BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM

VILLAMOSMÉRNÖKI ÉS INFORMATIKAI KAR
MÉRNÖK INFORMATIKUS SZAK

Vizuális informatika szakirány

Önálló laboratórium 1. (BMEVIIIML00)

Automatikus hardverszintézis adatfolyamgráfból

Készítette:

Olasz-Szabó Bence (Q95A6S)

Konzulens:

Suba Gergely

IRÁNYÍTÁSTECHNIKA ÉS INFORMATIKA TANSZÉK

2015

TARTALOMJEGYZÉK

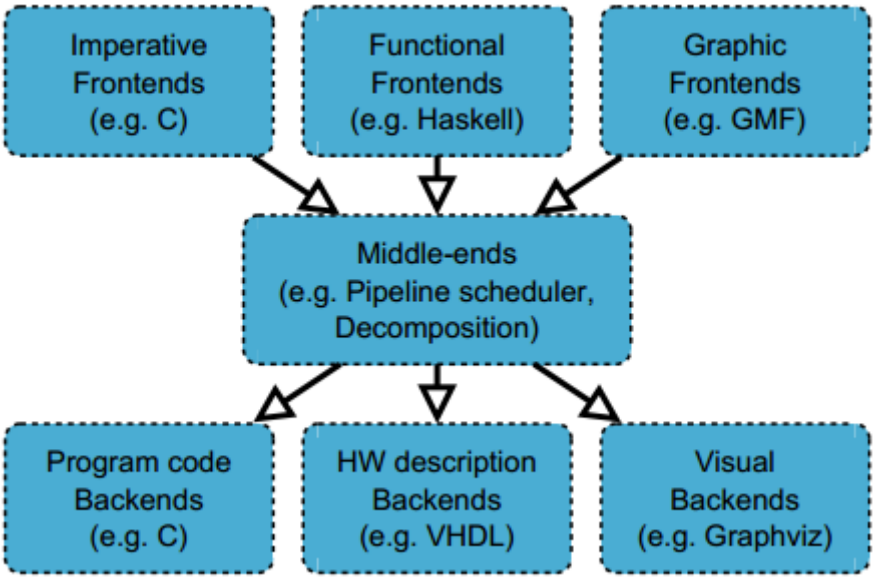
1. MAGAS SZINTŰ HARDVERSZINTÉZIS

Az önálló labor feladatom témája automatikus hardverszintézis adatfolyamgráfból, ami a magas szintű hardverszintézis (High Level Synthesis, HLS) témakörhöz kapcsolódik. A magas szintű hardverszintézis célja hardverleíró kód automatikus generálása magas szintű programnyelven leírt, tetszőleges algoritmusból kiindulva. Ez azért szükséges, mert a nagymértékben párhuzamosítható algoritmusok célhardvereken vagy FPGA-kon gyorsabban végrehajthatóak, mint a CPU-k alapvetően soros végrehajtási modellje szerint.

Azonban a hardverleírás készítésére jelenleg rendelkezésre álló nyelveken (VHDL, Verilog) nehézkes a fejlesztés. Egyrészt ezek a nyelvek nem adnak kellő rugalmasságot a fejlesztéshez, mivel alapvetően hardver egységek statikus szemléletű leírására készültek, nem algoritmusok leírására. Továbbá ezeken a nyelveken koncepció szinten párhuzamosan történik a műveletek végrehajtása, ami ugyan rákényszeríti a fejlesztőt a párhuzamosítható feladatok felismerésére, ugyanakkor jelentősen megnehezíti a nem párhuzamosítható feladatok leírását. Mindezek miatt szükség van a magas szintű programnyelveken leírt algoritmusok alapján hardverleírást generáló magas szintű hardverszintézis eszközökre.

1.1. PipeComp keretrendszer

Az Irányítástechnika és Informatika Tanszéken folyó HLS projekt egyik célkitűzése egy magas szintű szintézis keretrendszer implementálása. Ez a keretrendszer (PipeComp[1]) képes több kiindulási és célnyelv közötti kódgenerálásra is, amit a keretrendszer fejlesztői 3 rétegű architektúra segítségével valósítottak meg (1.1. ábra).



1.1. ábra. PipeComp keretrendszer architektúra

A felső réteg a Frontend, ami a kiindulási nyelven írt forráskódból egy köztes nyelven leírt adatfolyamgráfot (Hierarchy Intermediate Graph, HIG) generál. A középső réteg (Middle-end) az adatfolyamgráf transzformációját és optimalizációját hajtja végre. Az alsó réteg (Backend) az adatfolyamgráfból generál hardverleíró kódot.

1.2. HLS Backend

Az önálló labor feladatom során HIG adatfolyamgráfból Verilog kódot generáló (Backend) programot kellett terveznem és implementálnom a tanszéki HLS projekt részeként. A szoftver tervezésekor fontos szempont volt, hogy a program könnyen kiegészíthető legyen más hardverleíró nyelvű kód (elsősorban VHDL) generálását végző modulokkal is. A kódgenerátor tervezésének és elkészítésének lépéseit részletesen a 2. fejezetben ismertetem.

A kódgenerátor program elkészítése mellett a feladat része volt a generált Verilog fájlok szintaktikai ellenőrzése és tesztelése is szimulátor program segítségével. Mindemellett a generált Verilog fájlokból készült FPGA konfigurációs fájlokat SoCKit FPGA panelre távoli eléréssel feltöltő, valamint a feltöltött kóddal hálózaton keresztül történő kommunikációra képes rendszer (SoCKit tesztrendszer) fejlesztésével is foglalkoztam az önálló labor során, ezt a 3. fejezetben mutatom be.

2. HDL KÓD GENERÁLÁSA

A HIG adatfolyamgráfból hardverleíró kódot generáló szoftvernek képesnek kell lennie az alábbi feladatok elvégzésére:

- HIG adatfolyamgráf XML formátumú reprezentációjának beolvasása
- Adatmodell felépítése a memóriában a beolvasott adatok alapján
- Hardverleíró kód generálása az adatmodellből
- Alapműveletek generálása
- Komplex modulok generálása
- Ciklusok generálása
- Tömbök generálása

Fontos szempont továbbá, hogy a szoftver könnyen kiegészíthető legyen többféle nyelvű hardverleírás generálását végző modulokkal is.

2.1. Xtend keretrendszer

A kódgenerátor szoftvert alapvetően Java nyelven implementáltam. A Java API többféle lehetőséget is biztosít XML fájlok parse-olására, továbbá a Java nyelv interfészei segítségével könnyen kiegészíthető szoftver hozható létre. Mindezek miatt a feladat megoldására a Java nyelv ideális választás. Azonban a standard Java nem tartalmaz hosszú szövegek dinamikus generálását segítő elemeket, amire a kódgenerálás során szükség van. Ezért a szoftver kódgeneráláshoz szorosan kapcsolódó részeit az Xtend keretrendszerrel[2] készítettem el.

Az Xtend keretrendszer célkitűzése a Java nyelv modernizálása. Ezt azonban nem a JRE módosításával oldják meg, hanem fordítási idejű kódgenerálással: az xtend nyelvi kiegészítéseket használó forrásfájlokból fordítási időben generálódik Java 5 kompatibilis forráskód.

Az Xtend keretrendszer tartalmaz tipikusan kódgenerátor szoftverekhez használható nyelvi kiegészítéseket, amiket template-eknek[3] neveznek, ezeket használtam fel a Verilog kód generálásakor. Fontos megjegyezni, hogy ezek a template-ek nem azonosak a standard Java generikus template[4] nyelvi elemével. Az xtend template-ek olvasható string összefűzéshez biztosítanak nyelvi kiegészítéseket. A template kifejezések tripla idé-

zőjelek (’’’) közé kerülnek. A template kifejezésekbe a guillemets karakterek (« (Alt+174) és » (Alt+175)) közé írt kifejezések segítik a dinamikus kódgenerálást, elsősorban az IF és FOR nyelvi elemek segítségével, ahogy ez az alábbi kódrészletekben látható:

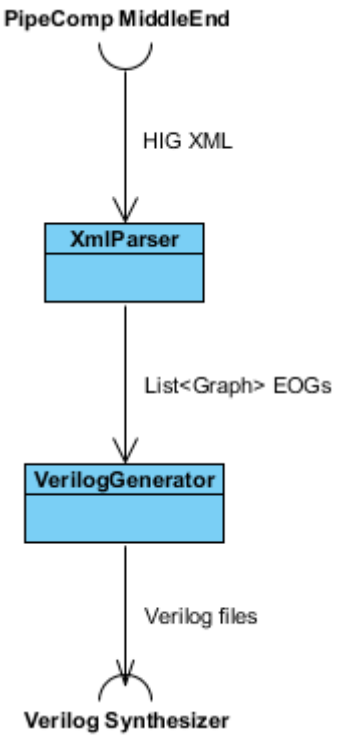
```
private def always(boolean synch)'''
      always @ («IF synch»posedge clk«ELSE»*«ENDIF»)
'''
```

```
override moduleInstantiation(String module, String instance, List<HDLPort>
ports)'''
«module» «instance»(
      «FOR p : ports SEPARATOR ',\n'».«p.innerName»(«p.outerName»)«ENDFOR»
);

'''
```

Megjegyzés: az xtend nyelven írt kód nem kompatibilis a standard java kóddal, csupán java kompatibilis kódra fordulnak le az xtend nyelven írt forrásfájlok. Azonban az xtend és java nyelven írt forrásfájlok korlátozás nélkül hivatkozhatnak egymásra, ezért ez nem okoz problémát.

2.2. Kódgenerátor architektúra

A kódgenerátor szoftver alapvető architektúráját a 2.1. ábra szemlélteti.



2.1. ábra. Kódgenerátor architektúra

Az XmlParser osztály az input HIG adatfolyamgráf XML formátumú reprezentációját beolvassa, felépíti belőle az adatmodellt a memóriában, majd az adatmodell referenciáját átadja a VerilogGenerator xtend osztálynak, ami elvégzi a kódgenerálást az adatmodell alapján. A folyamatot a main() függvény vezérli. A main() függvénynek paraméterül megadható a forrásfájl elérési útvonala (args[0]) és a generálandó fájlok kívánt elérési útja (args[1]).

```
/**
 * @param args[0] xml file (name without extension!) to parse
 * @param args[1] path to copy resource files to
 */
public static void main(String[] args) {
	System.out.println(args[0]);
	resourceDest = args[1];
	// Parse & generate
	(new VerilogGenerator()).generateSource((new
	XmlParser()).parseEOGs(args[0]+".xml"), args[0]);
	System.out.println("ready");
}
```

2.2.1. Input fájl beolvasása

Az input fájl beolvasását az XmlParser osztály végzi. A parser-nek az input fájlokból az alábbi elemeket kell értelmeznie:

- **eog:** a gráf elemeit összefogó tag
- **node:** a gráf egy csomópontját írja le. A csomópont típusát class és type attribútumai határozzák meg.
- **edge:** adott edge (él) 2 csomópont között definiál egy összeköttetést. Az élekhez portszámok is rendelhetőek. Ez azért szükséges, mert két csomópont több éllel is össze lehet kötve, valamint az egyes összeköttetéseknek különböző szerepük is lehet.

Adott eog-on belül előbb az összes node felsorolásra kerül, majd csak ezután következik az összeköttetések definíciója. Ezért a parser előbb beolvassa az eog-hoz tartozó node-okat és hozzáadja azokat az aktuális gráfhoz a modellben. Majd az élek beolvasásakor az aktuális élhez kikeresi a végpontjait. Az éleket reprezentáló Edge objektumok létrehozásukkor hozzáláncolják magukat a paraméterül kapott csomópontokhoz, így a gráf bejárható az összeköttetések mentén.

2.2.2. Adatmodell

A HIG alapján a memóriában felépített adatmodell a gráf elemeit reprezentáló objektumokból és az ezeket egységbe fogó Graph objektumokból áll. Adott Graph objektum a csomópontjainak referenciáit egy TreeMap-ben tárolja, amit a csomópontok id attribútumával indexel, így a csomópontok azonosító alapján gyorsan (O(logn)) elérhetőek, továbbá azonosító alapján rendezve vannak. Emellett a Graph a Port típusú csomópontjainak referenciáit egy listában is tárolja, mivel ezekre a kódgenerátornak külön szüksége van. A Graph az éleket nem tárolja, az egyes élek a csomópontok bejárásakor érhetőek el.

A gráf lehetséges elemeit osztályhierarchiába rendeztem, melynek ősosztálya az Element osztály. Az Element osztályból származnak az éleket és csomópontokat reprezentáló Edge és Node osztályok. Minden Edge tárolja a kezdő- és végpontja referenciáit és portszámait, valamint az él adattípus és adatszélesség információt is.

A Node-ok a bejövő éleik referenciáit egy TreeMap-ben tárolják, amit az élek célport (destination port, dport) attribútumaival indexelnek, ezáltal adott csomópontra adott porton bejövő él gyorsan megtalálható. A Node-ok a kimenő éleik referenciáit is tárolják, ezeket azonban csak egy egyszerű listában. A Node osztály a csomópont-típushierarchia ősosztálya, belőle származnak le a különböző funkciókat megvalósító csomópontok.

2.2.3. Output fájlok generálása

A kódgenerátor szoftver egyik fontos követelménye, hogy könnyen kiegészíthető legyen (a Verilog mellett) más hardverleíró nyelvekre történő kód generálását végző modulokkal. Ezért létrehoztam a HDLGenerator absztrakt osztályt és HDLConstructs java interfészt, amiket a VerilogGenerator és VerilogConstructs xtend osztályokban implementáltam. Ez utóbbiak mintájára, az előbbiek implementálásával könnyen készíthetőek más nyelvű forráskódot generáló osztályok, amik képesek együttműködni az adatmodellel a modell módosítása nélkül. Ez utóbbit az teszi lehetővé, hogy az adatmodell mindenhol a HDLConstructs interfészre hivatkozik, a VerilogConstructs osztályról nem tud. Ezáltal például a VHDLGenerator és VHDLConstructs osztályok implementálása és a VHDL nyelvű template fájlok létrehozása után a main() függvény apró módosításával generálható lenne VHDL nyelvű forráskód is az adatmodell és a parser módosítása nélkül.

```
(new VerilogGenerator()).generateSource((new XmlParser()).parseEOGs(args[0] +
".xml"), args[0]);
```

helyett:

```
List<Graph> EOGs = (new XmlParser()).parseEOGs(args[0] + ".xml");
(new VerilogGenerator()).generateSource(EOGs, args[0]);
(new VHDLGenerator()).generateSource(EOGs, args[0]);
```

2.2.4. A kódgenerátor és az adatmodell kapcsolata

A VerilogGenerator osztály generateFile() metódusa egy xtend template-et definiál egy verilog fájl generálásához. Egy HDL fájl általánosan a HDL modul interfészének definiálásából, a modul lokális változóinak deklarálásából és a modul tényleges funkcióját megvalósító állításokból áll. Ezért az adatmodell Node osztálya tartalmaz egy-egy metódust a fenti 3 funkció ellátására (portDef(), declaration() és statement()). A Node leszármazott osztályokban ezek a metódusok a konkrét Node funkciójának megfelelően kerülnek felüldefiniálásra. Az említett metódusok paraméterül egy HDLConstructs típusú paraméterként megkapják a HDL nyelvekben általánosan elérhető konstrukciók adott nyelvű implementációit. Azonban a modell elemei az adott nyelvi implementációt nem ismerik, csak az általános HDL nyelvi konstrukciókra hivatkozhatnak, így a szoftver a modell módosítása nélkül bővíthető más nyelvű hardverleíró kódot generáló részekkel. A VerilogGenerator osztály generateFile() metódusa az aktuális Graph Node-jainak fenti 3 metódusát hívja meg, ahogy ez az alábbi kódrészletben megfigyelhető:

```
protected override generateFile(Graph EOG, String moduleName)'''
	module «moduleName»(
		«FOR n : getPortDefNodes(EOG.getNodes) SEPARATOR ','»
			«n.portDef(hdlc)»
		«ENDFOR»
	);

	// Declarations
	«FOR n : EOG.getNodes»
		«n.declaration(hdlc)»
	«ENDFOR»

	// Statements
	«FOR n : EOG.getNodes»
		«n.statement(hdlc)»
	«ENDFOR»

	endmodule

'''
```

Megjegyzés: a hdlc paraméter statikus típusa HDLConstructs, dinamikus típusa VerilogConstructs.

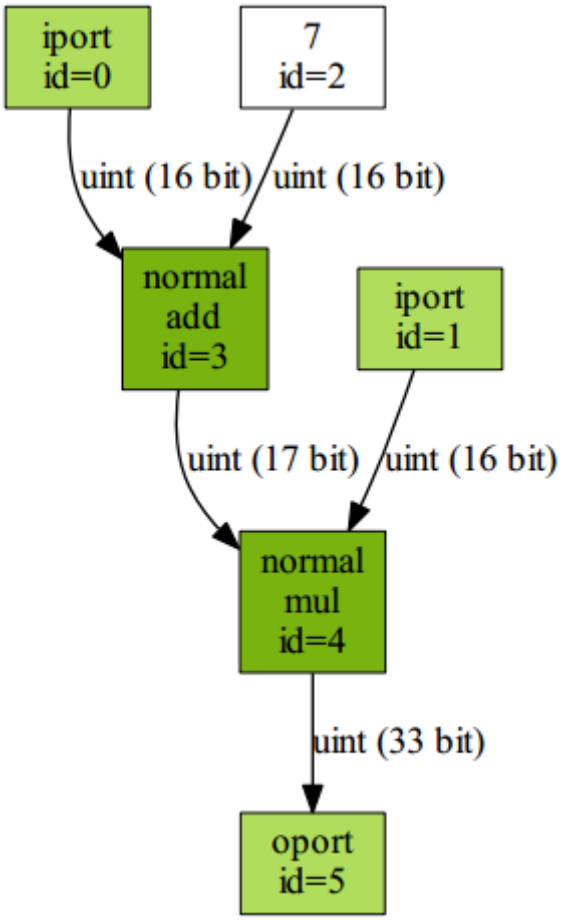
2.3. Kódgenerátor implementáció

A kódgenerátor architektúrájának megtervezése és alapvető működésének implementálása után az input HIG különböző típusú csomópontjainak megfelelő Verilog kód generálását végző funkciók megvalósítása következett. Ennek során először az alapvető kifejezések generálását oldottam meg, majd az ezekre építő, bonyolultabb nyelvi konstrukciók következtek.

2.3.1. Alapműveletek

Az alapvető kifejezések generálását a következő példa szemlélteti:

Adott az alábbi egyszerű adatfolyamgráf (2.2. ábra):



2.2. ábra. Egyszerű HIG

A 2.2. ábra látható HIG XML reprezentációja:

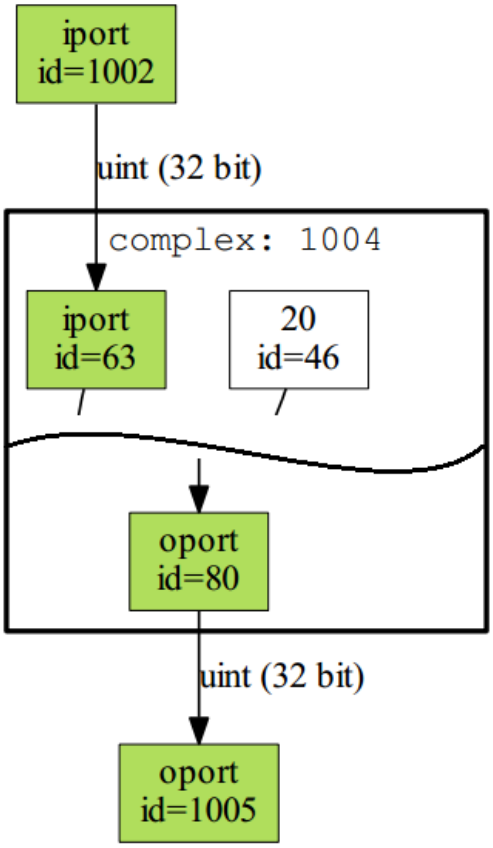
```
<pipe >
    <eog >
        <node id="0" class="iport" />
        <node id="1" class="iport" />
        <node id="2" class="const" value="7" />
        <node id="3" class="normal" type="add" />
        <node id="4" class="normal" type="mul" />
        <node id="5" class="oport" />
        <edge snode="0" dnode="3" dport="0" sport="0" type="link"
datatype="uint" datawidth="16" />
        <edge snode="2" dnode="3" dport="1" sport="0" type="link"
datatype="uint" datawidth="16" />
        <edge snode="3" dnode="4" dport="0" sport="0" type="link"
datatype="uint" datawidth="17" />
        <edge snode="1" dnode="4" dport="1" sport="0" type="link"
datatype="uint" datawidth="16" />
        <edge snode="4" dnode="5" dport="0" sport="0" type="link"
datatype="uint" datawidth="33" />
    </eog>
</pipe>
```

A fenti gráf ki- és bemeneti portokat, egyszerű aritmetikai műveleteket, egy konstanst, valamint az előbbieket összekötő éleket tartalmaz. A ki és bemeneti portok a generálandó verilog modul portjai lesznek. A többi csomóponthoz deklarálni kell 1-1 változót, ami a kimeneti értékét tartalmazza. A kimeneti érték adatszélességét a kimenő él datawidth attribútuma határozza meg. Ebben az egyszerű példában nincs szükség állapotokra, ezért kombinációs hálózat generálható, ezért a generált változók wire típusúak és az értékadásoknál használható az assign kulcsszó, nincs szükség még always blokkokra. A műveleteket a megfelelő verilog operátorok segítségével generálom. Ezek alapján a generált verilog kód:

```
module module0(
    input   [15:0] iport0,
    input   [15:0] iport1,
    output  [32:0] oport5,
    input   clk,
    input   reset
);

// Declarations
wire [15:0] p2 = 16'd7;
wire [16:0] add3res;
wire [32:0] mul4res;

// Statements
assign add3res = iport0 + p2;

assign mul4res = add3res * iport1;

assign oport5 = mul4res;


endmodule
```

2.3.2. Komplex modulok

Az alapműveletek után a komplex modulok generálását kellett megoldani. A komplex modul a HIG-ben egy olyan csomópont, ami egy újabb gráfot (EOG-ot) foglal magába a 2.3. ábra és az azt követő XML kódrészletben látható módon.



2.3. ábra. Komplex modul (egyszerűsített ábra)

```
<pipe >
    <eog >
        <node id="1004" class="complex" >
            <eog >
                <node id="46" class="const" type="" value="20" />
                <node id="63" class="iport" type="" />
                <node id="80" class="oport" type="" />
                ...
            </eog>
        </node >
        <node id="1002" class="iport" />
        <node id="1005" class="oport" />
        <edge snode="1002" dnode="1004" dport="63" sport="79"
datatype="uint" datawidth="32" />
        <edge snode="1004" dnode="1005" dport="65" sport="80"
datatype="uint" datawidth="32" />
    </eog>
</pipe>
```

Az egyes komplex modulokból generált kódoknak külön fájlba kell kerülniük, a komplex modult tartalmazó EOG-nak pedig gondoskodnia kell a beágyazott modul példányosításáról. A fő nehézséget itt a beágyazott és a tartalmazó modulok megfelelő jeleinek összekötése jelentette. A tartalmazó modul ugyanis magával a beágyazott modullal van összekötve, nem pedig a beágyazott modul megfelelő jeleivel, utóbbiakat csak portszám információként kapja meg. Ezért az összeköttetések generálását komlex modulok esetén ki kellett egészíteni. A modul példányosítás eredménye az alábbi kódrészletben figyelhető meg:

```
module module0(
    input   [31:0] iport1002,
    output  [31:0] oport1005,
    input   clk,
    input   reset
);

// Declarations
wire [31:0] iport63;
wire [31:0] oport80;

// Statements
assign iport63 = iport1002;

module1 module1_inst1004(
    .iport63(iport63),
    .oport80(oport80),
    .clk(clk),
    .reset(reset)
);

assign oport1005 = oport80;


endmodule
```

2.3.3. Ciklusok

A ciklusok generálásánál a fő megoldandó probléma abból adódott, hogy verilogban a műveletek alapvetően párhuzamosan hajtódnak végre, ezért nehéz egymás után végrehajtani a ciklus iterációit. Van ugyan verilogban for ciklus, azonban ez fordítási idejű konstrukció: a szintézer a ciklus magját egyszerűen egymás után másolja annyiszor, ahány iteráció van, a ciklusmag műveletei pedig egyszerre fognak végrehajtódni, nem lesz valódi iteráció. Tehát ha a ciklusmag műveletei függnek az előző iterációtól, akkor ez a konstrukció nem használható. Ebben az esetben a sorrendiséget ciklikus számlálók segítségével lehet biztosítani.

A HIG adatfolyamgráfban a ciklusokat reprezentáló, ún. loop csomópontok tulajdonképpen az előző fejezetben ismertetett komplex csomópontoknak felelnek meg. A loop-oknak azonban van 2 további attribútuma: a ciklus iterációinak számát megadó tripcount és az egy ciklus időtartamát órajelekben megadó restart attribútumok. Továbbá a loop-ok tartalmaznak egy speciális elemet, a loop_phi csomópontot, ami a ciklus iterációit vezérli az alábbi módon: ha az iterációs számláló 0, akkor az első bemenetéről fogadja a bemenetet (tehát cikluson kívülről), ha bármi más, akkor pedig a második bemenetéről (tehát cikluson belülről). Ezek alapján már elkészíthető a ciklusszámlálók verilog megvalósítása (a példában tripcount=10 és restart=4):

```
// Step loop counters
// Restart counter max + 1: clock cycles per iteration
// Trip counter max + 1: number of iterations per for cycle calls
always @ (posedge clk)
// On start
if(reset)
begin
    restart_counter <= 0;
    trip_counter <= 0;
end
// On iterations
else if(restart_counter==3)
begin
    restart_counter <= 0;
    if(trip_counter==9)
        trip_counter <= 0;
    else
        trip_counter <= trip_counter + 1;
end
// On clk
else
    restart_counter <= restart_counter + 1;

// LoopPhi
always @ (posedge clk)
// Phi loop is enabled on iterations
if(restart_counter==0)
    // On for cycle calls (cycle starts)
    if(trip_counter==0)
        loop_phiIDres <= outerInput;
    // On iterations
    else
        loop_phiIDres <= innerInput;
```

2.3.4. Tömbök

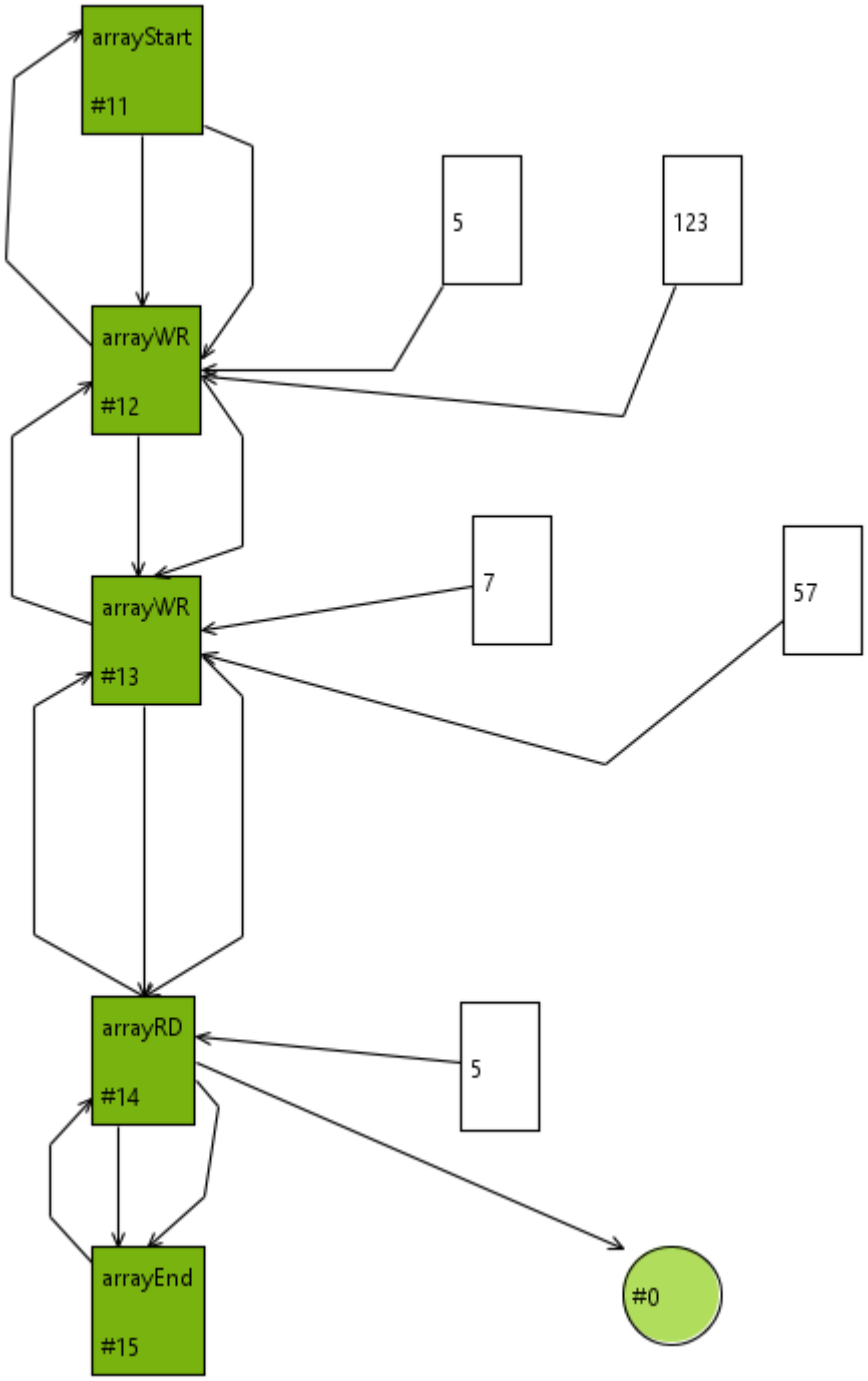
A tömböket FPGA platformon a beépített BlockRAM felhasználásával célszerű megvalósítani. A megvalósítással szemben további követelmény a gyártófüggetlenség: a generált verilog kódok tömbjeit a különböző gyártók szintézereinek egyaránt az FPGA beépített

BlockRAM-ának felhasználásával kell megvalósítania. Egy ilyen verilog kódot mutat be az „Inferring true dual-port, dual-clock RAMs in Xilinx and Altera FPGAs” című cikk[5]. A cikkben ismertetett verilog kód egyszerűsítésével elkészítettem a kódgeneráláshoz használt BlockRAM modul verilog kódját:

```
module bram_sp #(
    parameter DATA = 32,
    parameter ADDR = 8
) (
    // Port
    input   wire                clk,
    input   wire                wr,
    input   wire    [ADDR-1:0]  addr,
    input   wire    [DATA-1:0]  din,
    output  reg     [DATA-1:0]  dout
);

// Memory
reg [DATA-1:0] mem [(2**ADDR)-1:0];

// Access
always @(posedge clk) begin
    dout      <= mem[addr];
    if(wr) begin
        dout      <= din;
        mem[addr] <= din;
    end
end

endmodule
```

A fenti kódot az Altera szintézer valóban az FPGA beépített BlockRAM-jának felhasználásával szintetizálta. Mivel a fenti kód nem tartalmaz a kódgenerálás során kitöltendő paramétereket, ezért forráskódja változtatás nélkül másolható. Ezért a fenti kódot a kódgenerátor projektjében erőforrás fájlként kezelem: amennyiben szükség van rá, a fájl a projektből a generálandó forrásfájloknak beállított mappába másolódik át.

A fenti BlockRAM implementáció azonban csak egy interfészen keresztül érhető el, a tömbök elemeire viszont általában több helyről is hivatkozunk a kódban. A tömb elemeinek konkurens elérését a 2.4. ábra látható módon valósítottam meg a konzulensem javaslata alapján.



2.4. ábra.Tömbök

Az arrayEnd modul tartalmazza a tényleges BlockRAM implementációt. Az arrayRD modul kiolvassa a tömbből a megadott indexű elemet, az arrayWR modul pedig beírja a megadott adatot a tömb megadott indexű elemére. A köztes arrayRD és arrayWR modulok kétféle üzemmódban képesek működni: aktív üzemmódban az adott modul kommunikál a

tömbbel, transzparens üzemmódban pedig átereszti a többi modul kérését, ilyenkor tulajdonképpen vezetékként működik, azaz a kéréseket aszinkron módon, azonnal átereszti, nem órajelre továbbítja. Ez azért szükséges, hogy a megoldás sok köztes modul esetén is gyors maradhasson.

Fontos biztosítani, hogy a köztes modulok közül egyszerre csak pontosan 1 lehet aktív üzemmódban, az összes többi köztes modulnak transzparensnek kell lennie, különben az arrayEnd-től messzebb lévő modulok kérései nem jutnának érvényre. Ezt a feltételt kaszkádosított engedélyező jelek segítségével oldom meg. A soros tömbelérést az arrayStart modul kezdeményezi egy kezdeti engedélyező jel hatására. A következő órajelkor az arrayStart utáni modul kapja meg az engedélyező tokent, ekkor ez a modul aktív, az összes többi transzparens. A következő órajelre az aktuális modul továbbpasszolja az engedélyező tokent a következő modulnak. Az arrayEnd előtti modul ready jele pedig felhasználható a folyamat befejeződésének jelzésére.

Kódgenerálás szempontjából a fő problémát a tömbök esetében is a különböző arrayElement-ek összeköttetéseinek generálása jelentette. Ezt úgy oldottam meg, hogy az összekötő vezetékeket mindig az összeköttetés kiindulópontja definiálja, az összeköttetés végpontja pedig lekérdezi az összekötő vezeték nevét az Edge forrás port (source port, sport) attribútuma alapján. Az egyes portok funkcióit az alábbi táblázatban foglaltam össze:

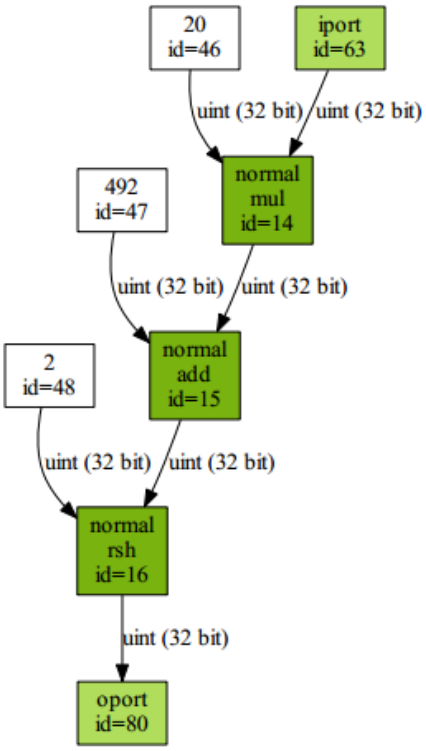
Port number	Name	I/O	datawidth	Direction	Note
0	addr_start	input	ADDR	from ArrayStart	Address channel
1	addr_end	output	ADDR	to ArrayEnd	Address channel
2	din_start	input	DATA	from ArrayStart	Data input channel
3	din_end	output	DATA	to ArrayEnd	Data input channel
4	dout_start	output	DATA	from ArrayStart	Data output channel
5	dout_end	input	DATA	to ArrayEnd	Data output channel
6	index	input	ADDR	from user code	Index of array element to which current operation corresponds
7	data	i/o	DATA	to/from user code	ArrayWR: input; ArrayRD: output
8	en	input	1	from user code	ArrayStart's enable starts the sequence of chained array operations
9	ready	output	1	to user code	Last operation's ready signs that all operations of the chain are completed
10	wr_start	input	1	from ArrayStart	1: write; 0: read
11	wr_end	output	1	to ArrayEnd	1: write; 0: read

2.4. Tesztelés

A generált verilog fájlok szintaktikai és szemantikai helyességét Altera ModelSim szimulációk segítségével ellenőriztem. A szimulációs eredményeket ebben a szakaszban ismertetem.

2.4.1. Alapműveletek tesztelése

Adott a 2.5. ábra látható bemeneti gráf:



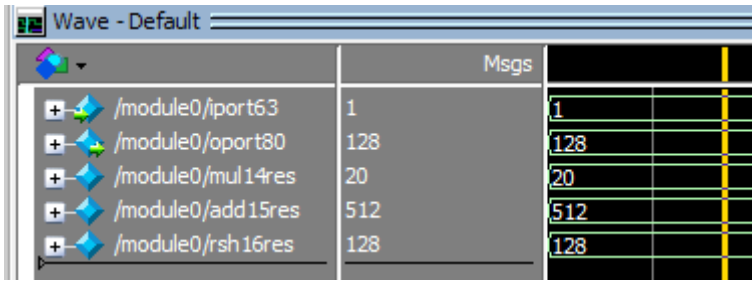
2.5. ábra. Alapműveletek tesztelése

Az ebből generált verilog kód:

```
module module0(
    input   [31:0] iport63,
    output  [31:0] oport80,
    input   clk,
    input   reset
);

// Declarations
wire [31:0] mul14res;
wire [31:0] add15res;
wire [31:0] rsh16res;
wire [31:0] p46 = 32'd20;
wire [31:0] p47 = 32'd492;
wire [31:0] p48 = 32'd2;
```

```
// Statements
assign mul14res = iport63 * p46;

assign add15res = mul14res + p47;

assign rsh16res = add15res >> p48;

assign oport80 = rsh16res;


endmodule
```

Szimulációs eredmény (2.6. ábra):

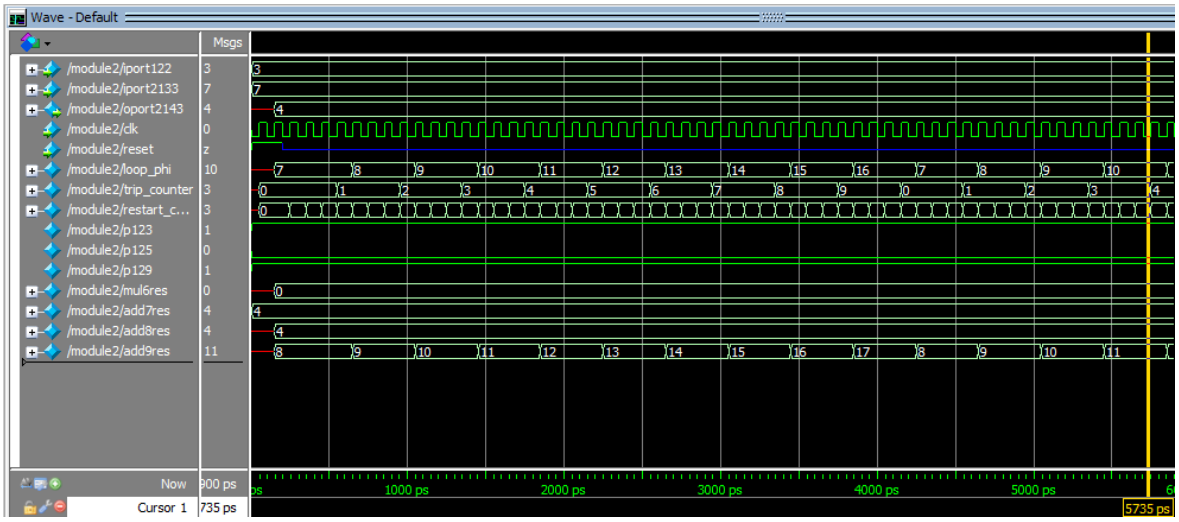


2.6. ábra. Alapműveletek tesztelése

Az adatfolyamgráf műveleteinek követésével belátható, hogy a szimuláció a várt eredményt adja.

2.4.2. Ciklusok tesztelése

A generált verilog kód ciklusszámlálóinak működése jól megfigyelhető a 2.7. ábra segítségével.

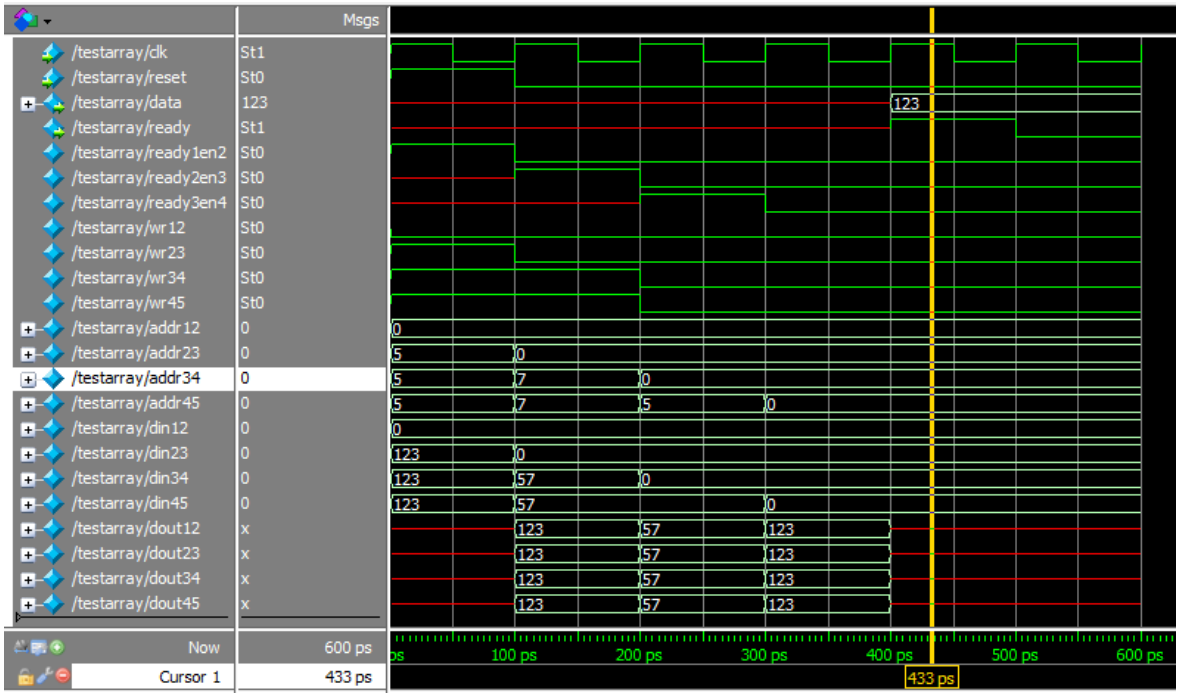


2.7. ábra. Ciklusok tesztelése

Látható, hogy a restart_counter 4 órajelenként lépteti a trip_counter-t, aminek hatására a ciklus egyesével számol felfelé 10 iteráción keresztül, majd elölről kezdi a számlálást.

2.4.3. Tömbök tesztelése

A 2.4. ábra látható adatfolyamgráfból generált verilog kód szimulációs eredménye az alábbi (2.8. ábra):



2.8. ábra. Tömbök tesztelése

Az ábrán megfigyelhető, hogy a tömbelérések a megadott sorrendben történtek, továbbá a tömbírások 1 órajelciklus alatt lezajlottak. A tömb olvasása 2 órajelciklust vesz igénybe: az első ciklusban az arrayRD modul átadja a kiolvasni kívánt tömbelem indexét a BlockRAM implementációt tartalmazó arrayEnd modulnak, a következő órajelciklusban pedig megkapja a kért tömbelem értékét. Látható továbbá, hogy az első ciklusban az 5. indexű helyre beírt 123 értéket az arrayRD művelet sikeresen kiolvasta a tömbből.

3. SOCKIT TESZTRENDSZER KÉSZÍTÉSE

A kódgenerátor szoftver elkészítése mellett az önálló labor során a generált Verilog fájlokból készült FPGA konfigurációs fájlokat SoCKit FPGA panelre távoli eléréssel feltöltő, valamint a feltöltött kóddal hálózaton keresztül történő kommunikációra képes rendszer (SoCKit tesztrendszer) fejlesztésével is foglalkoztam. A feladathoz az Altera SoCKit[6] fejlesztői paneljét használtam. Ez a panel egyaránt tartalmaz egy FPGA-t és egy 2 magos ARM CPU-t. A panel előnye, hogy beágyazott Linux telepíthető rá, továbbá az FPGA egységek elérhetőek a Linux alól is.

A feladatrész 3 fő lépésre bontható. Először telepíteni kell a panelra egy beágyazott Linux operációs rendszert. Ezután fel kell tudni tölteni a Linux alól a panelre az FPGA konfigurációs állományokat, valamint kommunikálni kell tudni a feltöltött kóddal. A 3. lépés pedig az, hogy az előbbieket távoli eléréssel, az interneten keresztül is meg lehessen tenni.

3.1. Beágyazott Linux telepítése

Az Altera SoCKit panelre elérhető egy előre lefordított beágyazott Linux konfiguráció, ezért először ezt[7] telepítettem fel a panelre. Azonban ezzel a konfigurációval problémák adódtak, ugyanis az FPGA-ra konfigurációs állományt csak az alábbi módszerrel sikerült feltölteni:

1) Set MSEL switch

MSEL[4:0]=00000 !!!!!

2) altera_gpio kernel modul átnevezése (altera-gpio2) (ki kellene lőni, de azt nem lehet)

3) **panel restart**

4) Disable enabled bridges

echo 0 > /sys/class/fpga-bridge/fpga2hps/enable

echo 0 > /sys/class/fpga-bridge/hps2fpga/enable

echo 0 > /sys/class/fpga-bridge/lwhps2fpga/enable

5) Program FPGA

dd if=sockit_test.rbf of=/dev/fpga0 bs=1M

vagy

dd if=soc_system.rbf of=/dev/fpga0 bs=1M (eredeti)

6) Enable needed bridges

echo 1 > /sys/class/fpga-bridge/fpga2hps/enable

echo 1 > /sys/class/fpga-bridge/hps2fpga/enable

echo 1 > /sys/class/fpga-bridge/lwhps2fpga/enable

7) gpio kernel modul betöltése

insmod /lib/modules/3.9.0/kernel/drivers/gpio/gpio-altera2.ko

A probléma itt abból adódik, hogy a módszer tartalmaz egy panel restart lépést is, ami hosszú ideig tart, ráadásul jelentősen megnehezíti távoli elérés megvalósítását. Ezért a fenti Linux konfiguráció helyett egy másik Linux-ot kellett telepíteni a panelre, amit azonban előbb le kellett fordítani és konfigurálni kellett egy másik Linux rendszer alatt az „Exploring the Arrow SoCKit Part II – Installing Linux”[8] tutorialban leírtak szerint.

3.2. FPGA konfiguráció feltöltése

A Linux alól az FPGA a HPS to FPGA bridge-ek segítségével érhető el, memóriába ágyazott I/O eléréssel[9]. FPGA konfiguráció feltöltése a „dd if=sockit_test.rbf of=/dev/fpga0 bs=1M” parancs segítségével tehető meg a HPS to FPGA bridge-ek letiltása után.

3.3. Távoli elérés

Az FPGA konfiguráció távoli eléréséhez szükség van a beágyazott Linuxon egy Http szerverre, ami kiszolgálja a távoli kéréseket. Emellett bonyolultabb protokollok is alkalmazhatóak (pl. SOAP), ez utóbbi azonban a beágyazott környezet limitált erőforrásai és a C nyelven implementált SOAP library-k fordítási nehézségei miatt megfontolandó. Az előbbiek miatt az FPGA távoli elérését Http kérések kiszolgálásával, egyszerű REST interfész definiálásával oldottam meg.

A feltöltött FPGA konfiguráció elérésének generikusnak kell lennie abban az értelemben, hogy bármelyik port-ot el kell tudni érni távolról. A memóriába ágyazott I/O modell itt jól jön, így ugyanis memóriacímekként érhetőek el az FPGA I/O portjai, amikre így távolról számokként lehet hivatkozni. Az alábbi script egyszerű távoli elérést biztosít az FPGA-hoz az előbbi módszerrel:

```
#!/bin/sh
```

```
echo "Content-type: text/html"
echo ""
data=`echo "$QUERY_STRING" | sed "s|q=||"`
$(/home/root/fpga_interface "$data")
echo "Data given: "
echo "$data"
```

4. MUNKANAPLÓ

Dátum	Munkaóra	Tevékenység
2/11/2015	4	Fejlesztőkörnyezet telepítése (Eclipse+Xtend), feladat értelmezése
2/12/2015	5	Quartus II telepítése. Xtend projekt létrehozása, adatstruktúra készítése
2/13/2015	7	XML parser, EOG tároló osztály (Graph) és VerilogGenerator elkészítése
2/14/2015	4	SoCKit dokumentációk olvasása, kódgenerálás absztrakció növelése
2/15/2015	1	Operation osztály átalakítása: operation mapping. Tesztelés
2/16/2015	2	Node input tömb -> SortedMap. Konstansok paraméter helyett értékkel generálva
2/18/2015	5	Complex node-ok kezelése, modul példányosítás
2/19/2015	2	GIT setup
2/20/2015	5	Proxy node-ok, összehasonlító műveletek, Complex->Complex összekötte-tés leképzése, konstansok bitszámhelyes megadása, Concatenation
2/22/2015	2	Phi node
2/24/2015	2	VerilogConstructs szinkron műveletek, Node sync paraméter bevezetése
2/26/2015	5	Egyszerű ciklusok generálása
2/27/2015	3	SoCKit panel beüzemelése, kipróbálása
2/28/2015	2	SoCKit: Linux telepítése, kipróbálása
3/1/2015	4	SoCKit: rendszer készítése Qsys segítségével
3/2/2015	2	SoCKit Linux hibakeresés
3/6/2015	1	Kódgenerátor: áttekinthetőbb kód generálása. SoCKit: dokumentációk olva-sása
3/10/2015	2	SoCKit: DS-5 fejlesztő környezet konfigurálása, saját program írása a SoCKit-re Linux platformra
3/11/2015	3	SoCKit: csoap és OpenSoap könyvtárak kipróbálása
3/14/2015	2	SoCKit: web service megvalósítási lehetőségek keresése
3/18/2015	2	SoCKit: gsoap dokumentációk olvasása
3/23/2015	2	SoCKit: cross-compile lehetőségek keresése
3/25/2015	4	SoCKit: Http szerver keresése, telepítése (lighttpd)
3/26/2015	3	SoCKit: FPGA LED-ek elérése távolról Http-n keresztül
4/1/2015	6	SoCKit: lwhps2fpga bridge elérése programkódból és Http-n keresztül: FPGA input változó értékének beállítása távolról
4/2/2015	4	SoCKit: hibakeresés
4/4/2015	2	SoCKit: Linux
4/5/2015	2	Verilog Block RAM modul készítése
4/6/2015	2	Block RAM modul generáló kód készítése
4/8/2015	2	Generált Verilog fájlok tesztelése Altera ModelSim szimulátorral
4/12/2015	5	Kódgenerátor: konstansok bitszámhelyes generálása. SoCKit: Linux cross compile, image készítése
4/28/2015	3	Elosztott tömbhozzáférés megvalósítása Verilog-ban
4/29/2015	1	Elosztott tömbhozzáférés tesztelése ModelSim-ben, hibák javítása
4/30/2015	8	Kódgenerátor: array file-ok generálása; SoCKit: image fájl készítése, hibake-resés
5/1/2015	2	Kódgenerátor: testarray.xml kiegészítése datawidth paraméterekkel, DATA

		és ADDR paraméterek generálása
5/2/2015	2	Kódgenerátor: ArrayElement összeköttetések
5/4/2015	2	Kódgenerátor: ArrayElement összeköttetések
5/9/2015	5	Kódgenerátor: tömbök generálása, szimulálása, testarray.xml kiegészítése a hiányzó összeköttetésekkel, dokumentálás
5/14/2015	8	Dokumentálás
5/15/2015	6	Dokumentálás

5. IRODALOMJEGYZÉK ÉS HIVATKOZÁSOK

[1] G. Suba and P. Arató, "Concept of the system-level synthesis framework PipeComp," WAIT 2015: Workshop on the Advances of Information Technology, Budapest, 2015.

[2] Xtend keretrendszer: https://eclipse.org/xtend/ - 2015.05.14.

[3] Xtend template-ek: https://eclipse.org/xtend/documentation/203_xtend_expressions.html#templates - 2015.05.14.

[4] Java generics: https://docs.oracle.com/javase/tutorial/extra/generics/intro.html - 2015.05.14.

[5] Inferring true dual-port, dual-clock RAMs in Xilinx and Altera FPGAs: http://danstrother.com/2010/09/11/inferring-rams-in-fpgas/ - 2015.05.15.

[6] Altera SoCKit Development Board: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=816 - 2015.05.15.

[7] Linux SoCKit GSRG image: https://s3.amazonaws.com/rocketboards/releases/2014.01/sockit-gsrd/bin/linux-sockit-gsrd-13.1-bin.tar.gz - 2015.05.15.

[8] Howard Mao: Exploring the Arrow SoCKit Part II – Installing Linux: http://zhehaomao.com/blog/fpga/2013/12/24/sockit-2.html - 2015.05.15.

[9] Howard Mao: Exploring the Arrow SoCKit Part III – Controlling FPGA from Software: http://zhehaomao.com/blog/fpga/2013/12/27/sockit-3.html - 2015.05.15.