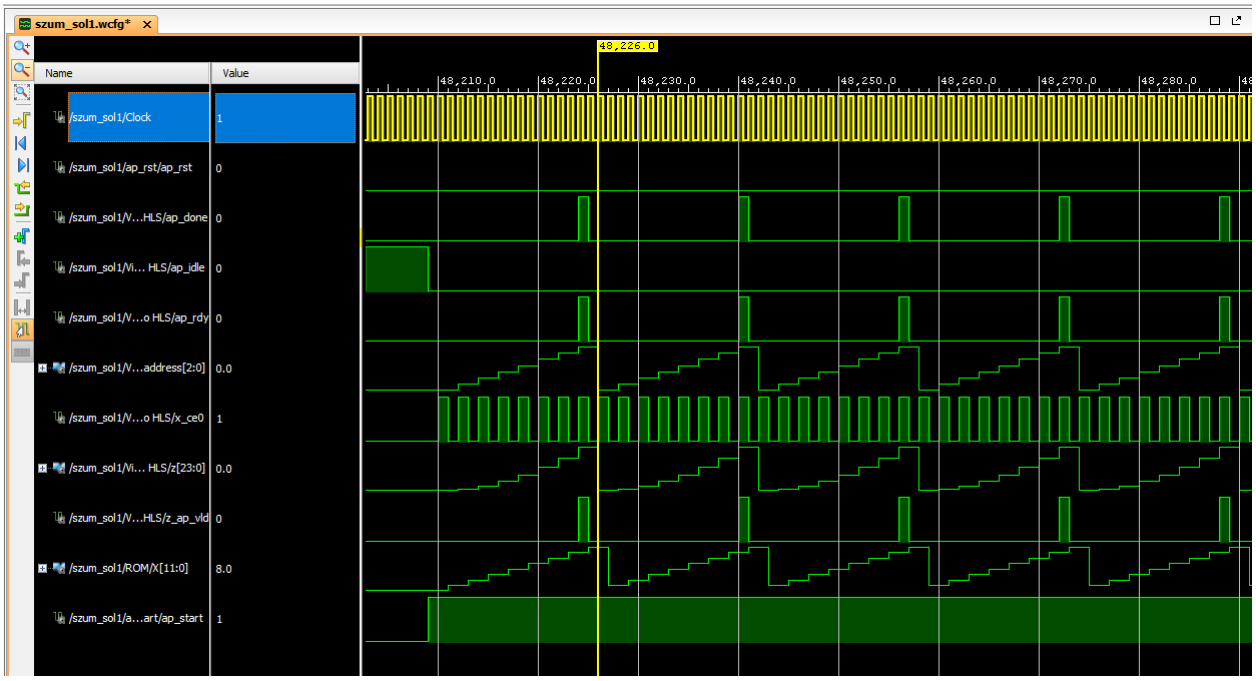

*10. Ábra Vektor elemei összegének a számítása. Alapértelmezett függvény és port szintű vezérlőjelek. Ha a bemenet tömb, a bemeneti tömbhöz alapértelmezetten bemeneti interfészként egyportos RAM interfész van alkalmazva.*

Értelmezzük a tervezést követően, milyen vezérlőjelekkel találkozunk a modulon:

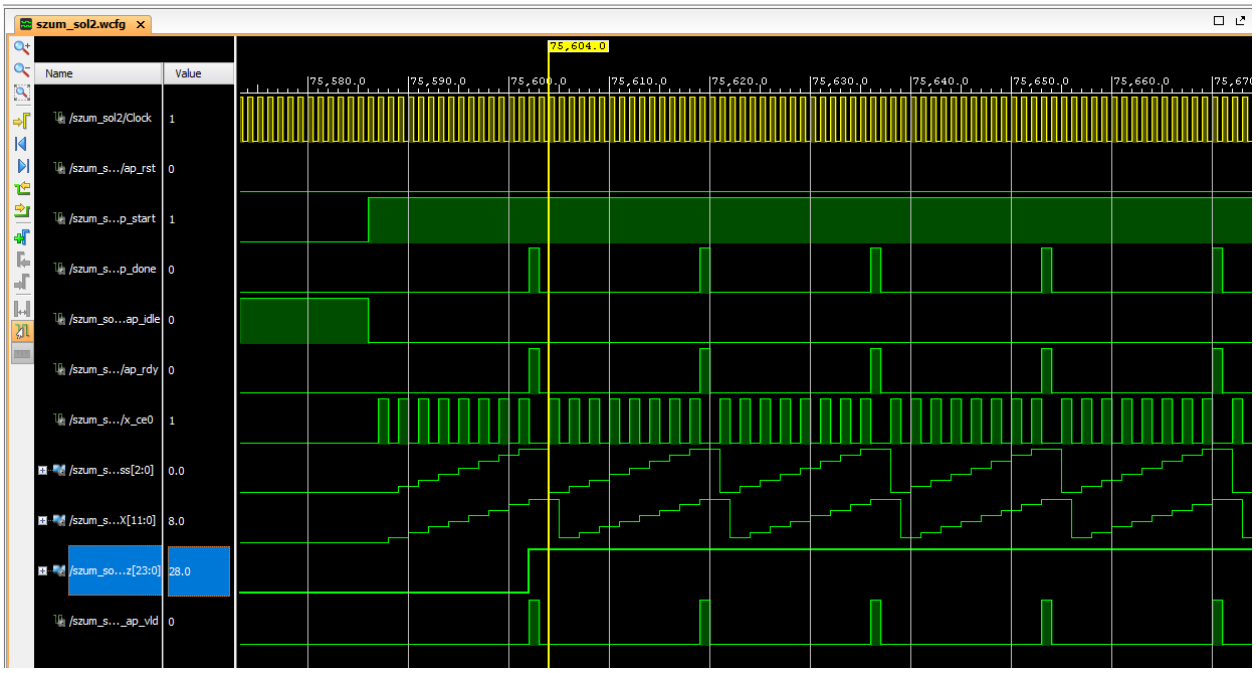
- függvény szintű (vagy blokk szintű) protokoll jelek:
  - -ap_start
  - ap_done
  - ap_idle
  - ap_rdy
  - ap_rst
- port szintű protokoll (argumentumhoz rendelt) jelek
  - -ap_vld –érvényes a z kimenet
  - x_address0, az x tömbhöz rendelt memória címzésére szolgál
  - x_ce- az x tömbhöz rendelt órajel engedélyező vezérlő jel, ütemezi a memóriából való olvasást

Mivel az X bemenet a függvény argumentumában deklarált tömb, alapértelmezetten BRAM típusú interfészként van kezelve. Ennek alapján a memória vezérlésére létrejöttek az x_address, x_ce vezérlőjelek. A szemléltetett példában az adatokat egy ROM memória szolgáltatja.

A szimuláció során elért eredmények a következő ábrákon vannak szemléltetve () ()




*11. Ábra Vektor elemei összegének a számítása. A kimeneti adat porthoz (z) nincsen regiszter illesztve. A szimuláció során a tömb elemeinek a száma 7. A szimuláció alapján is értelmezhető, hogy hét lépésben történik (x_ce0 jel) az adatok beolvasása és a nyolcadik lépésben érvényes az eredmény a kimeneten. Mivel a kimenethez nincsen regiszter csatolva, a részleges összeg folyamatosan megjelenik.*



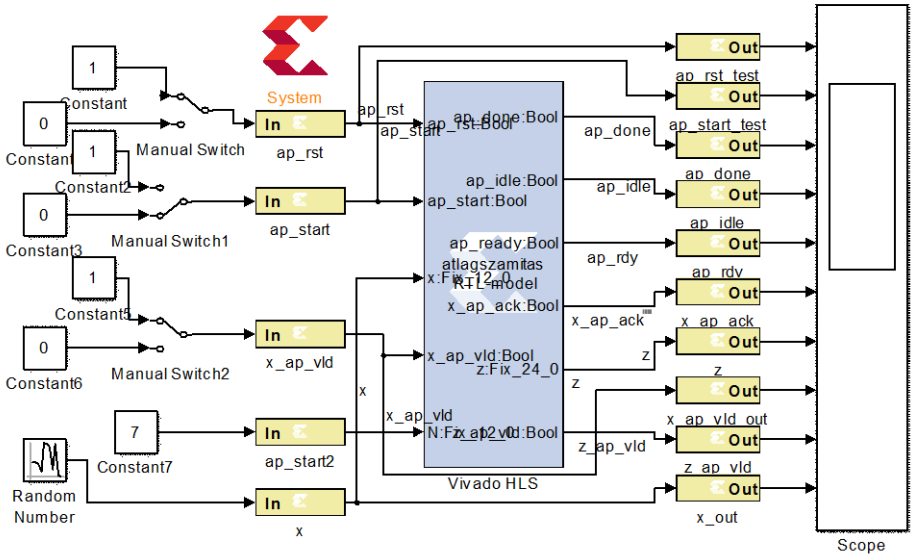

*12. Ábra Vektor elemei összegének a számítása. A kimeneti adat porthoz (z) regiszter van illesztve. A szimuláció során a tömb elemeinek a száma 7. A tömb értékei a ROM memóriában vannak eltárolva. Mivel a kimenethez regiszter van csatolva, a kimeneten nem a részösszeget, hanem egy lépéssel korábbi eredményt tárol a regiszter. Az ap_ready és ap_done vezérlő jelek egy óraciklust késnek a 12. ábrán bemutatott eredményekhez képest.*

Összehasonlítva a két szimuláció eredményét, a következő következtetéseket tudjuk levonni:

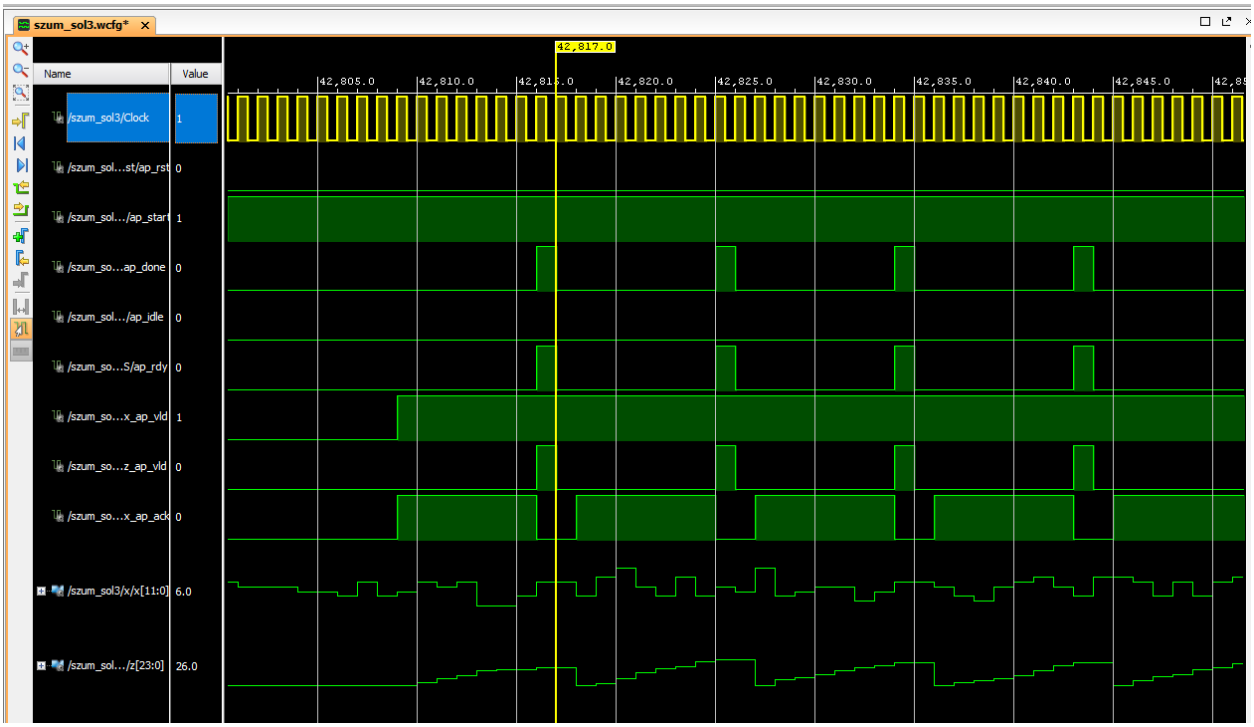
- az első változat szerint, amikor nincsen a kimenethez regiszter csatolva, a z kimeneten működés közben megjelenik a részleges eredmény is. A második változatban csak a végleges eredmény jelenik meg. A következő művelet végéig megmarad az azelőtti műveletvégzési ciklusból a művelet eredménye.
- második változatban, az utolsó memória olvasáshoz viszonyítva, egy órajellel később aktívak az ap_done, ap_ready, ap_vld jelek. Ebben az esetben a műveletvégzés egy óraciklussal többet tart.

Az alábbi példában az x bemeneti porthoz kézfogásos protokollt (ap_hs) rendeltünk. Ez alapján a tömb elemeit egyenként töltjük be a modulba. A protokoll alapján az X port jelhez az x_ap_vld és az x_ap_ack vezérlőjelek vannak csatolva. Az alábbi programrészben értelmezhető a programsor, amelyben az x bemenethez rendeltük az ap_hs protokollt.

```
void osszegszamitas(din_t x[M], din_t N, dout_t *z) {
#pragma HLS INTERFACE ap_hs port=x
  int tmp;
  int i;
  tmp=0;
  for (i=0;i<N;i++)
      tmp =tmp+x[i];
    *z=tmp;
  return ;
   }
```




*13. Ábra Tömb elemei összegének a számítása, létrehozott System Generator modul. Az x bemenethez ap_hs protokoll van rendelve. Vezérlőjelek x_ap_vld bemenet és x_ap_ack kimenet.*




*14. Ábra. Tömb elemei összegének a számítása, szekvenciálisan modulba betolt bemeneti adatok. Az ap_rst logiaki '0'-ra való kapcsolását követően a ap_start vezérlő jelet logikai '1'-re kapcsoljuk, majd az x_ap_vld jellel jelezzük, hogy az adatok készen állnak a beolvasásra Az x_ap_ack kimeneti jel logikai '0'-ra való változása nyugtázza, hogy az adatok beolvasása megtörtént. A beolvasást követő óraciklusban az ap_done, ap_ready ; ap_vld vezérlőjelek is egy óraciklus időre átváltanak logikai '1' szintre.*

A következő példákban szemléltetem azt, hogy ciklusra és tömbre kényszerfeltételeket alkalmazva, hogyan lehet optimalizálni, csökkenteni a számításhoz szükséges óraciklusok számát. Ha a bemeneti x tömböt két tömbre tördeljük, fele idő alatt kiszámolható a tömb elemeinek összege.

A megvalósításhoz a bemeneti tömböt kell feldarabolni két tömbre, és a ciklust ki kell fejteni (unrolling) úgy, hogy egy ciklus helyett párhuzamosan két ciklus legyen. Az egyik ciklus végzi az összegzést az egyik tömb elemeivel, a másik ciklusban pedig a másik tömb elemeire történik az összegzés.

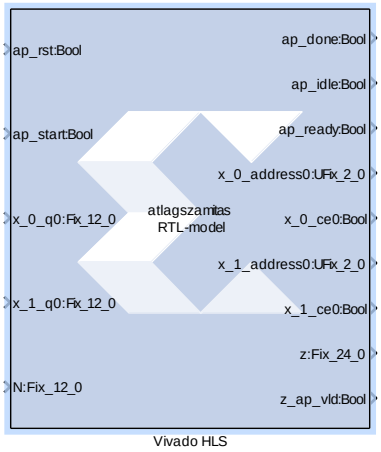
```
void osszegszamitas(din_t x[M], din_t N, dout_t *z) {
#pragma HLS ARRAY_PARTITION variable=x block factor=2 dim=1
  int tmp;
  int i;
  tmp=0;
  ciklus:for (i=0;i<N;i++)

#pragma HLS UNROLL factor=2
tmp =tmp+x[i];
    *z=tmp;
  return ;
}
```

A fenti program alapján két kényszerfeltétel van alkalmazva. Egy a bemeneti x tömb particionálására:
`#pragma HLS ARRAY_PARTITION variable=x block factor=2 dim=1`

valamint egy a ciklus kifejtésére:

```
#pragma HLS UNROLL factor=2
```

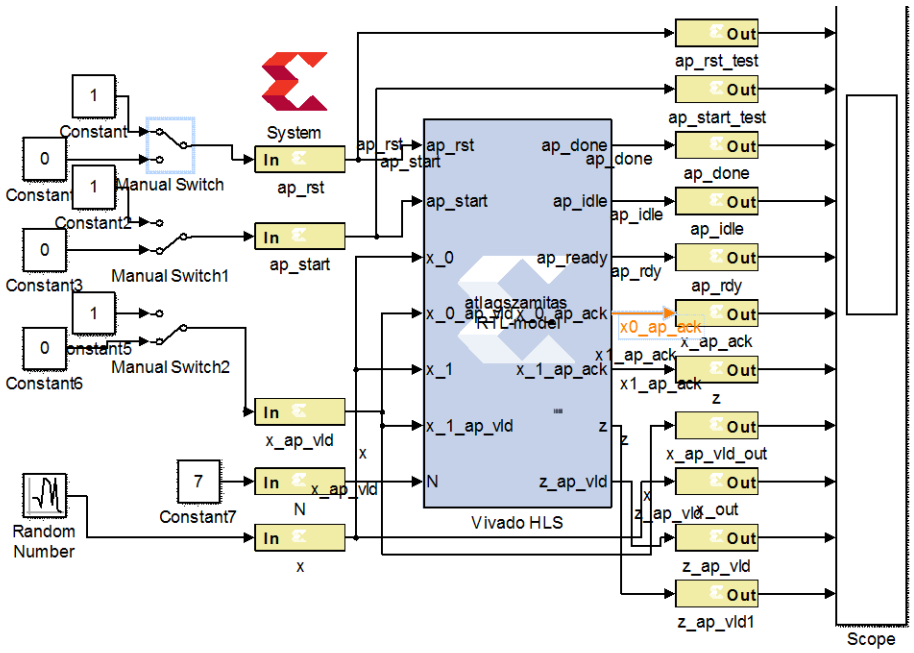
Eredményül a következő áramköri elemet kapjuk: egy-egy memóriát kell illeszteni az x_0 és x_1 bemenetekre. A memóriák vezérlésére az x_0_address, az x_0_ce, x_1_address ;s x_1_ce vezérlőjelek szolgálnak. A memória modulok illesztését a 11. ábra szerint kell megvalósítani.



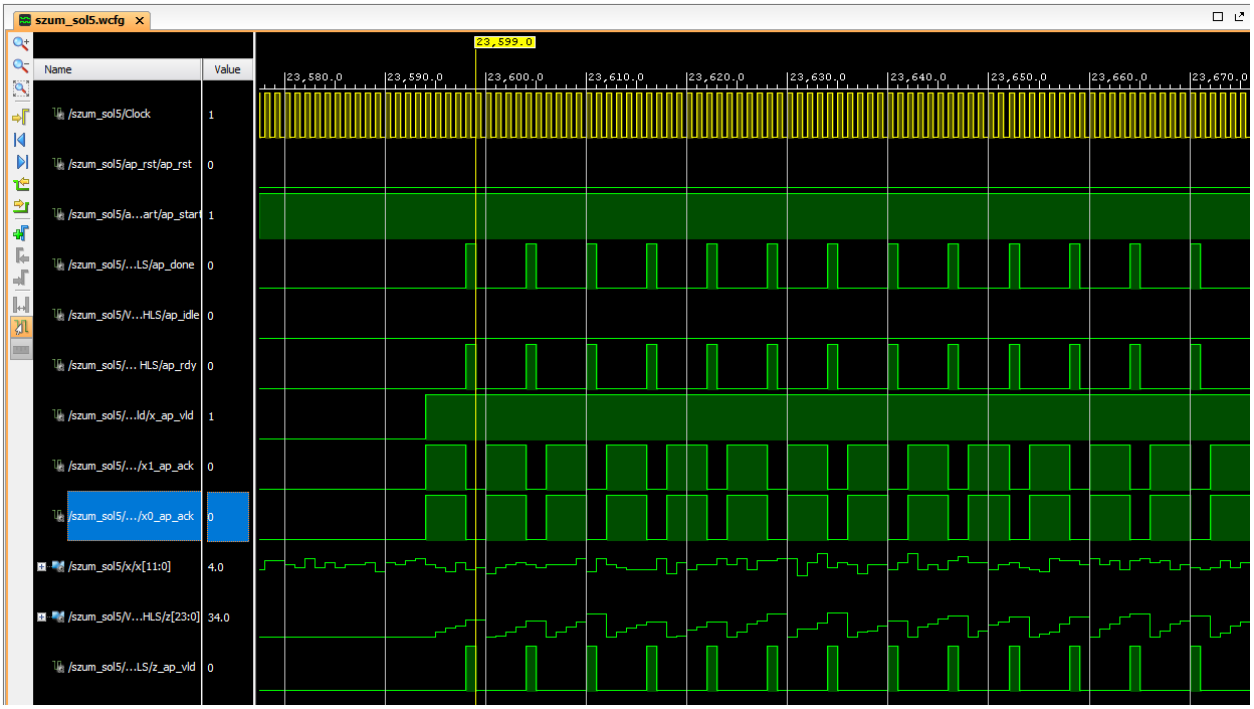

*15. Ábra Tömb elemei összegének számítása. Alapértelmezett függvény szintű protokoll, alapértelmezett port szintű interfészek (vagyis memória interfész ) az x bemenetre.*

Az alábbi programrész alapján az x tömb elemeit sorosan töltjük fel ap_hs protokollt alkalmazva. Az x tömböt felosztva két tömbre és ciklusra, összhangban a memória felosztásával, a megfelelő kényszerfeltételt szabjuk, felére csökkenthető az összegszámításhoz szükséges óraciklusok száma.

```
void osszegszamitas(din_t x[M], din_t N, dout_t *z) {
#pragma HLS ARRAY_PARTITION variable=x cyclic factor=2 dim=1
#pragma HLS INTERFACE ap_hs port=x
  int tmp;
  int i;
  tmp=0;
  ciklus:for (i=0;i<N;i++)

#pragma HLS UNROLL skip_exit_check factor=2
tmp =tmp+x[i];
    *z=tmp;
  return ;
}
```




*16. Ábra Particionált x bemeneti tömb (ap_hs) protokoll, két sínen szekvenciális adatbetöltés, közös ap_vld vezérlőjel mindkét bemenetre (x_0, x_1)*




*17. Ábra*

Az egyik ciklus végzi az összegzést az egyik tömb elemeivel, a másik ciklus pedig a másik tömb elemeivel. A tömb particionálása többféleképpen valósítható meg. Az egyik esetben, ha block típusú felosztást alkalmazzuk, a tömb elemeinek az első része az első tömbbe kerül, míg a második fele a második tömbbe. Jelen esetben a tömböt két kisebb tömbre particionáltuk.

Block paraméter alkalmazása a tömb feldarabolására:

```
#pragma HLS ARRAY_PARTITION variable=x block factor=2 dim=1
```


Ha a ciklikus (cyclic) típusú felosztást választjuk, jelen esetben a következő a felosztás:

```
pragma HLS ARRAY_PARTITION variable=x cyclic factor=2 dim=1
```


A különbség a cyclic és block paraméterek esetében a következő:

- blokk esetében az egymást követő elemek egy tömbbe kerülnek, míg a ciklikus felosztás során az első elem az első tömbbe, a második a második tömbbe, harmadik első tömbbe, negyedik második tömbbe stb.
- A ciklikus szétdarabolás lehetővé teszi, hogy a két kiterjesztett ciklus párhuzamosan működjön, ezáltal fele idő alatt elvégezve a műveleteket. A block típusú felosztás során a két ciklus egymás után végzi a számításokat

Összefoglaló

Könyvészet

https://www.xilinx.com/html_docs/xilinx2017_2/sdaccel_doc/topics/pragmas/ref-pragma_HLS_interface.html

https://www.so-logic.net/documents/knowledge/tutorial/Basic_HLS_Tutorial/sec2_developing_custom_ip_core_using_hls.html