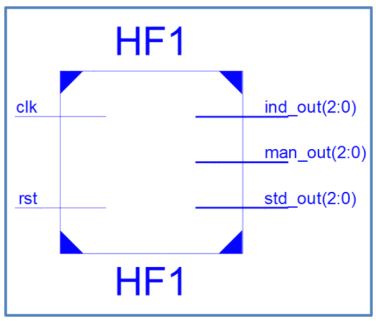
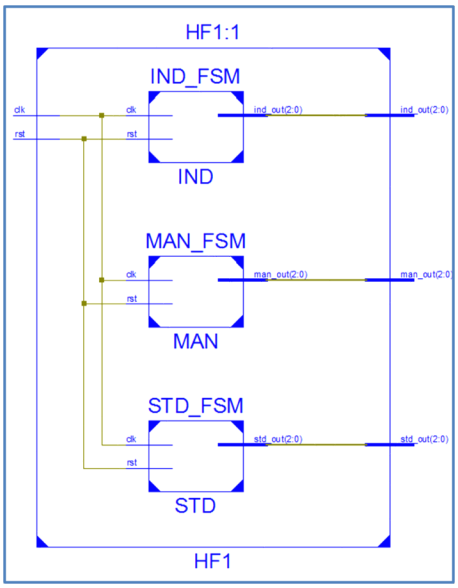
Véges állapotú gépek (FSM) tervezése 3.
(a tavalyi első házi feladat)

F1. A teljes terv 3 azonos funkciójú és interfészű almodulból épül fel. Az ezeket tartalmazó HF1.v felső szintű modul interfészspecifikációja a következő:




```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// HF1 mintamegoldás
// Használható referenciaként, a saját megoldás elkészítéséhez
//////////////////////////////////////////////////////////////////////////////////
module HF1(
    input clk,
    input rst,
    output [2:0] man_out,
    output [2:0] std_out,
    output [2:0] ind_out
    );
```

Mindegyik almodult a két közös vezérlőjel működteti, a clk és az rst. Az almodulok azonosítói a feladat kiírása szerint MAN_FSM, STD_FSM és IND_FSM. Az almodulok beépítésére a Verilog HDL szokásos példányosítási módszere alkalmazható.



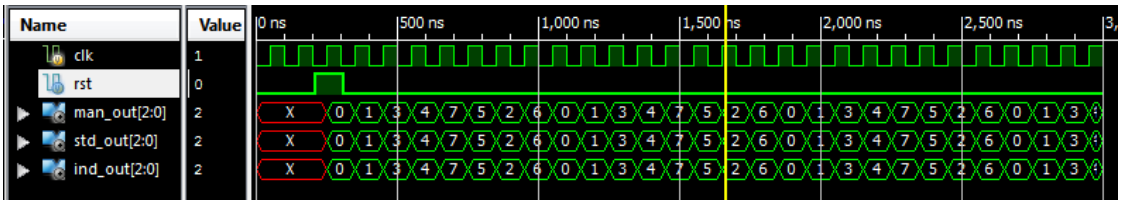
A teljes terv a források elkészítése után majd egy HF1_TF.v tesztkörnyezet segítségével lesz tesztelhető. A tesztkörnyezet automatikusan generálható, de kiegészítendő egy órajel generátorral a laboratóriumi mérések során ismertetett módon.

```
always                          // Folyamatos működésű órajelgenerátor
   begin
      #50 clk = ~clk;           // Periódusideje 2x50ns, azaz 100ns
   end
```

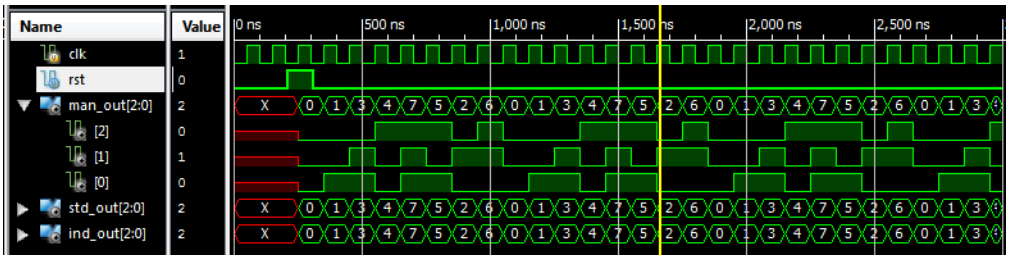
Ezen kívül a szükséges tesztvektorokat kell megadni. Minimum egy tesztvektor szükséges a modulok indításához.

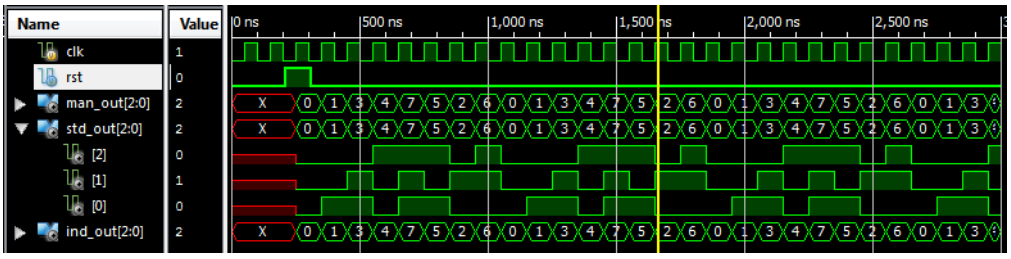
```
      // Add stimulus here
      #110 rst = 1;             // Kezdeti RESET pulzus az egységek inicializálásához
      #100 rst = 0;             // A RESET után indulnak a számlálási ciklusok
```

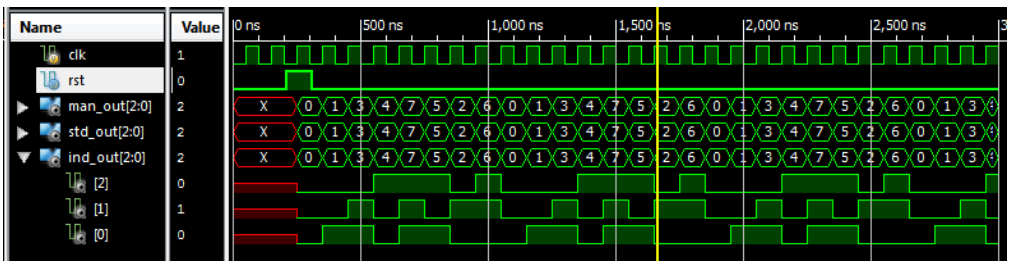
A szimulációt az egyes modulokon külön-külön vagy a teljes terv elkészítése után, a 3 egységet szinkron módon vizsgálva is elvégezhetjük. A segédlet alábbi ábrái ezzel a módszerrel készültek. A kimeneti bitvektorokat érdemes nem bináris, hanem valamilyen más kijelzési módba kapcsolni, mert úgy könnyebb az értékelés. Az ábrák a 1347526 minta DIGIT kód által meghatározott sorozatot mutatják.



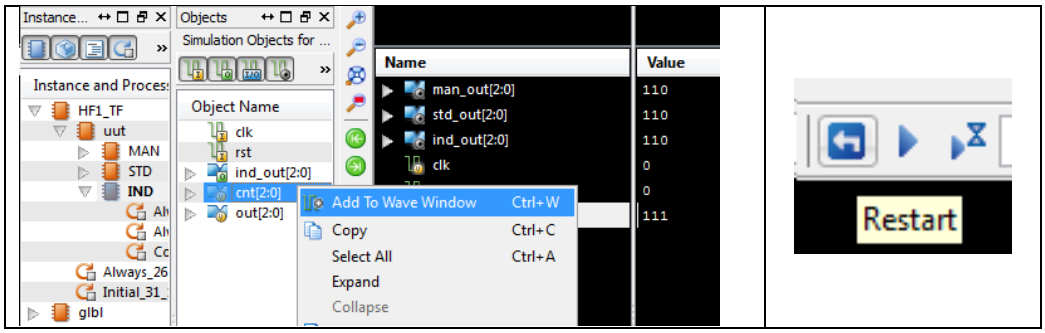
Természetesen hasznos lehet esetleg a bitvektorokat kibontva a részletes állapotváltozások vizsgálata, ellenőrzése is. Mindhárom bitképnek azonosnak kell lennie.



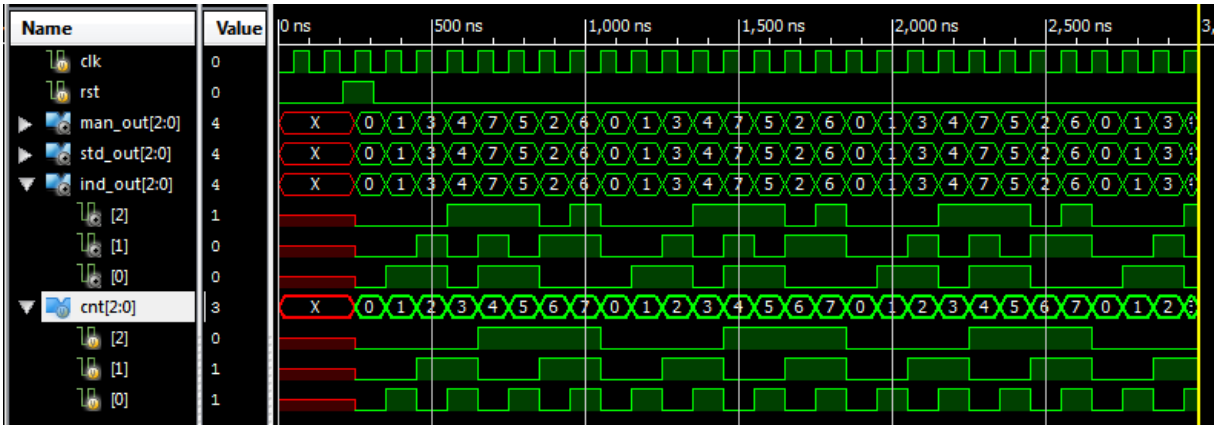




Az indirekt megvalósítású modul egy további lehetőséget kínál. A szimulált tesztkörnyezet hierarchiát kinyitva, a HF1 → uut → IND elérési úton keresztül hozzáférhetünk annak belső jeleihez, pl. a cnt[2:0] számláló jelhez. Ezt hozzáadva a hullámforma ablak jeleihez, majd a RESTART parancsot kiadva láthatóvá válik a számláló működése és az ez alapján generált kimenti jel.



AZ IND_FSM kimeneti és belső jeleinek kapcsolata:



Mindhárom modul interfészlistája azonos, a két bemeneti vezérlőjelet és a kimeneti 3 bites adatvektort tartalmazza.

```
////////////////////////////////////////////////////////////////////////////////
// Az XYZ_FSM modul interfész specifikációja
////////////////////////////////////////////////////////////////////////////////
module XYZ_FSM(
    input clk,
    input rst,
    output [2:0] xyz_out
    );
```

A MAN_FSM és az STD_FSM modulok működése egy 3 bites állapotregiszter órajel ciklusonkénti állapotváltozásán alapul, azaz a 3 bites **state** állapotváltozó felveszi a 3 bites **next_state** következő állapot változó értékét. Ezeket az állapot változásokat egy ***always*** blokkban adjuk meg, az órajel felfutó élével vezérelve és megadva a szinkron inicializálási feltételt is, amit az RST reset jel biztosít.

```
always @ (posedge clk)
   if (rst)     state <= 3'b0;
   else         state <= next_state;
```

A **state** és **next_state** változókat a használati feltételek szerint előzetesen deklarálni kell. A MAN_FSM esetén a **state** reg típusú, a **next_state** pedig wire típusú. Az STD_FSM esetén mindkettő reg típusú.

F1.a *Hagyományos (kézi) tervezési mód.* A MAN_FSM modul funkcióját az állapotátmeneti logika határozza meg. Ez egy 3 változós 3 kimenetű logikai függvény. A logika egyenletek felírásánál érdemes „rövidebb” egybetűs változókkal dolgoznunk.

```
////////////////////////////////////////////////////////////////////////////////
// Egy betűs átmeneti jelnevek bevezetése, a logikai kifejezések könnyebb
// kezelhetősége érdekében.
// A továbbiakban a == state[2], b == state[1], c == state[0].
////////////////////////////////////////////////////////////////////////////////

wire a,b,c ;
assign {a,b,c} = state;
```

Az állapotátmeneti logika igazság táblázata alapján a minimalizált egyenletek generálhatók algebrai átalakításokkal, Karnaugh tábla használatával (nem tananyag, de elfogadjuk a használatát), de akár más ismert általános célú eszköz segítségével is. A minta DIGIT kódnak választott 1347526 sorozat alapján az alábbi függvényeket kapjuk.

state[2:0] a b c		**next_state[2:0]**	
0	000	1	001
1	001	3	011
2	010	6	110
3	011	4	100
4	100	7	111
5	101	2	010
6	110	0	000
7	111	5	101

next_state[2] = /a·b·/c + **/a·b·c** + a·/b·/c + a·b·c = /a·b·/c + **/a·b·c** + a·/b·/c + a·b·c + **/a·b·c** =
= /a·b·**(c + /c)** + a·/b·/c + b·c·**(a + /a)** = ***/a·b + b·c + a·/b·/c***

next_state[1] = a·/b·/c + **a·/b·c** + /a·/b·c + /a·b·/c = a·/b·/c + **a·/b·c** + /a·/b·c + **a·/b·c** + /a·b·/c =
= a·/b·**(/c + c)** + /b·c·**(/a + a)** + /a·b·/c = ***a·/b + /b·c + /a·b·/c***

next_state[0] = **/a·/b·/c** + /a·/b·c + a·/b·/c + a·b·c = **/a·/b·/c** + /a·/b·c + **/a·/b·/c** + a·/b·/c + a·b·c =
= /a·/b·**(/c + c)** + /b·/c·**(/a + a)** + a·b·c = ***/a·/b + /b·/c + a·b·c***

next_state[2]

a \ bc	00	01	11	10
0	0	0	1	1
1	1	0	1	0

Piros: b·c
Kék: /a·b
Zöld: a·/b·/c

next_state[1]

a \ bc	00	01	11	10
0	0	1	0	1
1	1	1	0	0

Piros: /b·c
Kék: a·/b
Zöld: /a·b·/c

next_state[0]

a \ bc	00	01	11	10
0	1	1	0	0
1	1	0	1	0

Piros: /b·/c
Kék: /a·/b
Zöld: a·b·c

```
////////////////////////////////////////////////////////////////////////////////
// Az állapotátmeneti tábla feldolgozása után kapott legkedvezőbb
// logikai kifejezések a 3 bites következő állapot függvény kifejezésére
// Itt most nincsenek közösen használható szorzatok
////////////////////////////////////////////////////////////////////////////////

assign next_state[2] = ~a & b | b & c | a & ~b & ~c;
assign next_state[1] = a & ~b | ~b & c | ~a & b & ~c;
assign next_state[0] = ~a & ~b | ~b & ~c | a & b & c;
```

A DIGIT kód aktuális értékétől függően esetleg érdemes a 3 függvény együttes minimalizálását is megvizsgálni. A minta DIGIT kód esetén ez szerencsésen alakult és jelentős eredményt lehet elérni, 2 szorzatkifejezés is használható többszörösen a függvényekben. Ez a feladat már kézi módszerrel nem végezhető el egyszerűen, érdeklődő hallgatók megpróbálhatják a Logic Friday eszköz használatát.

```
////////////////////////////////////////////////////////////////////////////////
// Az állapotátmeneti tábla feldolgozása után kapott legkedvezőbb logikai
// kifejezések a 3 bites következő állapot függvény kifejezésére, ha kihasználjuk
// a közös szorzatok megosztásának lehetőségét.
// Itt a (~a & b & ~c) szorzatot 2x, a (a & ~b & ~c) szorzatot 3x tudtuk
// "újrahasznosítani", ami jelentős kapuszintű redukciót jelent.
////////////////////////////////////////////////////////////////////////////////

assign next_state[2] = ~a & b & ~c |  b & c | a & ~b & ~c;
assign next_state[1] = ~a & b & ~c | ~b & c | a & ~b & ~c;
assign next_state[0] =  a & b & c | ~a & ~b | a & ~b & ~c;
```

Ezután már csak az állapotérték átmásolása szükséges a kimeneti bitvektorba.

```
////////////////////////////////////////////////////////////////////////////////
// Az állapotregiszter értékének átmásolása a kimenetre
////////////////////////////////////////////////////////////////////////////////

assign man_out = state;
```

F1.b *Általános tervezési mód.* Az STD_FSM modul tervezésénél a modul funkcionalitására fókuszálunk. Az interfész specifikáció a szükséges bementi vezérlőjeleket és a 3 bites kimeneti std_out[2:0] vektort tartalmazza.

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Az STD_FSM modul
// Az általános FSM tervezési módszer használata
// Két always blokk
//    Az első a szinkron állapotregiszter megújítás
//    A második a következő állapotkódot előállító kombinációs logika
////////////////////////////////////////////////////////////////////////////////
module STD_FSM(
    input clk,
    input rst,
    output [2:0] std_out
    );
```

A Verilog HDL leírás a sorrendi hálózatok általános modelljét követi, az állapotregisztert és a következő állapot logikát viselkedési leírással, egy-egy ***always*** blokkon belül adjuk meg. Az első blokk egy szinkron, felfutó órajel él vezérlésű regiszteres működés.

```
////////////////////////////////////////////////////////////////////////////////
// Definiáljuk a 3 bites állapotregisztert és a next_state változót
// Az állapotregiszter egy RESET-elhető szinkron regiszter
////////////////////////////////////////////////////////////////////////////////

reg [2:0] state, next_state;

always @(posedge clk)
   if (rst) state <= START;
   else     state <= next_state;
```

A második közvetelenül megadja az állapot átmeneteket. Itt használhatunk közvetlen numerikus állapotkódokat, vagy szimbolikus állapotkódokat (Bár itt ebben az esetben semmivel nem adnak több információt a működésről, mint a numerikus értékadások.) Ha mégis használnánk, akkor a Verilog HDL parameter definicióval adhatjuk meg és javasoljuk a START, A, B, C, D, E, F G állapotneveket. A minta DIGIT kód esetén START=0, A=1, B=3, C=4, D=7, E=5, F=2 és G=6.

```
parameter <name> = <value>;
```

A viselkedési leírás egy kombinációs logikát specifikál.

```
////////////////////////////////////////////////////////////////////////////////
// Az állapotátmenetek előírása szimbolikus specifikációval
////////////////////////////////////////////////////////////////////////////////
always @(*)
   case (state)
      START  : next_state <= A;
      A      : next_state <= B;
      B      : next_state <= C;
      C      : next_state <= D;
      D      : next_state <= E;
      E      : next_state <= F;
      F      : next_state <= G;
      G      : next_state <= START;
      default: next_state <= START;
   endcase
```

Az állapotváltozó értékét ebben a modulban is ki kell vezetni a kimeneti bitvektorba.

```
////////////////////////////////////////////////////////////////////////////////
// A kimeneti érték azonos az állapotregiszter aktuális értékével
////////////////////////////////////////////////////////////////////////////////

assign std_out = state;
```

F1.c *Indirekt tervezési mód.* Az IND_FSM modul interfésze is hasonló az előző modulokhoz.

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Az IND_FSM modul
// Az állapotgép tervezése indirekt módon, egy belső számláló felhasználásával
// Két always blokk
//    Az első a számláló (itt most bináris számláló)
//    A második a CNT -> DIGIT kód átkódoló
////////////////////////////////////////////////////////////////////////////////
module IND_FSM(
    input clk,
    input rst,
    output [2:0] ind_out
    );
```

A 8 állapotból álló szekvenciát itt egy bináris számláló generálja. Ezt tekinthetjük egy olyan egyszerű állapotgépnek, ahol a következő állapot logika (ami a számláló regiszter értékét növeli) „bele van építve” a regiszter értékét megújító órajel vezérlésű blokkba.

```
////////////////////////////////////////////////////////////////////////////////
// Definiáljuk a 3 bites bináris számlálót
////////////////////////////////////////////////////////////////////////////////

reg [2:0] cnt, out;

always @(posedge clk)
   if (rst) cnt <= 3'd0;
   else     cnt <= cnt + 1;
```

A kimeneti logika itt egy 3 bemenetű, 3 kimenetű logikai hálózat. Mivel az első feladatban már végeztünk manuális függvényminimalizálást, itt most ezt a fejlesztőrendszerre hagyhatjuk. A teljes specifikációt egy kombinációs logikai hálózatot generáló viselkedési blokkban adhatjuk meg.

```
////////////////////////////////////////////////////////////////////////////////
// A kimeneti logika legegyszerűbb előírása
////////////////////////////////////////////////////////////////////////////////
always @(*)
   case (cnt)
      3'd0   : out <= 3'd0;
      3'd1   : out <= 3'd1;
      3'd2   : out <= 3'd3;
      3'd3   : out <= 3'd4;
      3'd4   : out <= 3'd7;
      3'd5   : out <= 3'd5;
      3'd6   : out <= 3'd2;
      3'd7   : out <= 3'd6;
      default: out <= 3'd0;
   endcase
```

A kész kimeneti értéket átmásolva a kimenetre, a HF1 feladat elkészült. Az ellenőrzést a korábban ismertetett szimulációkkal végezhetjük el.

```
////////////////////////////////////////////////////////////////////////////////
// A kimeneti érték átnevezése
////////////////////////////////////////////////////////////////////////////////
assign ind_out = out;
```