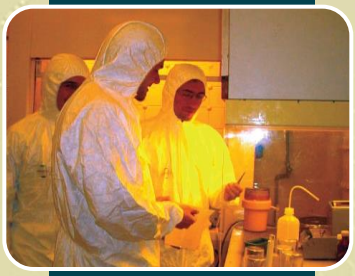
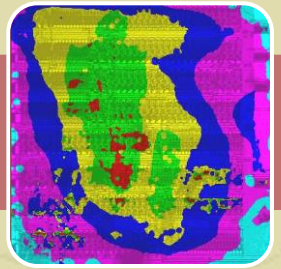
Budapesti Műszaki és Gazdaságtudományi Egyetem

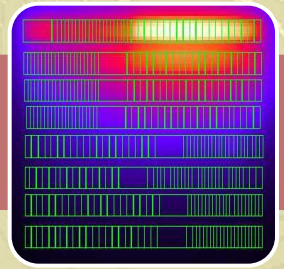
Elektronikus Eszközök Tanszéke

Magas szintű szintézis (High Level Synthesis)

Timár András







A hagyományos tervezési folyamat

- Indulás: **specifikáció**
- Lehet szöveges is, de általában C/C++/SystemC
- Tisztán funkcionális a leírás, nincsenek hardveres részletek
- Célja hogy ellenőrizze és finomhangolja a működést
- Tesztelés – leírás módosítása – tesztelés – módosítás … sok iteráció
- **Funkció: „mit” csinál a rendszer, architektúra: „hogyan” csinálja**
- Teljesítmény, fogyasztás és területigény definiálása
- **A tervezők lekódolják HDL nyelven (Verilog vagy VHDL)**

A tervezési folyamat problémái

- Mi a megfelelő architektúra?
- Mi az *optimális* architektúra?
- Probléma: ***kézi* megközelítés**
- Minden kézi beavatkozás hibák forrása lehet.
- A hibák „levadászása” sok idő

Elavult tervezési módszerek

- Egyre bonyolultabb rendszertervek
- Minden egyre bonyolultabb és kifinomultabb lesz: a minimális **csíkszélesség csökken**, az **órajelek gyorsulnak**, a beágyazott **magok száma növekszik**
- **Az RTL készítés folyamata a régi**
- A mai modern rendszereket (4G okostelefonok, H264 dekóderek, stb.) a ‘90-es évek közepéről származó eszközökkel akarjuk megtervezni

Magasabb szintű leírások

- **SystemC** vagy **Algorithmic C** (Mentor Graphics)
- Egyetlen algoritmikus C++ leírásból sok különböző RTL leírás születhet automatikusan, mindegyik más és más terület igénnyel és teljesítménnyel
- A kézi HDL kódolással ez napokat vehet igénybe, míg az új módszerrel ez teljesen automatizált és néhány másodpercig tart
- A magas szinten (pl. C++) leírt rendszer vagy algoritmus végül HDL nyelvű leírássá alakul (Verilog, VHDL)
- A C++ kód minősége nagyban befolyásolhatja a keletkező RTL leírás minőségét. Nagyon fontos, hogy a nem megfelelően megírt C++ kód rossz hatásfokú, rossz minőségű RTL kódot eredményez

Algoritmikus C

- A C++ kódból szintetizált HDL leírás nem úgy néz ki, ahogy azt kézzel írtuk volna
- Mégis optimálisabban működik, mint a kézi leírás
- A kézi leírás jóval hosszabb lenne.
- Lehet, hogy nem is jó elsőre
- A magas szintű szintézis (HLS) automatizál egy egyébként **manuális** folyamatot miközben több olyan lépést lehet kihagyni, mely hibákat vihet a rendszerbe.
- Gyorsítja az egyébként hosszú és iteratív fejlesztési ciklust

A magas szintű szintézis előnyei

- Hibamentes út az absztrakt specifikációktól az RTL leírásig.
- Sokkal rövidebb tervezési idő
- Egyszerűbb verifikáció
- **correct-per-construction RTL**
- A magas szintű leírás elkészítésekor sokkal kevesebb részlettel kell foglalkozni
- A funkcionális tervezéskor nem kell törődni a megvalósítás részleteivel: az **órajellel**, a **technológiával**, a **hierarchiával**
- Csak a kívánt viselkedéssel kell foglalkozni
- Egyszerűbb így elkészíteni a leírást. A kevesebb kóddal a **hibák esélye is lecsökken**. Egyszerűbb kimerítően tesztelni és verifikálni a tervet

Előnyök

- A magas szintű modellből az RTL készítés lépése teljesen automatikus!
- A hibalehetőségek megszűnnek, de megmarad a tervezői beavatkozás lehetősége
- Kényszerek beállítása a HLS eszközben → szintézer a megfelelő RTL kódot generálja
- A HLS eszköz automatizálja a hardver erőforrások kiosztását és ütemezését, kiépíti az adatutakat és vezérlési szerkezeteket
- Létrejön egy teljesen működőképes és optimális megvalósítás
- RTL leírás gyorsan elkészül és hibáktól mentes.
- Gyorsabb verifikáció, hibakeresés és tesztelés

Hatékony újrafelhasználhatóság

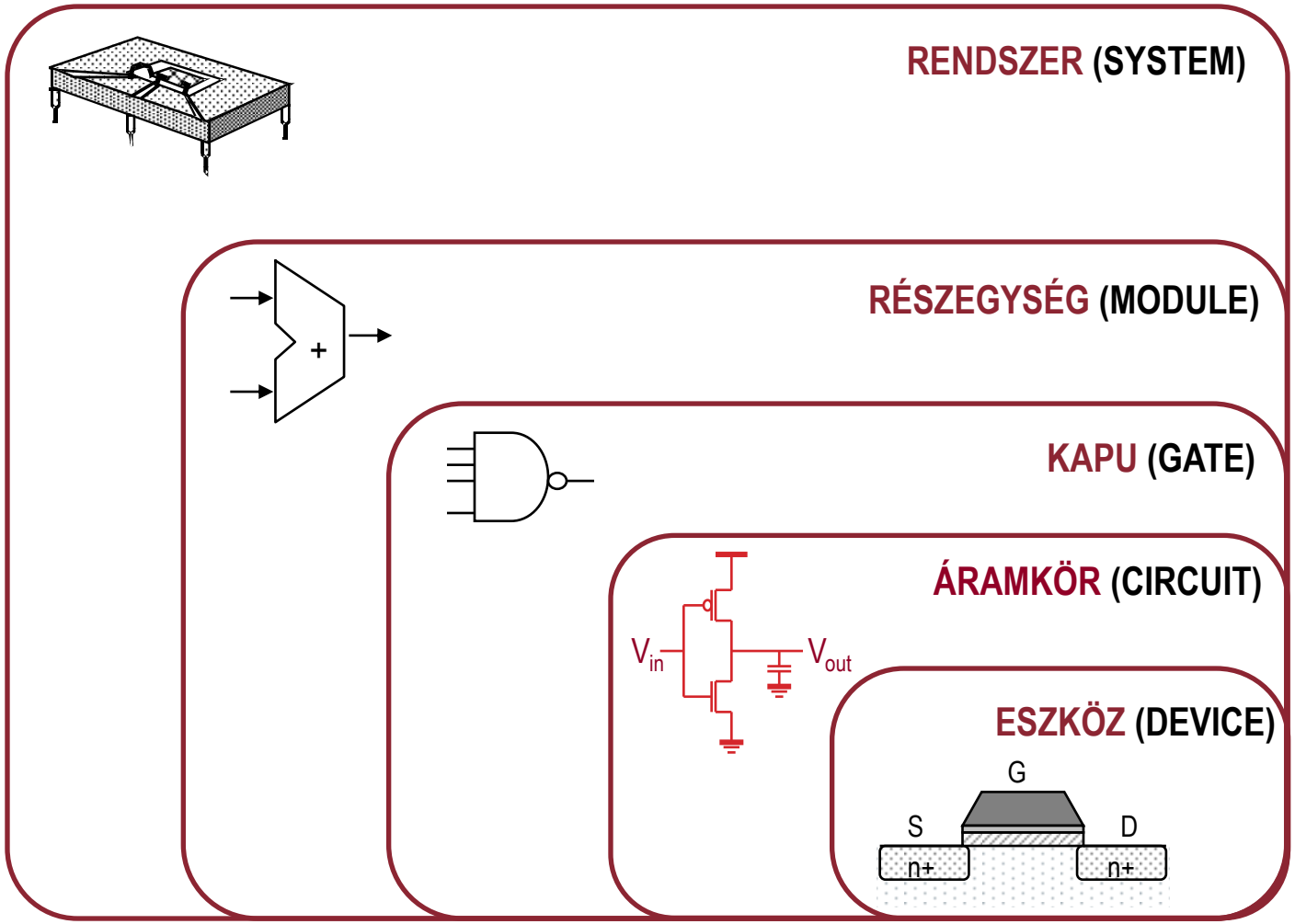
- Intellectual Property (IP)
- „Egyszer kitalálni, sokszor felhasználni”
- Az RTL IP-blokkok problémája:
  - Túlságosan alacsony szinten definiálják a működést.
  - A definíció **szorosan kapcsolódik** egy bizonyos **órajelhez** és **technológiához**
- Egy RTL leírás átültetése más órajelre és technológiára általában a terület és fogyasztás növekedésével és a teljesítmény csökkenésével jár
- Ha tisztán funkcionális specifikációkkal dolgozunk, akkor nincsenek olyan részletek, mint az órajel, technológia vagy mikro-architektúra
- Ezek az információk a magas szintű szintézis közben automatikusan kerülnek beépítésre
- A HLS használatával az IP-blokkok újrafelhasználása sokkal egyszerűbb

Kutatási és fejlesztési (K+F) erőforrások

- A HLS segítségével több hónapi kutatást és fejlesztést tudunk megspórolni
- Kevesebb erőforrást kell az RTL kódolásra és a verifikációra szánni
- Ha több erőforrás jut algoritmusok fejlesztésére, architektúra optimalizálásra, rendszerszintű teljesítmény-optimalizálásra az lényegesen befolyásolja a termék sikerét
- A „Time-to-market” mellett nagyon fontos a funkciógazdagság, az alacsony ár és a kis fogyasztás is

Tervezési szintek

- Visszatekintve: tranzisztorok → kapuk → RTL → Rendszer



Paradigmaváltás

- A magas szintű szintézis nem új ötlet
- Az ipar állandóan növekvő igényei miatt a minél magasabb szintű és egyszerűbb tervezés központi kérdés
- Az első magas szintű C szintézis eszközök 2004-ben jelentek meg
- Azóta rengetegszer bebizonyosodott, hogy ezeknek az eszközöknek óriási jelentősége van és, hogy a jövő rendszereit ilyen tervezőeszközökkel kell megalkotni
- Szükséges egy **paradigmaváltás**, akárcsak a **párhuzamos programozásban**

Általános C++ stílus

- Hogyan írjunk jó minőségű, szintetizálható kódot?
- Quality of Results (QoR)
- Egy általános C++ kód általában nem illeszthető be módosítás nélkül egy magas szintű hardver leírásba
- Fontos, hogy **hardver** készül a leírásból
- Meg kell tanulni a hardver szintézishez megfelelő programozási gyakorlatot

Ajánlott könyvtárstruktúra

- Példa egy ajánlott könyvtárstruktúrára
  - project
    - src
    - ccs
    - dat
    - sim
- project: az aktuális terv munkakönyvtára
- src: minden C/C++ forrásfájl, makefile, futtatható állományok (*.cpp, *.cxx, *.C, *.h, *.hpp, *.exe, Makefile)
- ccs: TCL fájlok a szintézishez
- dat: minden fájl I/O a teszt környezethez
- sim: Matlab és Simulink szkriptek

Makefile

- Unix/Linux alatt előszeretettel használjuk
- Automatizálja a fordítást
- Windows alatt is működik kis módosításokkal

```
# Example Makefile

#MACROS
CAT_HOME = $(MGC_HOME)
TARGET = my_tb
OBJECTS = main.o hello.o
DEPENDS = hello.h
INCLUDES = -
I"$(CAT_HOME)/shared/include"
DEFINES =
CXX = /usr/bin/g++
CXXFLAGS = -g -o3 $(DEFINES)
$(INCLUDES)

$(TARGET): $(OBJECTS)
      $(CXX) $(CXXFLAGS) -o $(TARGET)
$(OBJECTS)

$(OBJECTS): $(DEPENDS)

#phony target to remove all objetcs and
executables
.PHONY: clean
clean:
      rm -f *.o *.exe
```

Fejléc/Inlcude fájlok

- Függvény prototípusok
- Elődeklarációk
- Típusdefiniciók
- Konstans definíciók

```
//guard string to prevent multiple
inclusion
#ifndef __HELLO__
#define __HELLO__
//Forward declaration of fucntion
void hello();
const int ITERATIONS = 22;
typedef int dType;
#endif
```

Teszt környezetek (testbench)

- Magas szintű szintézisnél a C++ testbench egyaránt teszteli **a funkcionális C++ kódot** és a szintetizált **RTL leírást** is
- Nagyon fontos a jó programozási stílus
- Úgy kell megírni, hogy mindig az eredeti specifikációnak megfelelő funkcionalitást tesztelje, akárhogyan is változott a magas szintű kód
- A kód szintetizálhatóvá tétele nem szabad, hogy elrontsa a funkcionalitást
- Minden lépés után verifikálni kell a tervet
- Tehát: **testbench-et kell írni!**

Első prototípus

- Mindig legyen meg az első verzió a kódból
- Így lehet aztán később összehasonlítani a QoR-t teljesítmény és terület szempontjából
- Pl. egy meglévő fix- vagy lebegőpontos algoritmust módosítani kell jobb QoR elérése és/vagy szintetizálhatóság miatt
- **CatapultC**
- Minden változtatás egy revízió
- Vissza lehet követni minden változtatást (mint a verziókövetés)

Kód módosítása

- A két kód funkcionálisan egyforma
- Az új kód jobb teljesítményű, a működés párhuzamosítható

Eredeti kód

```
#include “test.h”
void test(dType a[2], dType b[2],
dType dout[2], bool sel){

if(sel){
  dout[0] = a[0] + b[0];
  dout[1] = a[1] + b[1];
}
else {
  dout[0] = a[0] - b[0];
  dout[1] = a[1] - b[1];}
}
```

Módosított kód

```
#include “test_mod.h”
void test_mod(dType a[2], dType b[2],
dType dout[2], bool sel){

for(int i=0;i<2;i++){
  if(sel){
    dout[i] = a[i] + b[i];
  }
  else{
    dout[i] = a[i] - b[i];
  }
 }
}
```

Módosított testbench

- Mindkét kódot teszteljük, ha megváltozott a funkcionalitás, hibát jelzünk
- Minden módosítás után futtassuk le a tesztet!

```
#include <iostream>
using namespace std;
#include “test.h”
#include “test_mod.h”
int main(){
  dType a[2] = {10, 20};
  dType b[2] = {10, 20};
  dType dout[2];
  dType dout_mod[2];
  bool sel = true;
  bool error = false;
  //DUT original
  test(a,b,dout,sel);
  //DUT modified
  test_mod(a,b,dout_mod,sel);
```

```
  for(int i=0;i<2;i++){
    if(dout_mod[i] != dout[i]){
      cout << “ERROR” << endl;
      error = true;
    }
    else
      cout << dout[i] << endl;
  }
  if(error)
    return -1; //indicates test failure
  else
    return 0; //test passed
}
```

Javított testbench

- Minden lehetséges kombinációt teszteljünk!
- Ellenkező esetben a kód nagyobb módosítása után derülhet fény a hibára
- Az előző példában csak **`sel=true`** feltételre teszteltünk

```
#include <iostream>
using namespace std;
#include “test.h”
#include “test_mod.h”
int main(){
  dType a[2] = {10, 20};
  dType b[2] = {10, 20};
  dType dout[2];
  dType dout_mod[2];
  bool sel = true;
  for(int j=0;j<2;j++){
    sel = j;
    //DUT original
    test(a,b,dout,sel);
    //DUT modified
    test_mod(a,b,dout_mod,sel);
    for(int i=0;i<2;i++){
      if(dout_mod[i] != dout[i])
        cout << “ERROR” << endl;
      else
        cout << dout[i] << endl;
    }
  }
}
```

Nem inicializált változók

- A nem inicializált változók nagy bajt okozhatnak
- „don't care"-nek minősíti a szintézer
- Minden változót inicializálni kell!
- **`tmp = (don't care) + din[i];`**

```
void acc(int din[4], int &dout){
  int tmp;
  for(int i=0;i<4;i++)
    tmp += din[i];

  dout = tmp;
}
```

Adott bithosszúságú típusok

- A C++ kódból hardver készül, ezért szükség van az adattípusok pontos bitméretének beállítására
- Angol terminológia: **Bit Accurate Data Type**
- Sok saját készítésű osztálykönyvtár létezik, de ezek csak szimulációkra alkalmasak, szintézisnél túl lassú kódot eredményeznek (rossz QoR)
- Két szabványos bit-accurate osztálykönyvtár létezik a piacon
  - A SystemC adattípusai
  - Mentor Graphics Algorithmic C adattípusok[1]
- Az első a **SystemC** volt, de túl lassúnak bizonyult futásidőben ezért a Mentor kifejlesztette a saját adattípusát
- Az **Algorithmic C** nemcsak gyorsabb szimulációt tesz lehetővé, de jobb QoR is érhető el vele (jobb HDL készül)

[1] http://www.mentor.com/products/esl/high_level_synthesis/ac_datatypes

Fordítás, hibakeresés, szimuláció sebesség

- Header fájlok

```
#include <ac_int.h>
#include <ac_fixed.h>
```

- Szimulációnál érdemes beállítani legjobb optimalizálást (-O3)

```
g++ -O3 -I<path to Alogrithmic C data types> -o
hello.exe hello.cpp
```

- Hibakeresésnél -O0

```
g++ -g -Wall -I<path to Alogrithmic C data types>
-o hello.exe hello.cpp
```

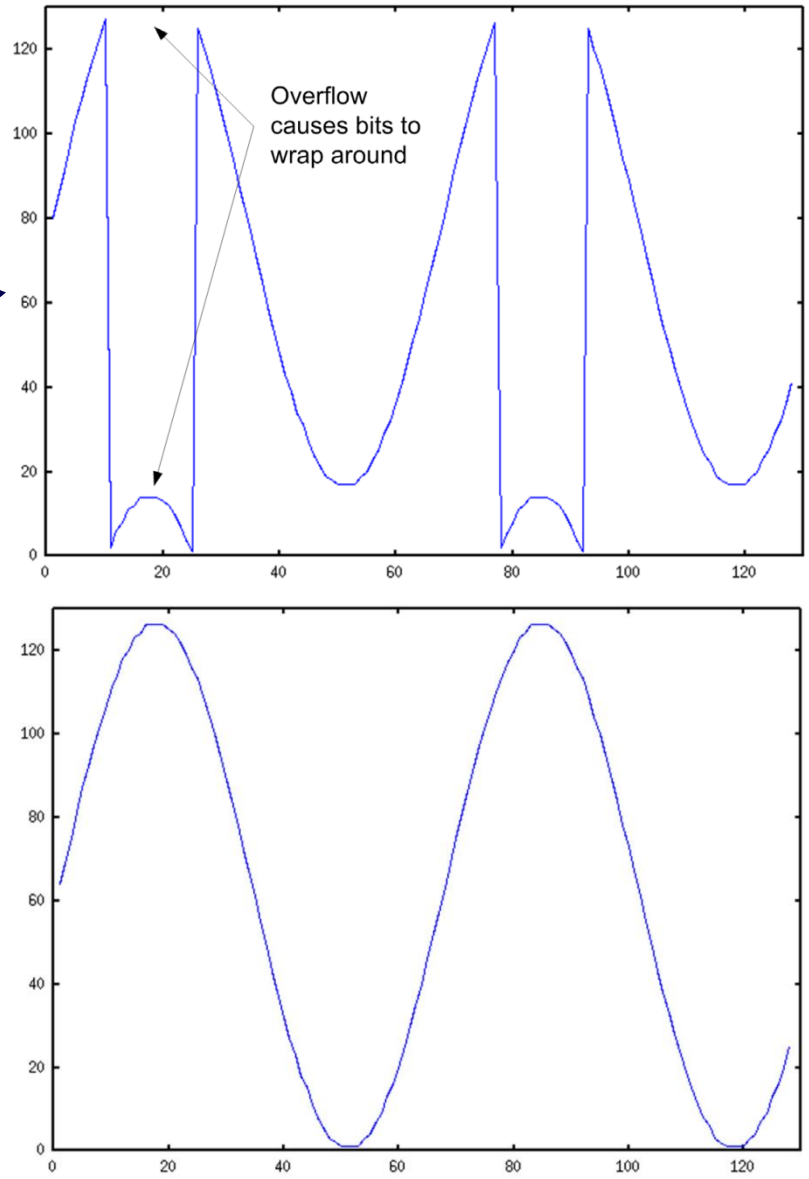
- Szimulációnál érdemes lehet áttérni natív típusokra a sebesség miatt

```
#ifndef __TYPEDEFS__
#define __TYPEDEFS__
#include <ac_int.h>
#ifdef NATIVE_TYPES
  typedef short int dType;
  typedef int oType;
#else
  typedef ac_int<7,true> dType;
  typedef ac_int<14,true> oType;
#endif
#endif
```

```
#include “typedefs.h”
void test(dType a, dType b, oType &c){
  oType tmp;
  tmp = a*b;
  c = tmp;
}
```

Egész adattípusok (integers)

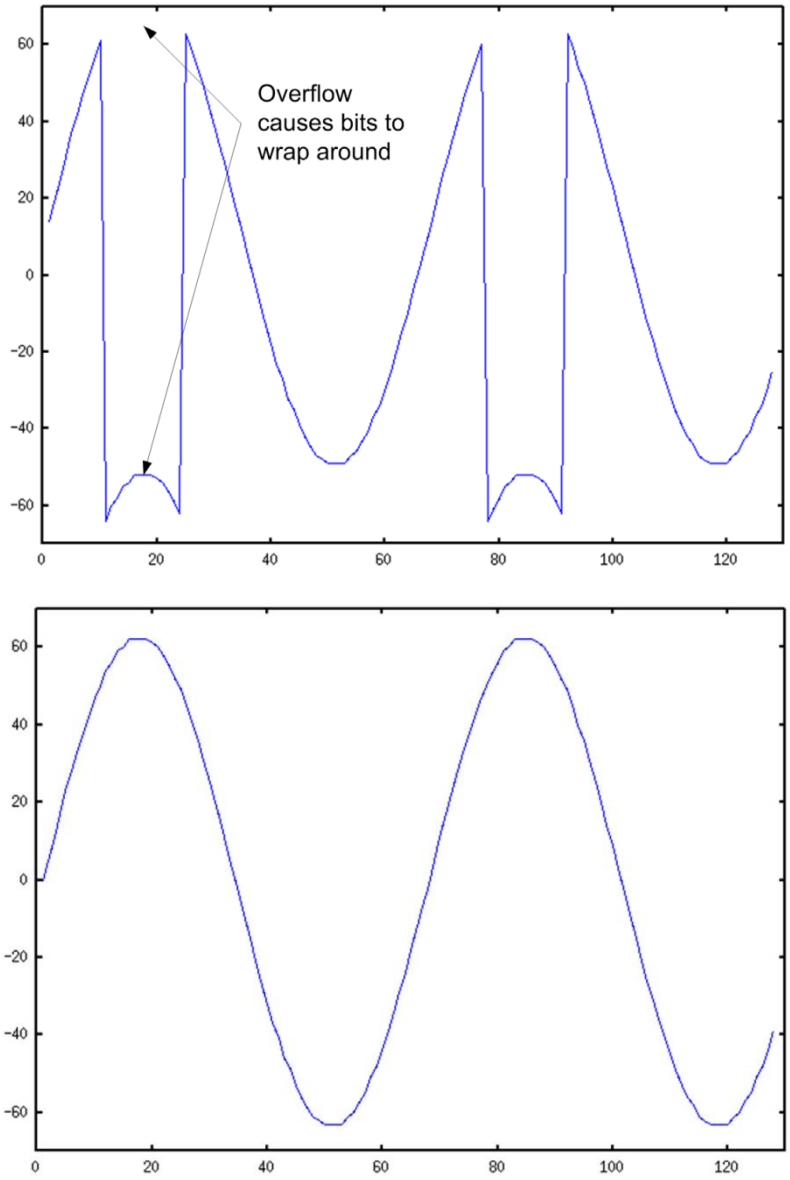
- Unsigned integer
  - `ac_int<W,false> x;`
  - `W: bitszélesség`
  - `0 <= x <= 2^W-1`
  - „körbefordulás”




```
#include <ac_int.h>
#include <fstream>
#include <cmath>
using namespace std;
const double pi = 3.14;
const int OFFSET = 64;
int main(){
  fstream fptr;
  fptr.open(“tmp.txt”, fstream::out);
  ac_int<7,false> x[128];
  for(int i=0;i<128;i++){
    x[i] = OFFSET + 63*sin(2*pi*i/64);
    fptr << x[i] <<endl;
  }
  fptr.close();
}
```

Egész adattípusok (integers)

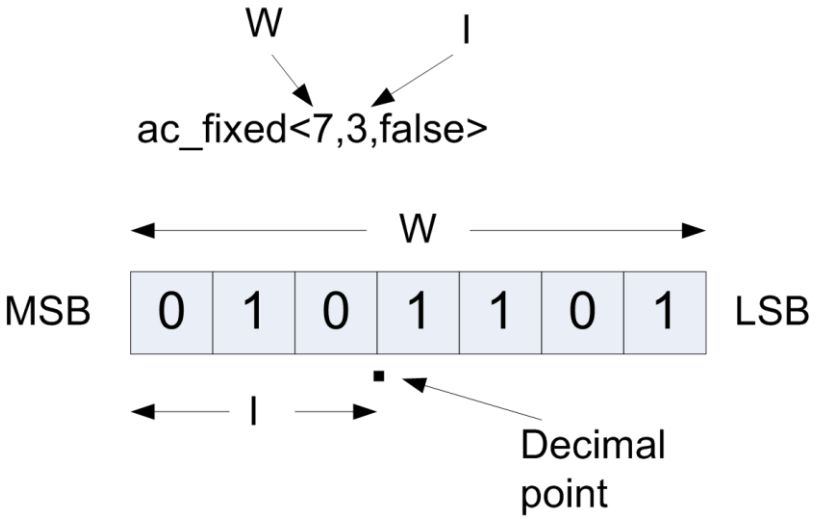
- Signed integer
  - `ac_int<W,true> x;`
  - `W: bitszélesség`
  - $-2^{W-1} <= x <= 2^{W-1}-1$
  - „körbefordulás”



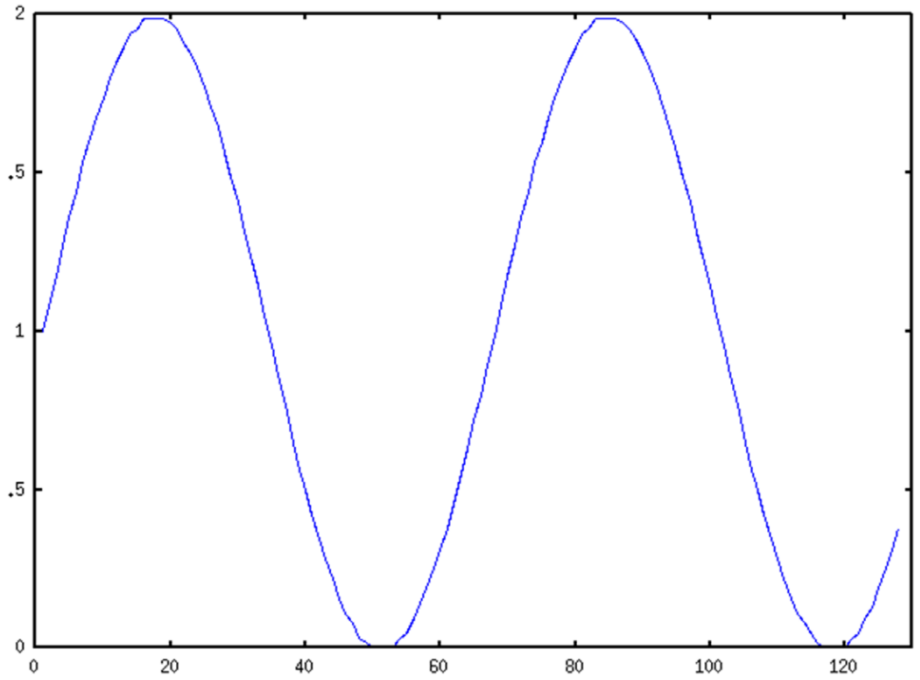

```
#include <ac_int.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>
const double pi = 3.14;
const int OFFSET = 0;
int main(){
  fstream fptr;
  fptr.open(“tmp.txt”, fstream::out);
  ac_int<7,true> x[128];
  for(int i=0;i<128;i++){
    x[i] = OFFSET + 63*sin(2*pi*i/64);
    fptr << x[i] <<endl;
  }
  fptr.close();
}
```

Fixpontos adattípusok

- Unsigned fixed point
  - `ac_fixed<W,I,false> x;`
  - `W: teljes bitszélesség`
  - `I: egész bitek száma`
  - $0 <= x <= (1-2^{-W})2^{I}$



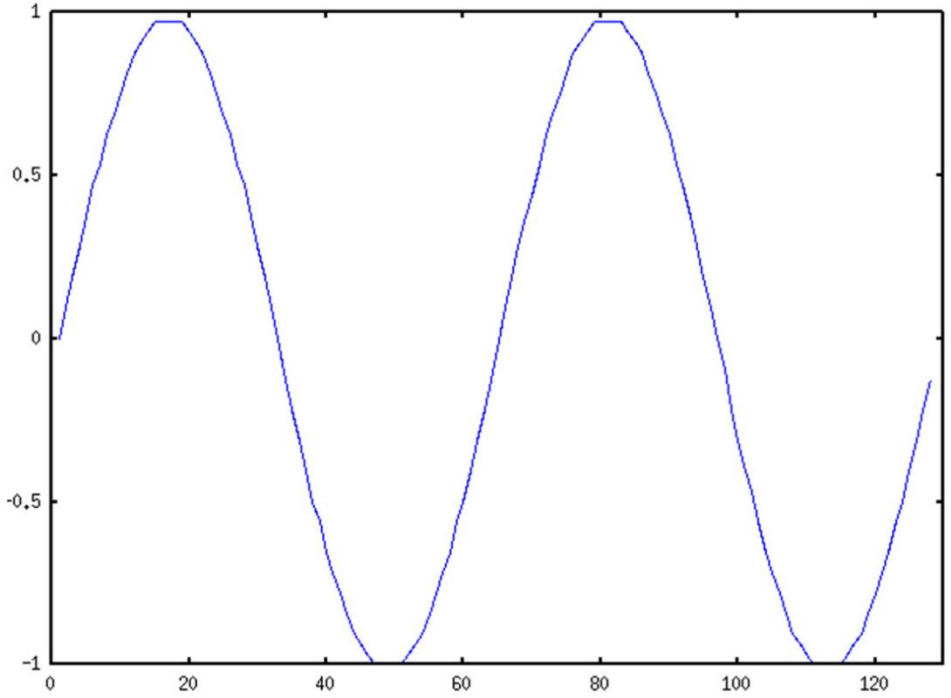

```
#include <ac_fixed.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>
const double pi = 3.14;
const double OFFSET = 1.0;
int main(){
  fstream fptr;
  fptr.open("tmp.txt", fstream::out);
  ac_fixed<7,1,false> x[128];
  for(int i=0;i<128;i++){
    x[i] = OFFSET + 0.98*sin(2*pi*i/64);
    fptr << x[i] <<endl;
  }
  fptr.close();
}
```



Fixpontos adattípusok

- Signed fixed point
  - ac_fixed<W,I,true> x;
  - W: teljes bitszélesség
  - I: egész bitek száma
  - $-0,5*2^I <= x <= (0,5-2^{-W})2^I$

```
#include <ac_fixed.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>
const double pi = 3.14;
const double OFFSET = 0.0;
int main(){
  fstream fptr;
  fptr.open("tmp.txt", fstream::out);
  ac_fixed<7,1,true> x[128];
  for(int i=0;i<128;i++){
    x[i] = OFFSET + 0.98*sin(2*pi*i/64);
    fptr << x[i] <<endl;
  }
  fptr.close();
}
```



Kvantálás és túlcsordulás

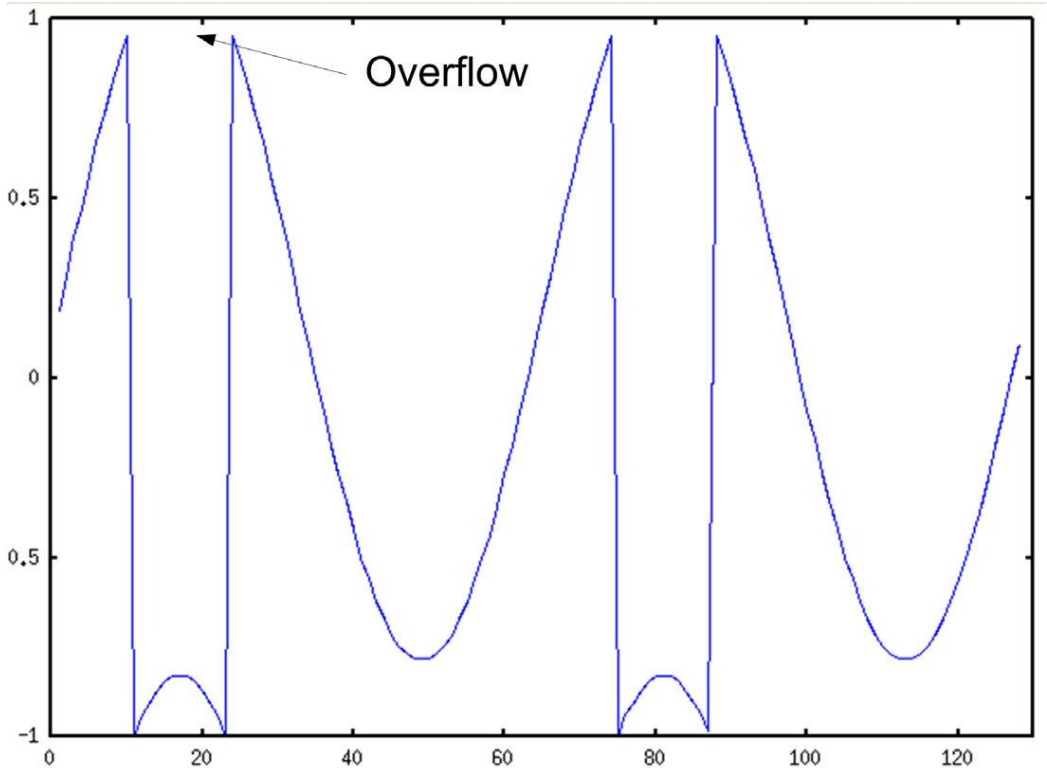
- A fixpontos adattípusok több lehetőséget is biztosítanak a túlcsordulások és a kvantálás finomhangolására
- Alapértelmezetten úgy működik, mint az **`ac_int<>`**
- Az alapértelmezett beállítás nem foglal több helyet
- Az alapértelmezett beállítás nem mindig megfelelő
- **`ac_fixed<W,I,Q,O> x`**
- Q: quantization
- O: overflow

Csonkolás és kerekítés

- Alapértelmezés: csonkolás és LSB-től jobbra lévő bitek eldobása
- Pl:
  - `ac_fixed<7,7,true> x = 0.5;`
  - `x`-nek nincs tört bitje, `x == 0`
- Lehet kerekíteni
  - `ac_fixed<7,7,true,AC_RND> x = 0.5;`
  - `AC_RND` $+\infty$ felé kerekít
  - `X == 1`

Telítés és túlcsordulás

- A bitek „átfordulása” sokszor nemkívánatos
- Szükség van az átfordulás megakadályozására
- Pl.

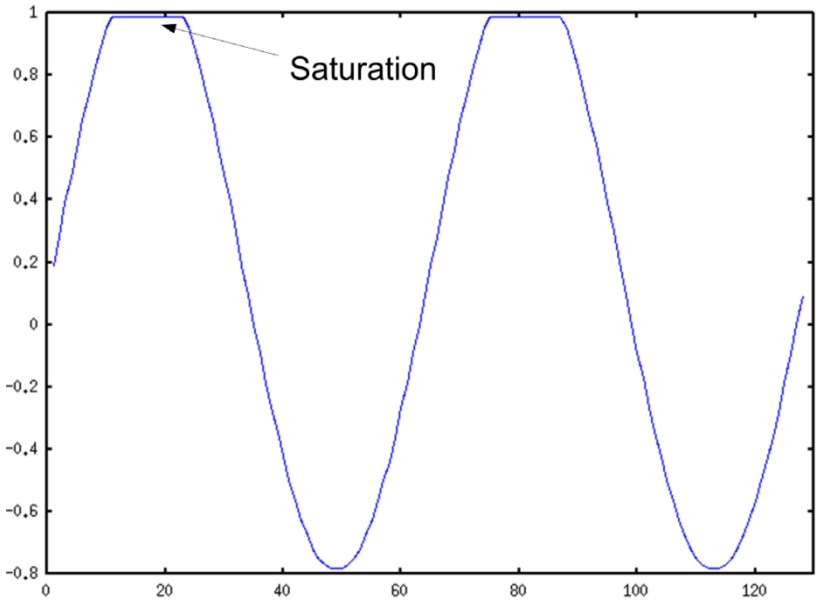


- Ez kritikus körülmények között végzetes lehet

Telítés és túlcsordulás

- Engedélyezzük a telítést a fixpontos típusra
- Csak a legszükségesebb esetben használjuk!
- Zajt visz a rendszerbe és megnöveli a helyfoglalást

```
#include <ac_fixed.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>
const double pi = 3.14;
const double OFFSET = 0.2;
int main(){
  fstream fptr;
  fptr.open("tmp.txt", fstream::out);
  ac_fixed<7,1,true,AC_TRN,AC_SAT> x[128];
  for(int i=0;i<128;i++){
    x[i] = OFFSET + 0.98*sin(2*pi*i/(double)64);
    fptr << x[i] <<endl;
  }
  fptr.close();
}
```



Operátorok

- Minden C++ operátor rendelkezésre áll az **`ac_fixed<>`** és **`ac_int<>`** típusokra
- A moduló (%) és osztás (/) operandusok is, de ezekkel óvatosan
- Nagy területet foglal és vannak olcsóbb megoldások is, mint pl. a CORDIC algoritmus
- A HLS eszközök nagy része támogatja ezeket
- Moduló vagy osztás konstanssal már elfogadható
- Moduló vagy osztás 2 egész többszörösével OK (shift)

Aritmetikai operátorok

- *, +, -, /, &, |, ^, %
- Figyeli a bitszélesség növekedést és az előjeles/előjel nélküli operandusokat is kezeli
- Pl.
  - `ac_int<8,true> a,b; // 8-bits unsigned`
  - a*b **`ac_int<16,true>`** típussal tér vissza, mert két 8 bites szám szorzatához 16 bit szükséges
    - Miért? Pl. $11_{bin} * 11_{bin} = 3_{dec} * 3_{dec} = 9_{dec} = 1001_{bin}$
    - 2 bit → 4 bit
  - a+b **`ac_int<9,true>`** típussal tér vissza, mert két 8 bites szám összegéhez 9 bit szükséges

Bit kiválasztó operátor ([])

```
ac_int<11,true> x;
bool is_neg;
if(x[10]) //test for sign bit treated as bool
  is_neg = true;
else
  is_neg = false;
```

```
ac_fixed<9,1,false> x = 0;
bool add_one = true;
if(add_one)
  x[8] = 1; //set MSB of x
```

Léptető operátorok

- A baloldali operandus pontosságával térnek vissza!
- Unsigned shift right (>>)

```
ac_int<4,false> x = -1; //set all bits to 1's
int idx;
ac_int<4,false> y = x >> idx;
```

y	idx	Value of y
1 1 1 1	0	15
0 1 1 1	1	7
0 0 1 1	2	3
0 0 0 1	3	1
0 0 0 0	4	0

Léptető operátorok

- Signed shift right (>>)
- Előjeles változó jobbra léptetése megtartja az előjelet!

```
ac_int<4,true> x = 0;
x[3] = 1 //set x equal -8
int idx;
ac_int<4,true> y = x >> idx;
```

y	idx	Value of y
1 0 0 0	0	-8
1 1 0 0	1	-4
1 1 1 0	2	-2
1 1 1 1	3	-1
1 1 1 1	4	-1

Léptető operátorok

- Unsigned shift left (<<)

```
ac_int<4,false> x = -1; //set all bits to 1's
int idx;
ac_int<4,false> y = x << idx;
```

y	idx	Value of y
1 1 1 1	0	15
1 1 1 0	1	14
1 1 0 0	2	12
1 0 0 0	3	8
0 0 0 0	4	0

Léptető operátorok

- Signed shift left (<<)

```
ac_int<4,true> x = -1; //set all bits to 1's
int idx;
ac_int<4,true> y = x << idx;
```

y	idx	Value of y
1 1 1 1	0	-1
1 1 1 0	1	-2
1 1 0 0	2	-4
1 0 0 0	3	-8
0 0 0 0	4	0

Nem várt pontosság csökkenés

- Balra léptetésnél következhet be, nem várt, de korrekt

```
ac_int<4,false> x = -1; //set all bits to 1's
int idx;
ac_int<8,false> y = x << idx;
```

- Egy 4 bites változót léptetünk balra és az eredmény egy 8 bites változóba kerül. Az eredmény:

y	idx	Value of y
0 0 0 0 1 1 1 1	0	15
0 0 0 0 1 1 1 0	1	14
0 0 0 0 1 1 0 0	2	12
0 0 0 0 1 0 0 0	3	8
0 0 0 0 0 0 0 0	4	0

Nem várt pontosság csökkenés

- Emlékeztető: a léptetés pontossága mindig a bal oldali operandus pontossága

```
ac_int<4,false> x = -1; //set all bits to 1's
int idx;
ac_int<8,false> y = x << idx;
```

- Megoldás: **cast**olás 8 bitesre

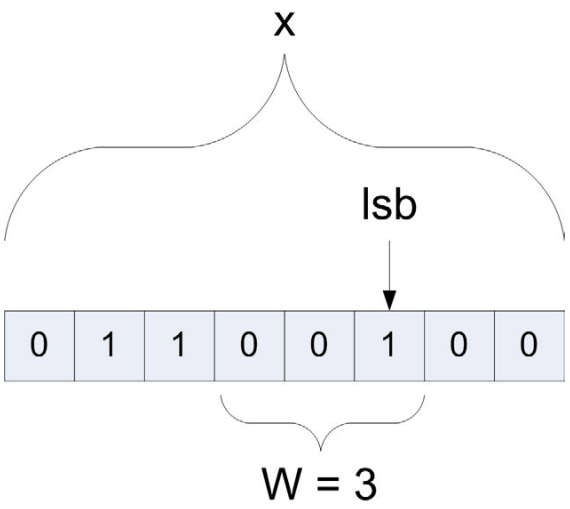
```
ac_int<4,false> x = -1; //set all bits to 1's
int idx;
ac_int<8,false> y = (ac_int<8,false>) x << idx;
```

y	idx	Value of y
0 0 0 0 1 1 1 1	0	15
0 0 0 1 1 1 1 0	1	30
0 0 1 1 1 1 0 0	2	60
0 1 1 1 1 0 0 0	3	120
1 1 1 1 0 0 0 0	4	240

Metódusok

- Szelet olvasás (slice read)
  - `slc<W>(int lsb)`
  - W: a szelet szélessége
  - lsb: a szelet kezdete (hányadik bit)

```
ac_int<8,false> x = 100;
ac_int<3,false> y;
y = x.slc<3>(2);
```



Metódusok

- Szelet írás (slice write)
  - `set_slc(int lsb, const ac_int<W,S> &slc)`
  - W: a szelet szélessége
  - S: előjeles (true)/előjel nélküli (false)
  - lsb: a szelet kezdete (hányadik bit)

```
ac_int<8,false> x = 0;
ac_int<3,false>       y
= 5;
x.set_slc(2,y);
```

Explicit konverzió

- Tipikusan akkor kell, ha
  - pl. **`ac_fixed<>`** típust akarunk értékül adni egy **`ac_int<>`** típusnak
  - printf() hívással ki szeretnénk írni a tartalmát

```
ac_fixed<8,3,false> x = 3.185;
ac_int<8,false> y;
y = x; // Compiler error!
```

- Megoldás: `y = x.to_int();`
- Futási időben `printf()`

```
printf("%d\n",y); // Segmentation fault!
printf("%d\n",y.to_int()); // OK!
```

Segítő funkciók

- **`ac::init_array`**
  - Tömb inicializálása és „un-inicializálása”
  - A szintézer így tudja, hogy nem kell hardvert szintetizálni egy **`for()`** ciklushoz, ami feltölti vagy „don’t care”-be teszi a tömböt

```
bool ac::init_array<"init value">("base address of array", "number of elements");
```

```
static int a[1000];
static bool dummy =
ac::init_array<AC_VAL_DC>(a,1000);
```

- **`ceil(), floor() és nbits()`**
  - Pl. a **175-ös** szám ábrázolásához szükséges bitek száma
    - $2^x = 175 \rightarrow x = \lceil \log_2 175 \rceil$

```
const int WORDS = 175;
void foo(int data[WORDS], ac_int<ac::log2_ceil<WORDS>::val,false> idx, int &dout){
  dout = data[idx];
}
```

- Komplex adattípusok: **`ac_complex<>`**

A HLS alapjai

- Nem mindegy, hogy milyen stílusban írjuk meg a C++ kódot
- Hardver készül!
- Egy rosszul megírt C++ kód szub-optimális RTL-t eredményez, ha egyáltalán szintetizálható
- A javasolt kódolási stílust kell használni
- Figyelni kell a memória architektúrára, stb.

Top modul

- Meg kell mondani a HLS eszköznek, hogy melyik modul a hierarchia teteje
- Mint Verilogban, vagy VHDL-ben
- Interfész a külvilággal (portok, irányaik, szélesség, stb.)

D FF aszinkron reset (Verilog)

```
module top(clk,arst,din,dout);
input clk;
input arst;
input [31:0] din;
output [31:0] dout;
reg [31:0] dout;
always@(posedge clk or posedge arst)
begin
  if(arst == 1'b1)
    dout = 1'b0;
  else
    dout = din;
end
endmodule
```

C++

```
#pragma design top
void top(int din, int& dout){
  dout = din;
}
```

Alapértelmezett viselkedés

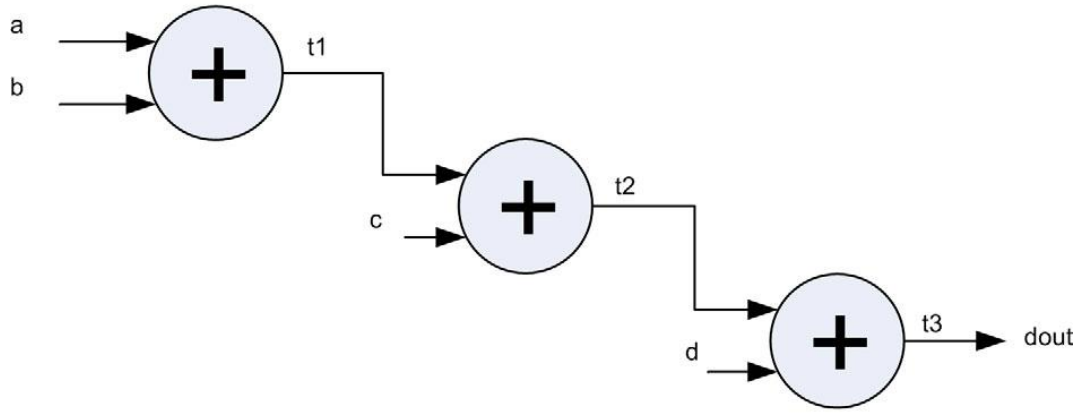
- Regiszter kimenetek
  - Minden szinkron, a kimenetek regiszterek
- Vezérlő portok
  - Nincs időzítés, órajel, enable, reset. Ezeket a szintézer adja hozzá és kényszerekkel állíthatók be.
- Port szélességek
  - Az adattípus határozza meg (pl. int = 32 bit, ac_int<8,true>)
- Port iránya
  - A kódból kiderül
- Bemenő portok
  - Ha egy interfész portot **csak olvasunk**, az **bemeneti** port lesz
- Kimenő portok
  - Ha egy interfész portot **csak írunk**, az **kimeneti** port lesz (referencia)
- Kétirányú portok
  - Írjuk és olvassuk is

Magas szintű C++ szintézis

- 4 számot összadó áramkör

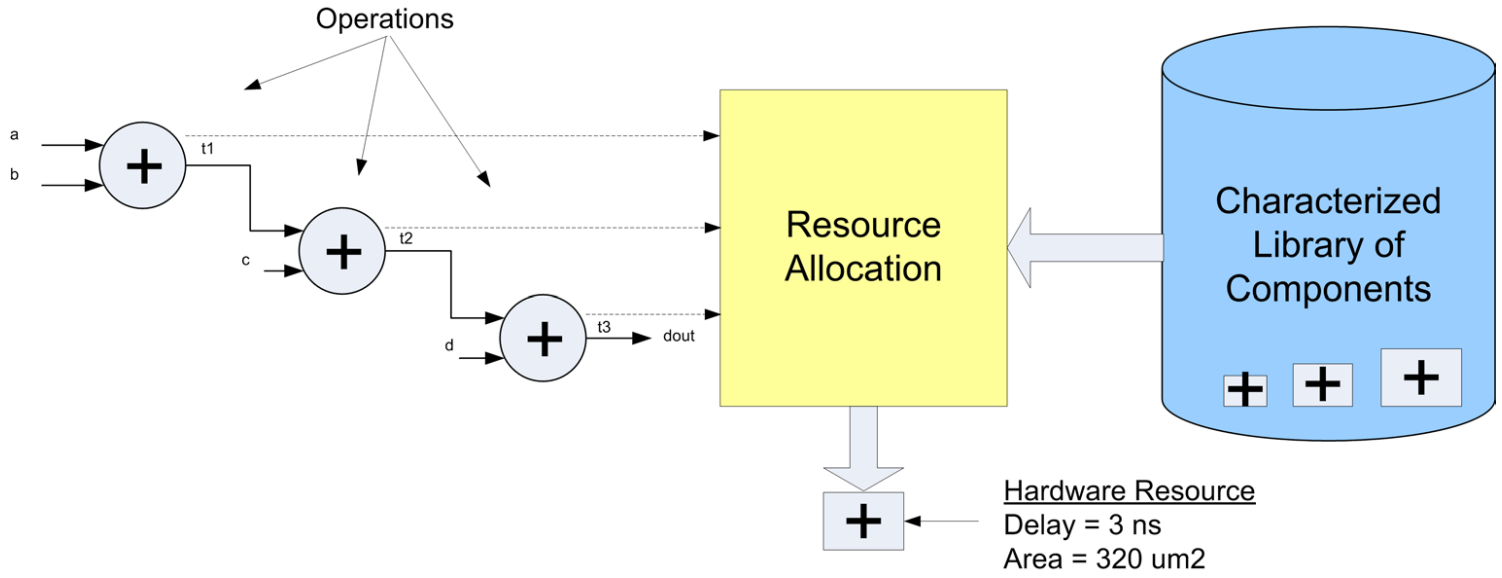
```
#include “accum.h”
void accumulate(int a, int b, int c, int d, int &dout){
  int t1,t2;

  t1 = a + b;
  t2 = t1 + c;
  dout = t2 + d;
}
```

- Adatfolyam gráf analízis (Data Flow Graph, DFG)
  - Minden csomópont egy utasításnak felel meg
  - A kapcsolatok meghatározzák a sorrendet (t2 előtt t1)



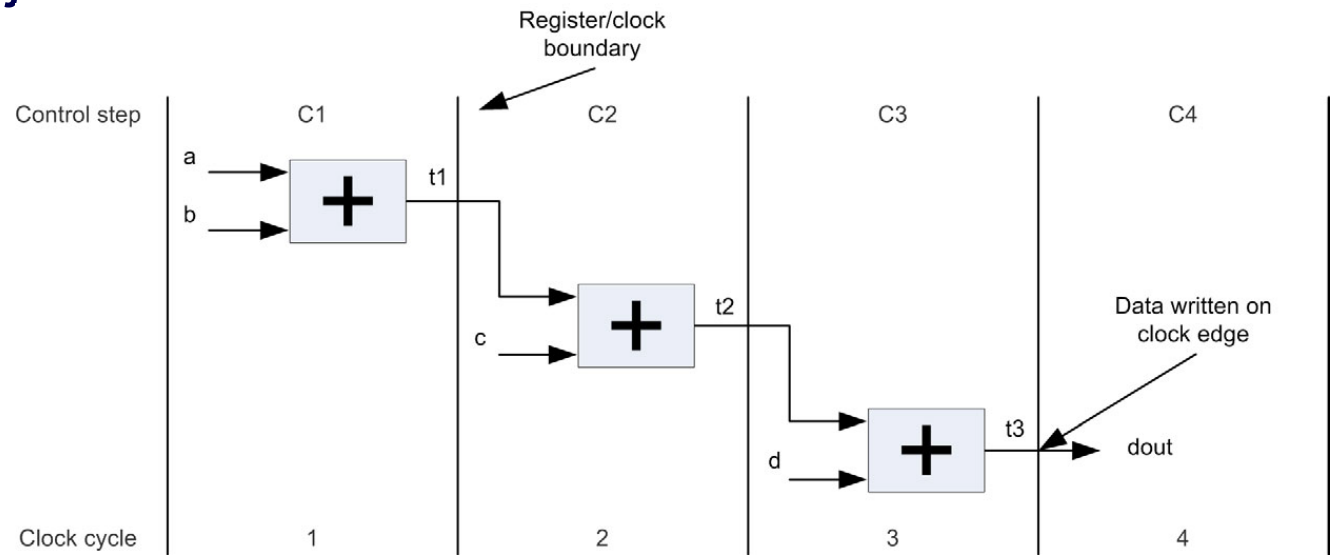
Erőforrás kiosztás

- Ha megvan a DFG, minden művelet egy HW erőforrásra képződik le
- Erőforrás == hardver implementáció
- Időzítés és terület adatok (több fajta erőforrás ua. típus)
- Különböző időzítés, terület, késleltetés, lappangás, stb.
- A szintézer egy előre karakterizált könyvtárból válogatja ki az erőforrásokat a kényszerek alapján



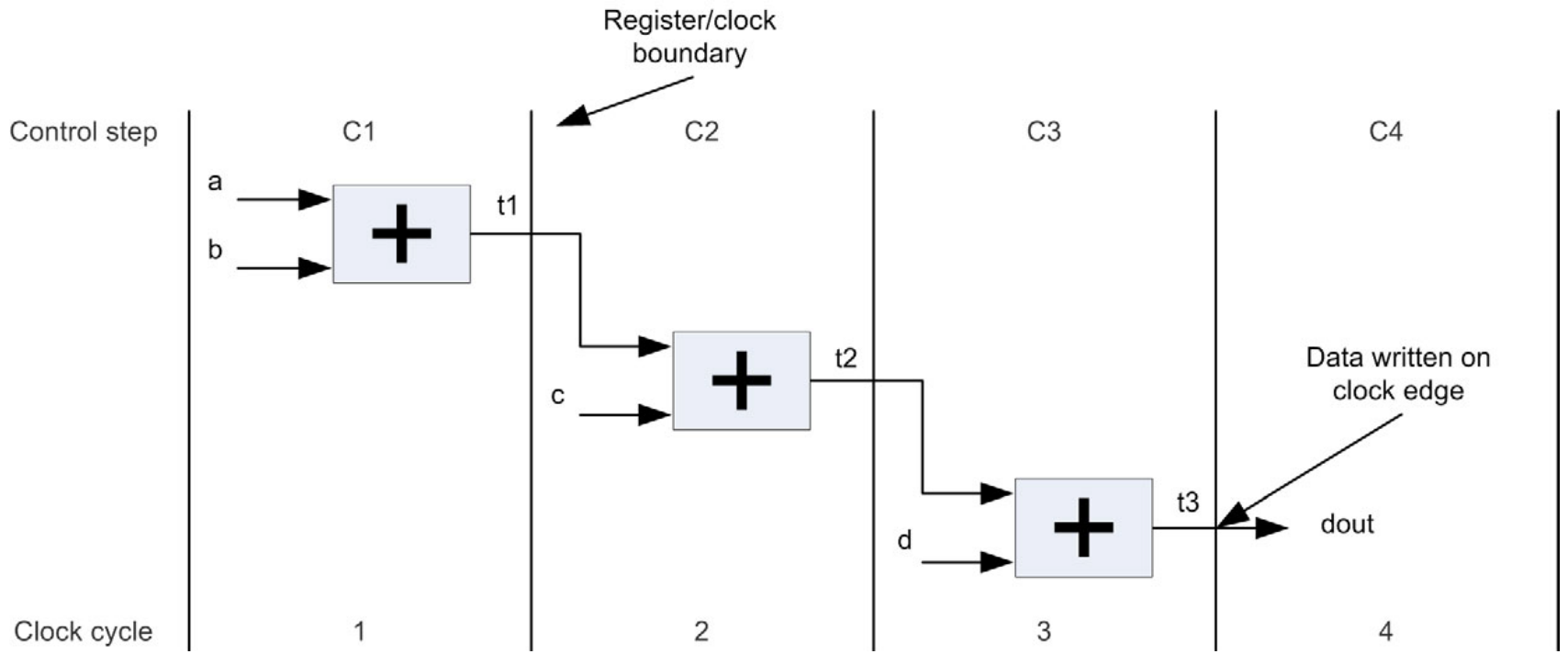
Ütemezés

- Megjelenik az „idő” fogalma az ütemezés lépésben
- Az ütemezés eldönti, hogy a DFG műveletei mikor (melyik órajel ciklusban) hajtódjanak végre
- Ez regiszterek hozzáadását jelenti a cél órajel frekvencia alapján
- Az RTL tervezők ezt hívják **pipeline**-osításnak
- Tegyük fel, hogy az „összadás” művelet 3ns-ig tart egy 5ns-os órajel ciklusnál



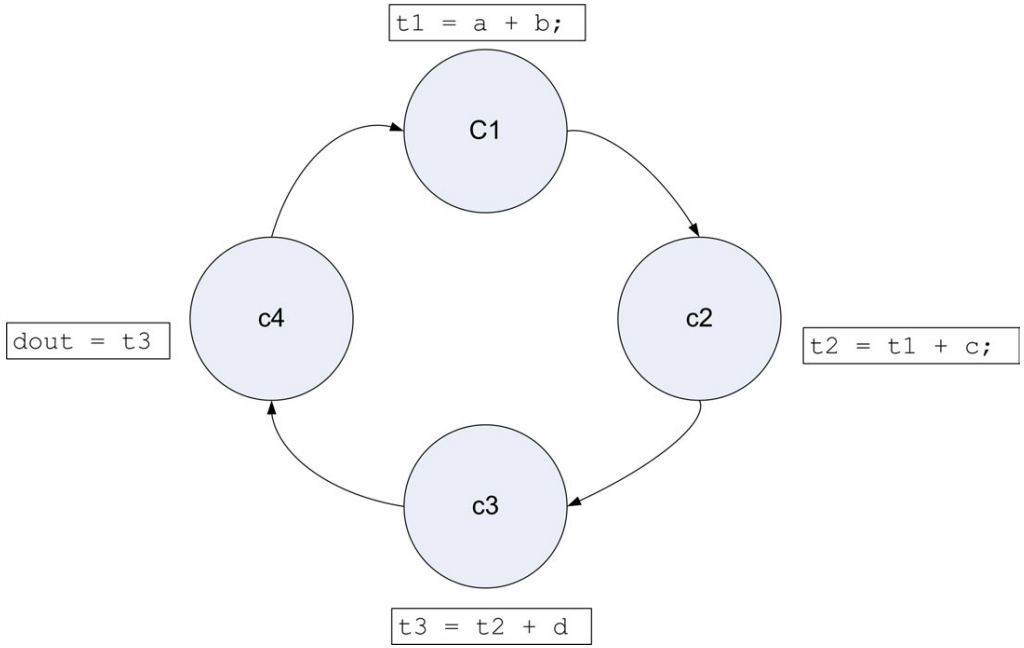
Ütemezés

- Minden „összeadás” művelet a saját órajel ciklusába kerül (C1,C2,C3)
- Minden összadó közé automatikusan egy regiszter kerül
- C4 azért kell, mert itt tudjuk kiolvasni először eredményt (szinkron hálózat!)



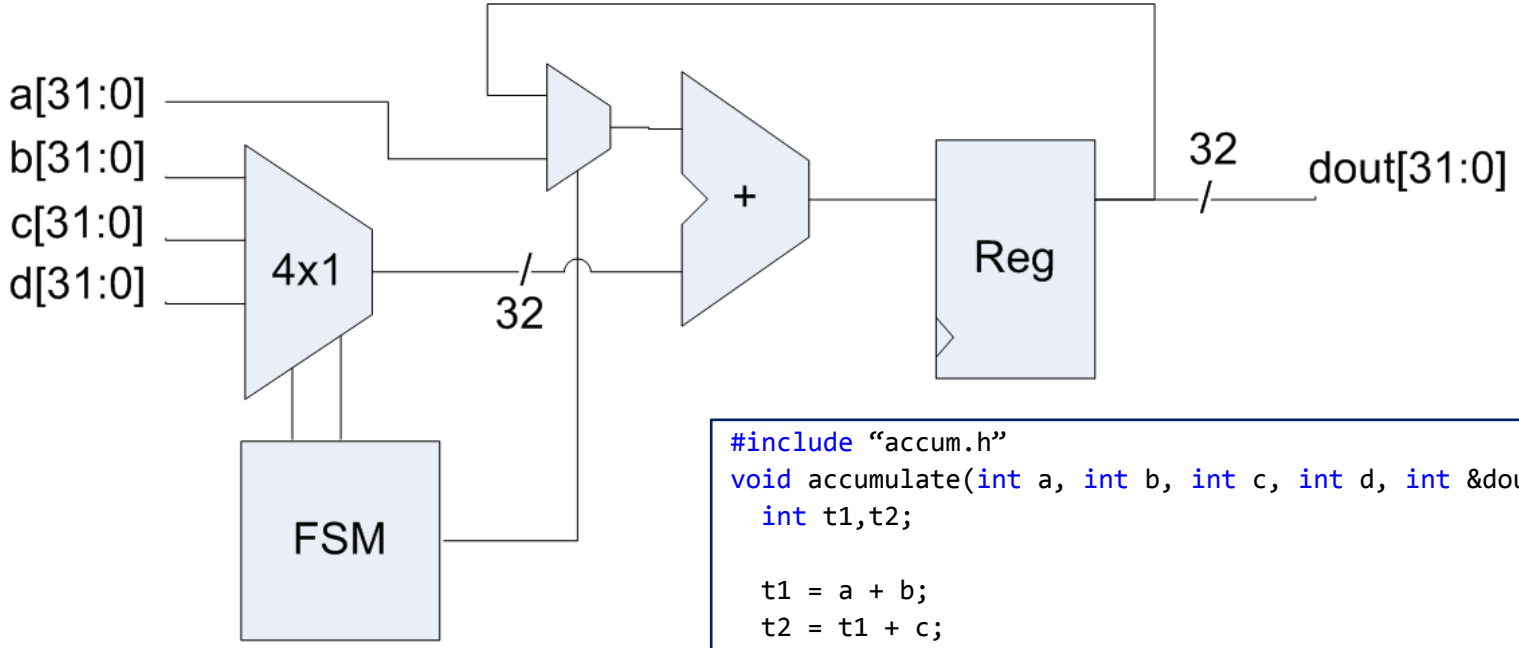
Ütemezés

- Egy „adatút állapotgép” (Data Path State Machine, DPFSM) készül az ütemezett terv vezérlésére
- Az állapotgép 4 csomópontot tartalmaz, melyek az egyes órajel ciklusokhoz tartoznak
- Ezek a „control step”-ek, vagy „c-step”-ek
- Az állapot diagrammból látszik, hogy minden 4. óraciklusban jelenik meg egy új kimenet (C4 után újra C1, stb.)



A szintetizált hardver

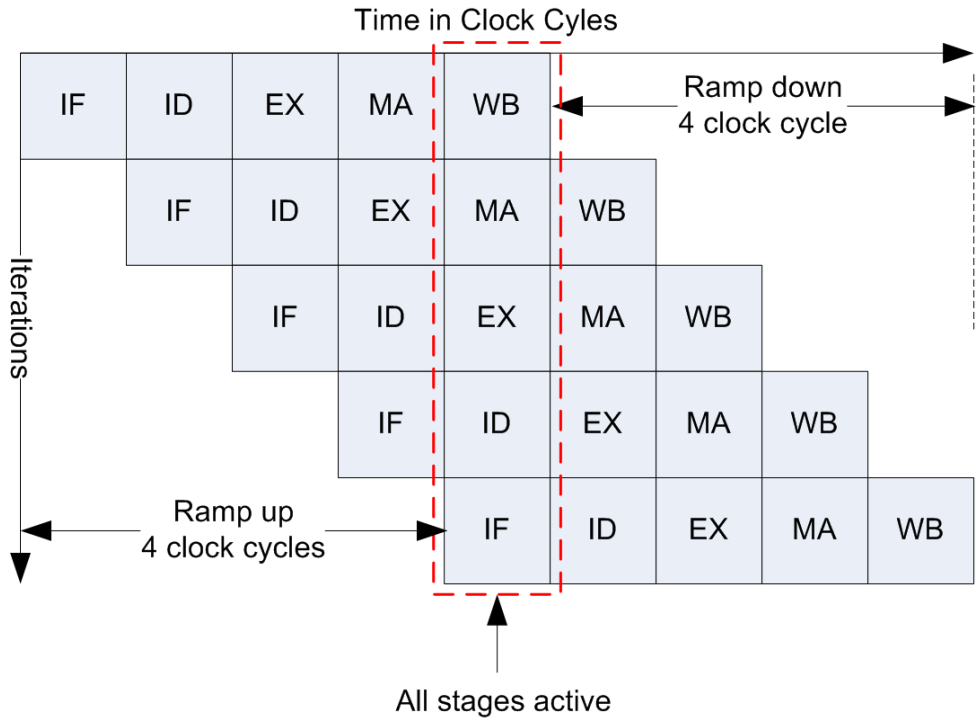
- A generált hardver a kényszerek és az ütemezés alapján készül el
- A kényszerek nélküli tervet a szintézer a minimális számú erőforrással valósítja meg (itt: minimális számú összeadó)
- Adatút: 32 bit az **`int`** miatt




```
#include “accum.h”
void accumulate(int a, int b, int c, int d, int &dout){
  int t1,t2;

  t1 = a + b;
  t2 = t1 + c;
  dout = t2 + d;
}
```

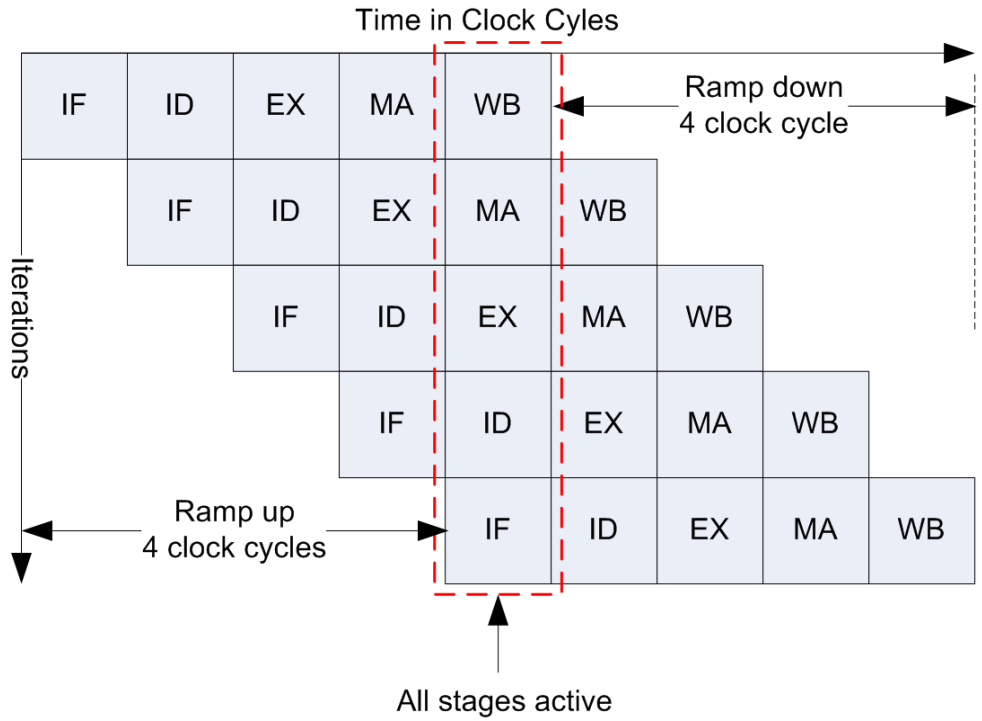
Klasszikus RISC csővezeték (pipeline)

- A HLS-ben a „ciklus csővezeték” (loop pipelining) hasonló, mint a klasszikus RISC csővezeték
- Az alap 5 fokozatú pipeline a RISC architektúrában
  - Instruction Fetch (IF)
  - Instruction Decode (ID)
  - Execute (EX)
  - Memory Access (MA)
  - Register write back (WB)



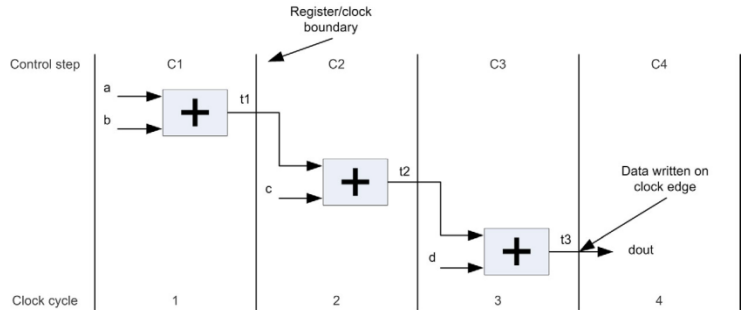
Klasszikus RISC csővezeték (pipeline)

- Az egyes fokozatok egymás után aktiválódnak
- „Ramp up”: ennyi idő kell, hogy minden fokozat bekapcsoljon
- „Ramp down”: ennyi idő kell, hogy minden fokozat inaktívvá váljon



Loop pipelining

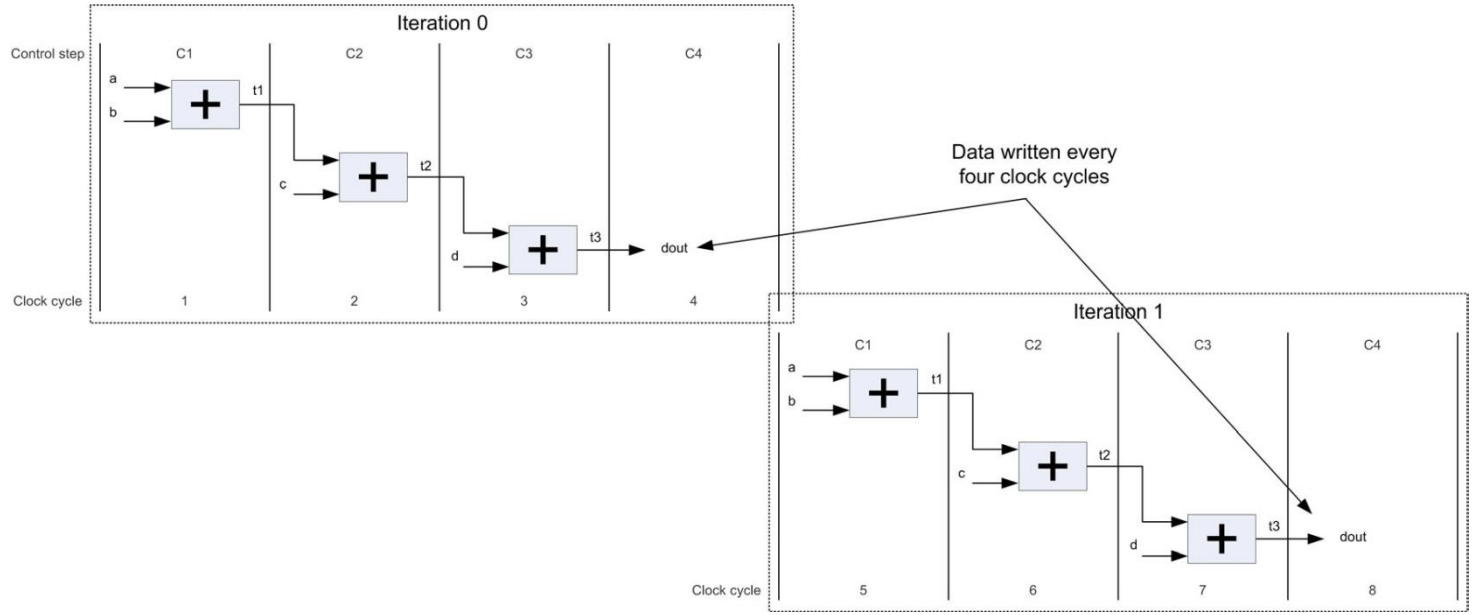
- Egy új ciklus-iteráció kezdődhet el miközben az aktuális még fut
- A példában ugyan nincs explicit ciklus, de az **`accumulate()`** függvény ciklikusan hívódik, ezt hívjuk „main loop”-nak
- A main loop minden iterációja a már ismert ütemezést hajtja végre
- A loop pipelining segítségével az iterációk átlapolhatók
- Gyorsítja a rendszert és párhuzamosít
- Az átlapolás nagyságát az **Initiation Interval** (II) adja meg




```
#include “accum.h”
void accumulate(int a, int b, int c, int d, int &dout){
  int t1,t2;

  t1 = a + b;
  t2 = t1 + c;
  dout = t2 + d;
}
```

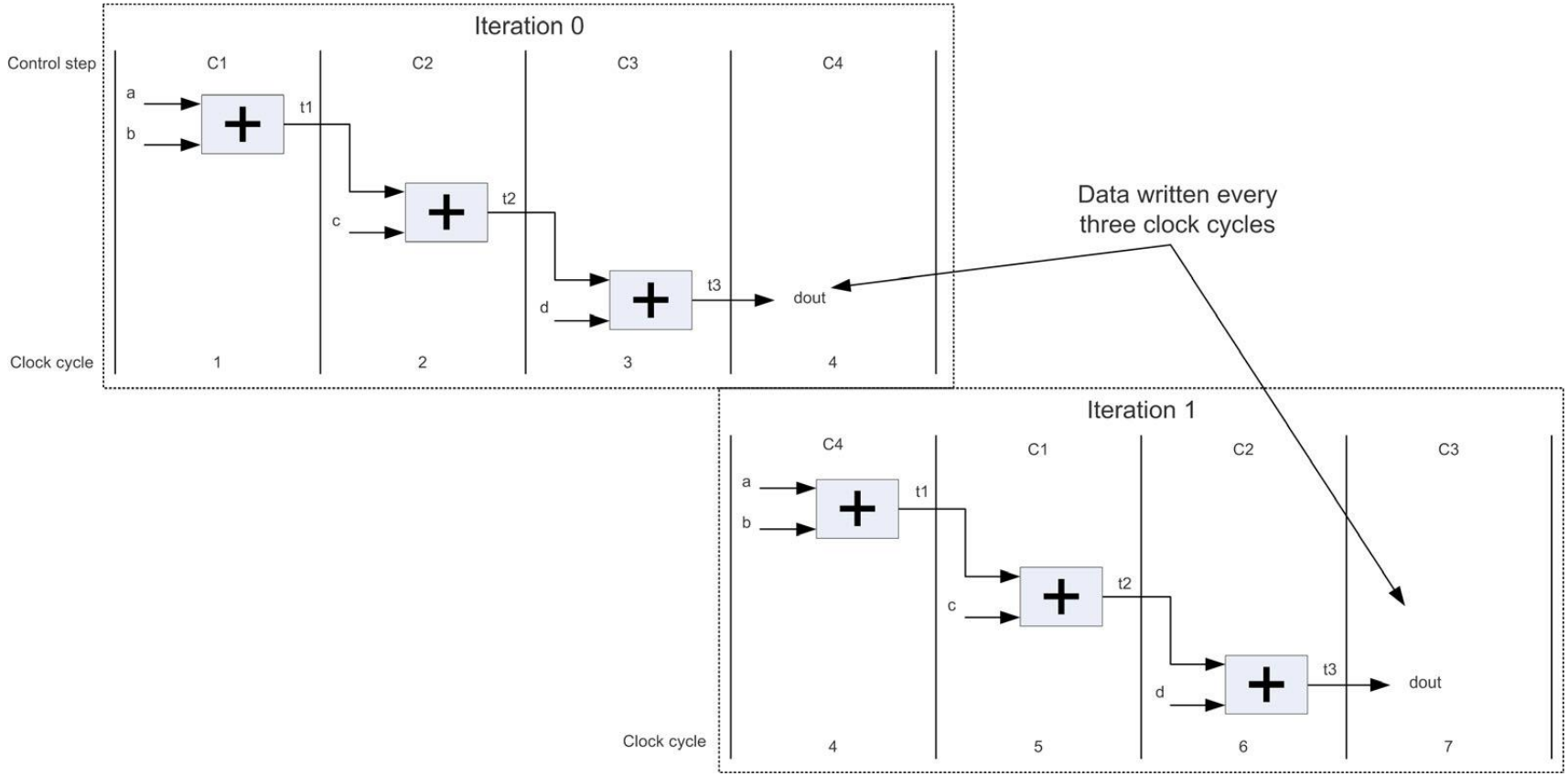
Indítási intervallum (Initiation Interval,II)

- II: mennyi óraciklus telik el az új iteráció megkezdéséig
- Pl. II = 1, minden óraciklusban kezdődjön egy iteráció
- Latency (lappangás): az idő, óraciklusban mérve, az első bemenettől az első kimenet megjelenéséig
- Throughput (áteresztőképesség): mennyi időnként lesz egy új kimenetünk (óraciklusban mérve)
- A kényszermentes példában: nincs pipeline, L=3, TP=4



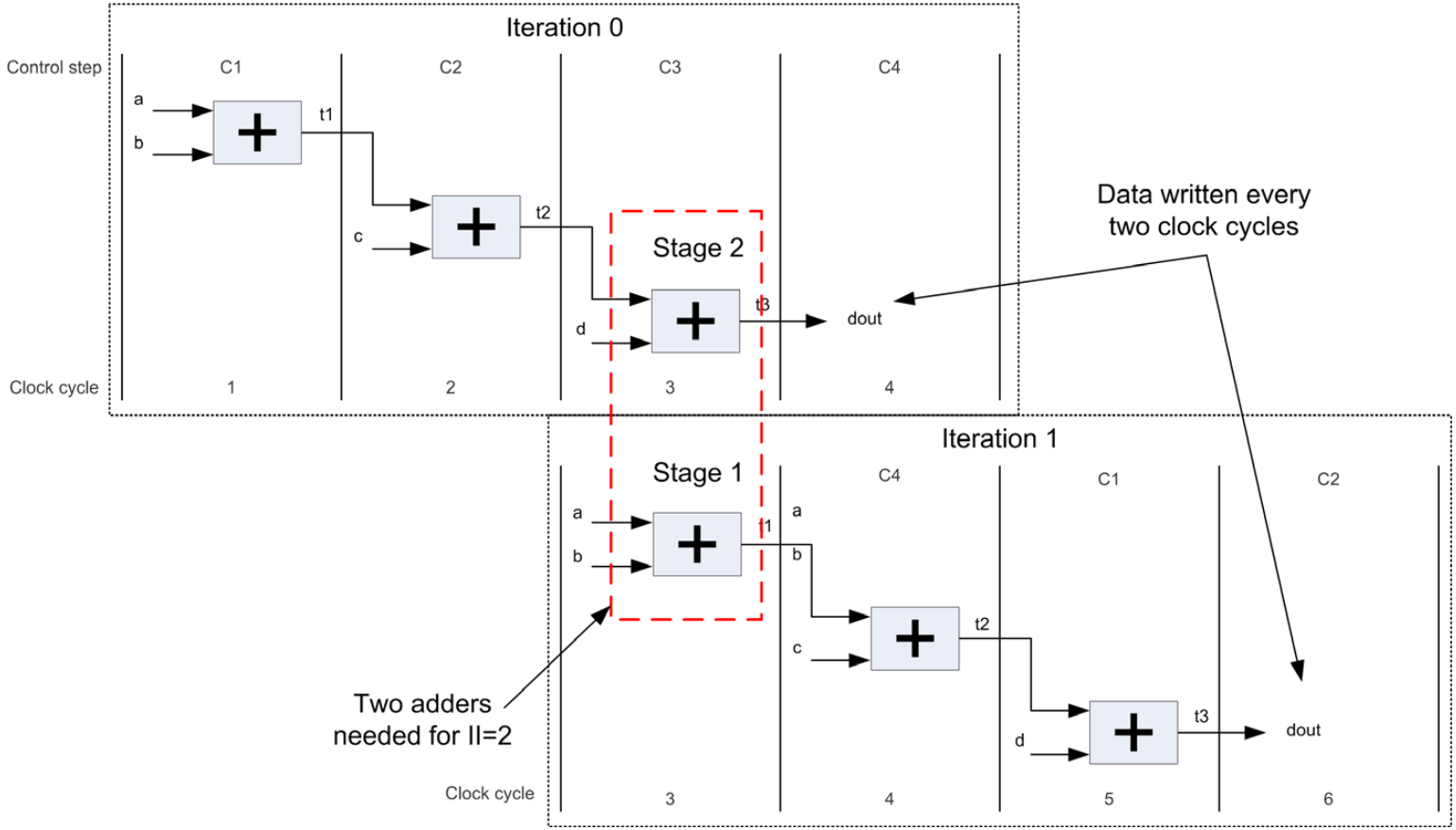
Initiation Interval = 3

- Ha a fő cikluson (main loop) II=3 kényszert állítunk be, az eredmény kiírása és az új adat beolvasása átlapolódhat
- Még mindig csak 1 összadó szükséges
- II=3, L=3, TP=3



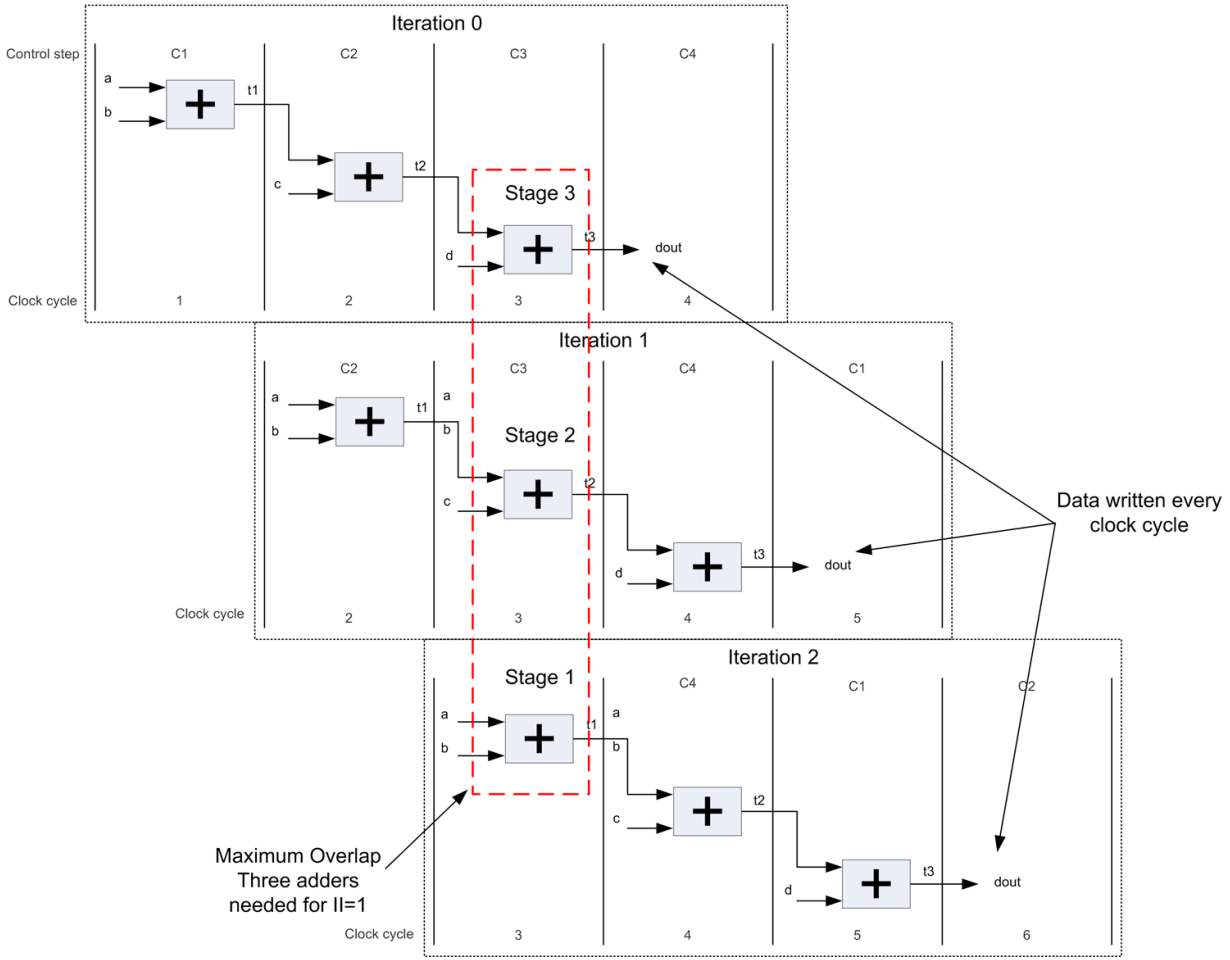
II = 2

- 2 órajelciklus átlapolódik
- Már 2 összadóra van szükség, de TP=2
- II=2, L=3, TP=2



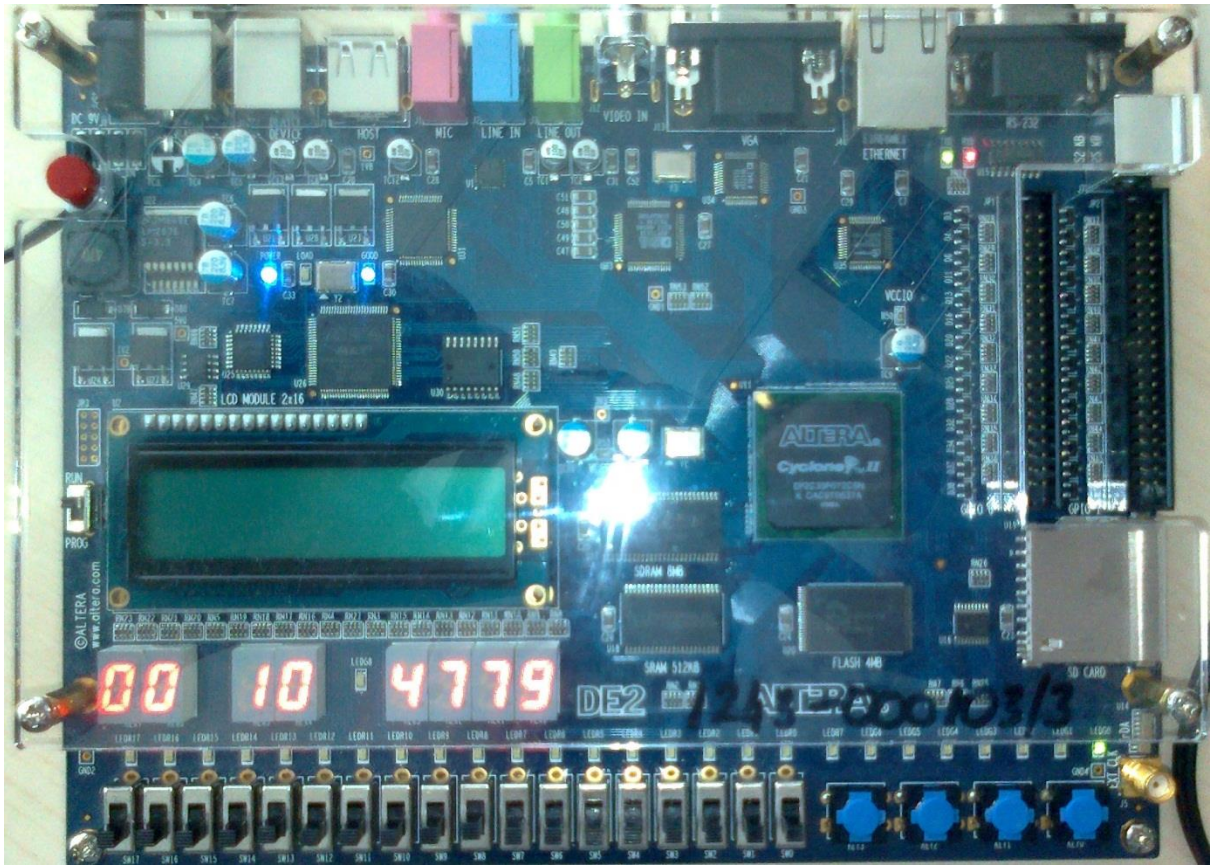
II = 1

- 3 órajelciklus átlapolódik
- Már 3 összadóra van szükség, de TP=1
- II=1, L=3, TP=1



Esettanulmány

- Egy egyszerű stopper megvalósítása Verilog/C++ szintézissel



Felhasznált eszközök

- Mentor Graphics CatapultC (C/C++ → RTL)
- Precision Synthesis RTL
- Altera Quartus II
- Altera DE2 Development board

Top modul

- Verilog modul
- Számláló

```
module counter8(clk, res, ledh, ledm, leds, HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7);
    input clk;
    input res;
    output reg ledh;
    output reg ledm;
    output reg leds;
    output  [6:0]   HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7;

    integer count;

    integer hours;
    integer mins;
    integer secs;
    integer msecs;

    wire [7:0] hourbcd;
    wire [7:0] minbcd;
    wire [7:0] secbcd;
    wire [7:0] msecbcd;

    always@(posedge clk)
    begin
    if (!res)
        begin
        ...
        end
    else
    begin
        ...
    end

    end

    hextobcd h0( .hex_rsc_z(msecs[7:0]), .bcd_rsc_z(msecbcd), .clk(clk), .rst(!res) );
    hextobcd h1( .hex_rsc_z(secs[7:0]), .bcd_rsc_z(secbcd), .clk(clk), .rst(!res) );
    hextobcd h2( .hex_rsc_z(mins[7:0]), .bcd_rsc_z(minbcd), .clk(clk), .rst(!res) );
    hextobcd h3( .hex_rsc_z(hours[7:0]), .bcd_rsc_z(hourbcd), .clk(clk), .rst(!res) );

    SEG7_LUT_8 seg7( HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7, {hourbcd,minbcd,secbcd,msecbcd} );
endmodule
```

Hextobcd modul, mely C-ben lett megírva
Szintézis után Verilog/VHDL RTL modul készül, ami itt példányosítható

Hex2BCD C modul

- Egy 8 bites hexadecimális számot BCD-re alakít

C modul

```
void hextobcd( const unsigned char hex, unsigned char *const bcd)
{
    unsigned char ten = hex / 10;
    unsigned char one = hex % 10;

    *bcd = ten << 4 | one;
}
```

Verilog RTL modul

```
module hextobcd ( hex_rsc_z, bcd_rsc_z, clk, rst ) ;

    input [7:0]hex_rsc_z ;
    output [7:0]bcd_rsc_z ;
    input clk ;
    input rst ;

    wire [7:0]bcd_rsc_z_dup_0;
    wire [1:0]hextobcd_core_inst_fsm_output;
    wire clk_int, rst_int;
    wire [7:0]hex_rsc_z_int;
    wire hextobcd_core_inst_acc_imod_sva_2_, hextobcd_core_inst_acc_imod_sva_1_,
         hextobcd_core_inst_conc_imod_sg1_sva_2_,
         hextobcd_core_inst_bcd_rsc_mgc_out_stdreg_d_0n1s9_2_,
         hextobcd_core_inst_bcd_rsc_mgc_out_stdreg_d_0n1s9_1_,
         hextobcd_core_inst_bcd_rsc_mgc_out_stdreg_d_0n1s6_3_,
         hextobcd_core_inst_bcd_rsc_mgc_out_stdreg_d_0n1s6_2_,
         hextobcd_core_inst_bcd_rsc_mgc_out_stdreg_d_0n1s15_2_,
         hextobcd_core_inst_bcd_rsc_mgc_out_stdreg_d_0n1s15_1_;
    wire [3:0]hextobcd_core_inst_bcd_rsc_mgc_out_stdreg_d_0n1s5;
    wire [2:0]hextobcd_core_inst_bcd_rsc_mgc_out_stdreg_d_0n1s2;
    wire [3:1]hextobcd_core_inst_acc_imod_sva_11n0s4;
    wire hextobcd_core_inst_rtlc11_copy_n89_0_, nx51874z3, nx51874z4, nx51874z1,
         nx51874z7, nx51874z2, hextobcd_core_inst_acc_imod_sva_0_, nx51874z5,
         nx51874z6, nx51874z10,
         hextobcd_core_inst_bcd_rsc_mgc_out_stdreg_d_0n1s15_0_, nx51874z8,
         nx51874z12, nx51874z11, nx51874z9, nx51874z14, nx51874z15, nx51874z13;
    wire [424:0] xmplr_dummy ;

    ...

    endmodule
```

Irodalomjegyzék

Michael Fingeroff, „*High-Level Synthesis – Blue Book*”, Mentor Graphics Corporation, 2010, ISBN: 978-1-4500-9724-6