ActiveCells 1.0

User Guide

Draft ver. 4

by
D. Shulga

2014

Contents

Acknowledgments

1 Introduction

1.1 Welcome to ActiveCells!

Dear software and hardware engineers, programmers, enthusiasts and newcomers! We are very pleased that you are interested in *ActiveCells* .

ActiveCells is a high-level language and programming environment that allows you to design high-performance embedded systems on FPGA.

ActiveCells uses the concept of a distributed system on chip composed of interconnected "*cells*"[1]. Cells are implemented on an FPGA as soft-processors or dedicated hardware components. An easy-to-use development environment based on a high-level language allows simple and fast development of your applications on FPGA.

ActiveCells is the result of collaborative work between academic and industrial partners. The main objective was the development of a computing model that on one hand is easy to teach and understand but on the other hand can be used to solve computationally demanding problems. We hope that our language and development tools will be helpful for implementing your remarkable ideas and will find broad use in your projects.

Our *ActiveCells* team wishes you an enjoyable trip through the *ActiveCells* User Guide that introduces to you wonderful world of high-level FPGA programming!

For any questions please contact us by email: *activecells@highdim.com*.

1.2 What is FPGA?

FPGA stands for Field Programmable Gate Array and represents a silicon integrated programmable electronic circuit. An instance is presented in Fig. 1. An FPGA contains a large array of programmable logic blocks which are connected via configurable interconnections. An FPGA can be programmed to a literally any desired computational hardware architecture.

Fig. 1: FPGA microchip on board

Modern FPGAs usually feature many prebuilt components ready for use, such as dedicated DSP-blocks, blocks of memory (BRAM), high-speed transceivers, memory interfaces and analog mixed signal circuits. This makes FPGA exceptionally efficient and flexible for different computational-demanding, real-time tasks and computing applications.

[1]This idea is reflected in *ActiveCells* logo that is showed on title page.

Because of their hardware configurability, FPGAs are much more flexible than off-the-shelf CPUs or Application Specific Integrated Circuits (ASICs): FPGA can be configured into complex system consisting of multiple CPUs and dedicated hardware blocks (accelerators). Implementations of an algorithm on an FPGA can heavily utilize parallelism, pipelining and plain data streaming eliminating CPU-inherited instructions fetching, as the instructions are built into the data path themselves.

The drawback of today's FPGAs is the relatively low achievable clock speed and the comparably high static power consumption. But because they can be adapted to the problem at hand, it is still very often possible to outperform standard architectures in terms of computing power per Watt. This makes them suitable for embedded systems that require complex or demanding computations.

2 Getting Started

2.1 How to install *ActiveCells*

1. *ActiveCells* requires Xilinx ISE to be installed in Windows. Xilinx ISE Design Suite can be downloaded form the Internet. Xilinx offers the free WebPACK version of ISE. *ActiveCells* uses ISE utilities in the background and there is no need to be experienced with ISE.

2. The paths to ISE `bin` and `lib` directories should be added to the Windows `Path` variable. In the case of ISE verson 14.7 installed in 64-bit Windows the paths are as follows: `c:\Xilinx\14.7\ISE_DS\ISE\bin\nt64;c:\Xilinx\14.7\ISE_DS\ISE\lib\nt64`

3. Download the *ActiveCells* installation file from web-page *http://highdim.com/activecells*. Execute the *ActiveCellsSetup.exe* to install *ActiveCells* into your operating system.

Now you are ready to run *ActiveCells* and create your first application!

2.2 Make a "Hello World!" Application

Following the tradition of starting with a "Hello World!" application we are going to design a simple *ActiveCells* application that consist of a soft-processor (CPU), an UART receiver (UartRx) and transmitter (UartTx), a general-purpose output ports (GPO) and a LED Digits Display (LEDDigits). The block diagram of this "Hello World" basic system is shown in Fig. 2. The application will send the "Hello World!" sentence to host-computer terminal, blink by LEDs and display "FPGA" word on the LED Display.

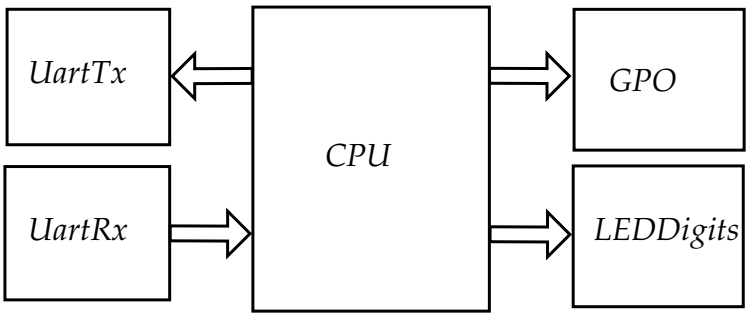


Fig. 2: "Hello World" system block diagram

To run this application we use a Spartan-3 Starter Board produced by Digilent. The board is shown in Fig. 3. This board contains a Xilinx XC3S200 Spartan-3 FPGA and peripheral hardware like SRAM, GPIO, LEDs, switches, LED Display, RS-232 serial port, VGA port.

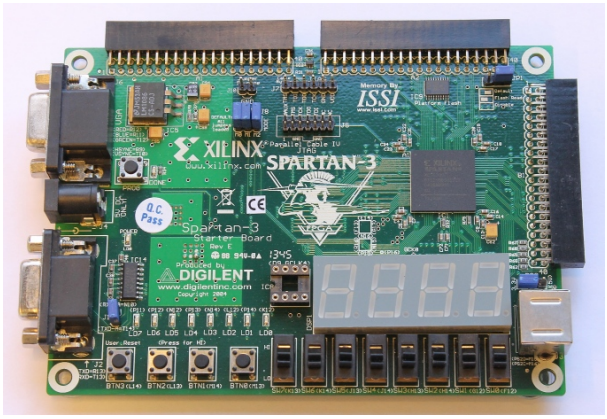


Fig. 3: Spartan-3 Starter Board

To create this *ActiveCells* "Hello World!" application following steps are needed:

1. Start *ActiveCells* by double clicking on the *ActiveCells* desktop icon or by clicking *ActiveCells* in Windows "Start" Menu. The *ActiveCells* desktop should appear as shown in Fig. 4. *ActiveCells* runs within the *A2* OS. The *A2* is a light-weight OS and it is very easy to start work with it.

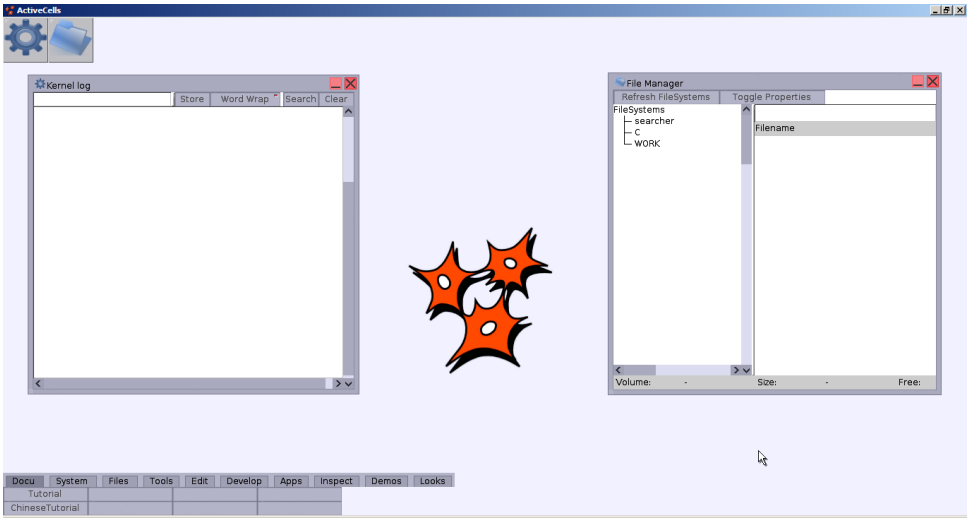


Fig. 4: *ActiveCells* desktop

2. In the menu at the bottom of the screen, select "Develop" tab and click on "IDE" button. The Programmer's Editing Tool (PET) will be opened (Fig. 5). We will use it to program our application.

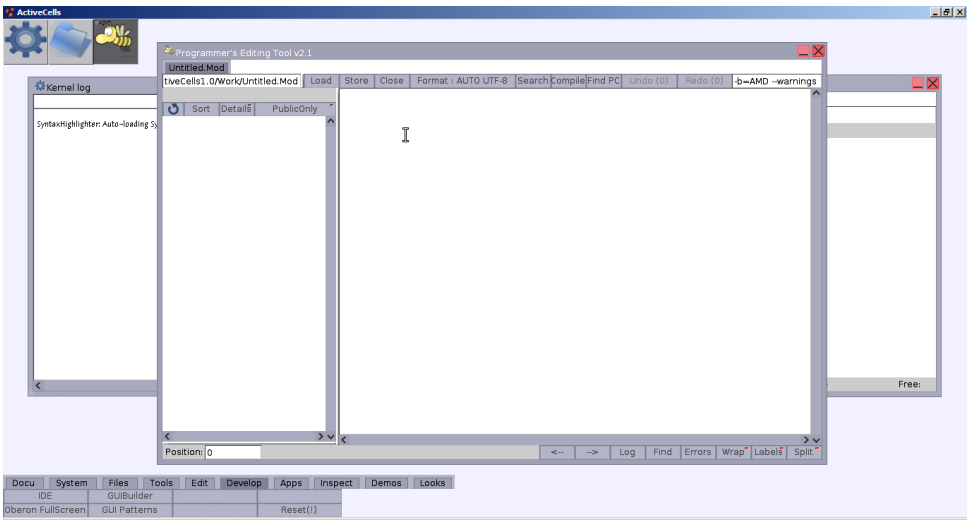


Fig. 5: Programmer's Editing Tool

3. You can see `Untiltled.Mod` file has already been opened in PET. Change this name to `HelloWorldSpartan3.Mdf` in the filename bar at the top-left of PET and press "Enter" key. Now we have empty file called `HelloWorldSpartan3.Mdf` that can be saved in the `ActiveCells/work` directory by pressing the "Store" button in PET.

4. Now we will start to write our *ActiveCells* Application. In file `HelloWorldSpartan3.Mdf` first we should define a **cellnet** called the same as file:

```
cellnet HelloWorldSpartan3;

end HelloWorldSpartan3.
```

   Cellnet is a composite entity that is consisted of other cellnets or **cells**. Cells are processing elements that on FPGA are implemented as soft-processors or specific hardware components.

5. Within the cellnet we now create hardware architecture of our "Hello World" system as is shown in Fig. 2. As our system consists of a CPU, a LED Display, a GPO, a UartTxa and UartRx: they are declared as cellnet variables in a **var** section:

```
cellnet HelloWorldSpartan3;

import
   Engines;

var
   controller: Controller;
   ledDigits: Engines.LEDDigits;
   gpo{DataWidth=8}: Engines.Gpo;
   uartTx{Baudrate=115200}: Engines.UartTx;
```

```
   uartRx{Baudrate=115200}: Engines.UartRx;
end HelloWorldSpartan3.
```

To use components we import **Engines** module (in section **import**). The "Controller" type plays the role of a CPU and will implement it later. LEDDigits, Gpo, UartTx, UartRx types are standard types from the *ActiveCells Hardware Library* and are declared in `Engines.Mdf` module. Some components have parameters: for UartRx and UartTx we specify Baudrate and for Gpo we specify DataWidth (in bits).

6. Now we implement the Controller type in a **type** section:

```
cellnet HelloWorldSpartan3;

import
   Engines;

type
   Controller = cell{Arch="TRM",CodeMemorySize=2048,DataMemorySize
   =2048}
            (
               uartOut: port out;
               uartInp: port in;
               gpoOut: port out;
               ledOut: port out
            );
   begin

   end Controller;

var
   controller: Controller;
   ledDigits: Engines.LEDDigits;
   gpo{DataWidth=8}: Engines.Gpo;
   uartTx{Baudrate=115200}: Engines.UartTx;
   uartRx{Baudrate=115200}: Engines.UartRx;

end HelloWorldSpartan3.
```

Controller is a cell type with a specific architecture (TRM - Tiny Register Machine originally proposed by Niklaus Wirth). We also specified Code and Data Memories sizes in words. Controller has input and output ports for communication with other hardware components.

7. In this step we instantiate our hardware components and connect them with each other. We do this in the body of the cellnet:

```
cellnet HelloWorldSpartan3;

import
   Engines;
```

```
type
   Controller = cell{Arch="TRM",CodeMemorySize=2048,DataMemorySize
   =2048}
            (
               uartOut: port out;
               uartInp: port in;
               gpoOut: port out;
               ledOut: port out
            );
   begin

   end Controller;

var
   controller: Controller;
   ledDigits: Engines.LEDDigits;
   gpo{DataWidth=8}: Engines.Gpo;
   uartTx{Baudrate=115200}: Engines.UartTx;
   uartRx{Baudrate=115200}: Engines.UartRx;

begin
   new(controller);
   new(uartTx);
   connect(controller.uartOut,uartTx.input);
   new(uartRx);
   connect(uartRx.output,controller.uartInp);
   new(gpo);
   connect(controller.gpoOut, gpo.input);
   new(ledDigits);
   connect(controller.ledOut, ledDigits.inp);
end HelloWorldSpartan3.
```

8. In the body of Controller ("**begin**...**end**") we implement the software of our application: we sends "Hello World!" to the UartTx, blinks the LEDs and display "FPGA" on the LED Display:

```
   Controller = cell{Arch="TRM",CodeMemorySize=2048,DataMemorySize
   =2048}
            (
               uartOut: port out;
               uartInp: port in;
               gpoOut: port out;
               ledOut: port out
            );
   var
      k,m: longint;
      str: array 16 of char;
   begin
      str := "Hello␣World!";
      for k:=0 to len(str)-1 do
         uartOut ! str[k];
```

```
        end;
        m := 8;
        loop
           ledOut ! {1,2,3,7,11};   (*F*)
           k := 0; while k<25000 do inc(k); end;
           ledOut ! {1,2,3,6,7,10}; (*P*)
           k := 0; while k<25000 do inc(k); end;
           ledOut ! {2,3,4,5,7,9}; (*G*)
           k := 0; while k<25000 do inc(k); end;
           ledOut ! {1,2,3,5,6,7,8}; (*A*)
           k := 0; while k<25000 do inc(k); end;
           gpoOut ! {m};
           dec(m);
           if m = -1 then
              m :=8;
           end;
        end;
     end Controller;
```

The operator "!" defines a *blocking write to the port* operation (similarly operator "?" defines a *blocking read from the port* operation). To perform bit-wise write operation we use **set** data type: within "{ ... }" we defines number of bits that we want to write. In next section we will give more detailed description of these operations.

9. We used a few additional modules in the controller body. They need to be imported:

```
import
   Engines;
```

10. To compile the code we have to execute compilation commands. In A2 OS commands can be executed from any place. It is convenient to place them below the cellnet:

```
...
end HelloWorldSpartan3.

ActiveCellsComponents.AddPath "ActiveCellsHWLib" ~

Compiler.Compile -b=TRM --objectFile=Intermediate --activeCells
HelloWorldSpartan3.Mdf
~

ActiveCellsComponents.BuildHardware --target="Spartan_XC3S200" --
   outputPath="HelloWorldSpartan3" HelloWorldSpartan3 ~
```

11. We can now connect JTAG programmer to the FPAG board, connect FPGA board and PC by RS232toUSB cable, power up the FPGA board.

We need to open a serial terminal to receive data from the board. In A2 menu go to the "Tools" tab and click "Terminal" button. In Terminal settings set the appropriate COM Port number and click "Open".

Note: the command `V24.Scan` shows the list of available serial ports. To execute it write it down after previous commands and double-click on it.

Now we need to compile "Hello World!" Application and build hardware:

(a) Execute the `ActiveCellsComponents.AddPath ...` command by double clicking on it. This command adds the *ActiveCells Hardware Library* components to the scope of the compiler.

(b) Execute the `Compiler.Compile ...` command to compile the application module `HelloWorldSpartan3.Mdf`

(c) Execute `ActiveCellsComponents.BuildHardware ...` to build hardware and program the FPGA board.

After performing the above steps, a bitstream will be created and the FPGA will be programmed to run our application. Then you should see the 'Hello Word!' sentence in the terminal, "FPGA" on the LED Display and fast LEDs blinking (Fig. 6 and Fig. 7).

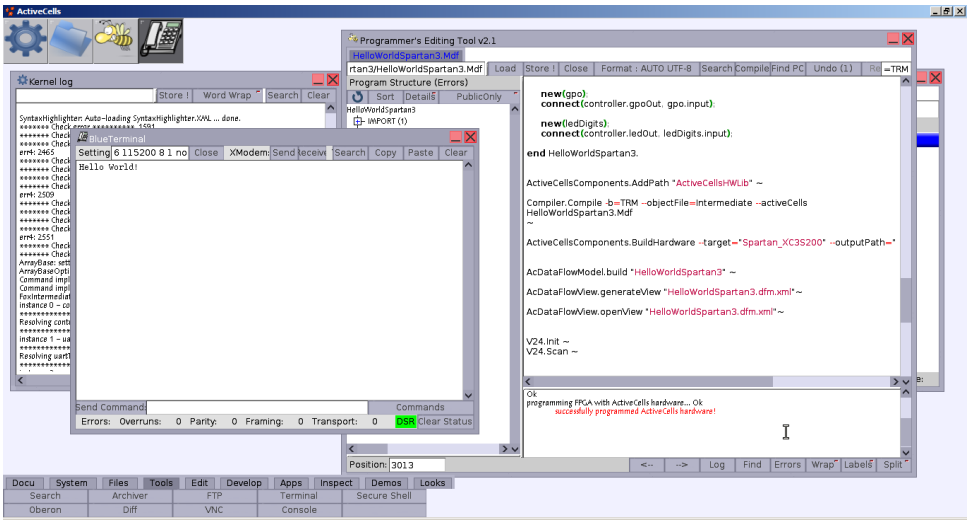


Fig. 6: "Hello World!" message in terminal window

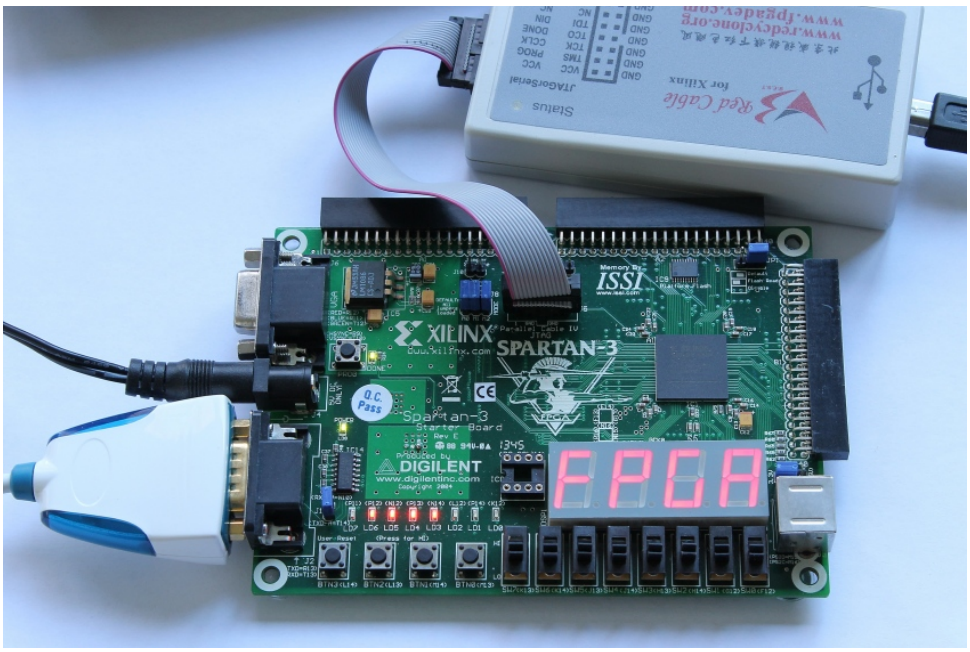


Fig. 7: Blinking LEDs and displayed "FPGA" word

12. Optionally we can build a visual diagram of our application. To do this we add the following commands at the end of file:

```
AcDataFlowModel.build "HelloWorldSpartan3" ~

AcDataFlowView.generateView "HelloWorldSpartan3.dfm.xml"~

AcDataFlowView.openView "HelloWorldSpartan3.dfm.xml"~
```

The first command builds the Data Flow Model, the second generates visual diagrams and opens Data Flow Viewer, the third could be used to open already generated and edited diagram.

The diagram of "Hello World!" Application is shown in Fig. 8.

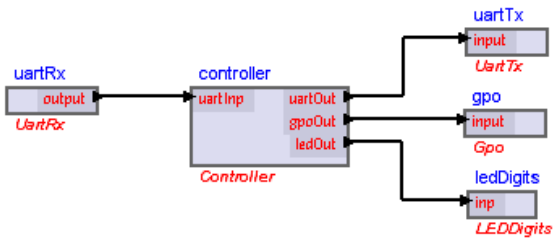


Fig. 8: Visual Diagram of "Hello World" system

3 *ActiveCells* Programming Model

3.1 Concepts

ActiveCells is a programming paradigm and development environment that applies a hybrid hardware/software co-design approach to build embedded systems on FPGA. The *ActiveCells* programming model unifies hardware and software programming under one high-level language and introduces a suitable abstraction level to simplify development of systems with high complexity.

Basic unit of *ActiveCells* programming language is a **cell**: a processing element that performs some specific activity. The diagram of a cell is shown in Fig. 9.

Cells have **ports** to communicate with each other via **channels**. Ports can be of type input or output. Generally a cell is an high-level abstraction of arbitrary activity and can be used without knowledge of how it is implemented.

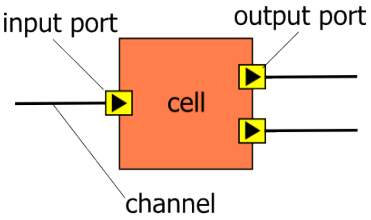


Fig. 9: Cell

Cells can be interconnected with each other and combined into a **cellnet**. A cellnet structure is shown in Fig. 10.

Thus cellnet represents a composite component that can also be interconnected with cells or cellnets. An example of such connection is shown in Fig. 11.

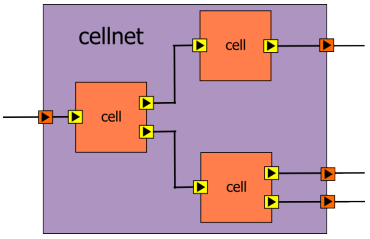


Fig. 10: Cellnet

In hardware a cell is implemented as a configurable dedicated component like multiplier, filter, UART, timer or soft-processor.

High-level abstraction of cells and sellnets hides the low-level details of FPGA implementation into reusable, independent components. *ActiveCells* was developed with the idea to permit implementation of complex things clearly.

ActiveCells is highly extensible platform at both low and high levels. Low level implementation of cells in HDL code can be easily added to the library and used in designs. High-level extensibility is provided by cellnets.

ActiveCells uses high-level language that is an extension of *Active Oberon* language. *Active Oberon* is a general-purpose high-level programming language that combines power and flexibility with minimal and clear syntax. *Active Oberon* language follows the tradition of the Pascal, Modula, Modula-2, Oberon and Oberon-2 languages that were created by Niklaus Wirth, prominent Swiss computer scientist and pioneer computer engineer. The philosophy of *Active Oberon* is to keep things as simple as possible but to be able to implement any complex task.

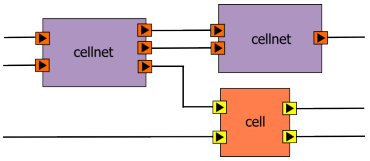


Fig. 11: Cellnets and cells interconnection

3.2 Language

The *ActiveCells* Language has hardware definition and software description structures. Hardware definition structures are cells and cellnets that are also appears as special language composite data types. In addition to these hardware definition structures, keywords are used to handle cell interconnections and port delegations. Software description structures include data types, expressions, statements, procedures and modules.

3.2.1 Cells and Cellnets

As it was defined above, a cell is a processing element that performs some specific activity. Generally a cell can be used without knowledge of how it is implemented. Cells can communicate with each other by means of buffered channels. To write data into channels and read data from them cells use ports.

Each cell has a specific interface that defines name, input, output ports and parameters of configuration. Examples of cell interfaces are shown below:

```
Fifo = cell{Engine,DataWidth=32,Length=32}
         (input: port in; output: port out)
end Fifo;

Controller = cell{Arch="TRM",CodeMemorySize=4096, DataMemorySize=4096}
         (uartOut: port out; uartInp: port in);
begin
...
end Controller;
```

After keyword **cell** within {...} the first parameter defines the type of **cell**: the Engine keyword specifies dedicated hardware component, Arch="TRM" specifies a TRM architecture soft-processor. After the type parameter custom capability parameters of a cell are defined. These parameters are static, their values define cell properties at time of instantiation.

Cells are declared, instantiated and connected (or delegated) within a cellnet. A cellnet can provide its own interface and delegate ports of internal cells. Such a structure provides more abstraction and encapsulation to clarify design of complex systems. A cellnet can also be a most top structure, not having any interface, but creating a complete hardware architecture ready for implementation. The "life-cycle" of cell is shown below:

1. **Declaration**: each cell should be declared in **var** section of cellnet. For example:

```
cellnet System;
var
   ...
   uartTx{Id=0,Baudrate=115200,RtsCtsEnable=0}: Engines.UartTx;
begin
   ...
end System;
```

2. **Instantiation**: each cell should be instantiated in **begin** ... **end** section of cellnet using operator **new**. For example:

```
cellnet System;
var
   ...
   uartTx{Id=0,Baudrate=115200,RtsCtsEnable=0}: Engines.UartTx;
begin
   ...
   new(uartTx);
end System;
```

3. **Connection**: cell should be connected using operator **connect** in **begin** ... **end** section of cellnet. For example:

```
cellnet System;
var
   ...
   uartTx{Id=0,Baudrate=115200,RtsCtsEnable=0}: Engines.UartTx;
begin
   new(uartTx);
   connect(controller.uartOut,uartTx.input,64);
end System;
```

   The **connect** statement connects output ports of one cell with input port of another cell trough buffered channels (FIFO). The depth of channel (length of FIFO) could be defined by third argument of **connect** statement. If third argument is omitted the default FIFO length is used.

4. **Delegation**: ports of cell can be delegated outside of cellnet using operator **delegate**. This allows the programmer to create hierarchies based on previously defined cell types. For example:

```
cellnet System;
import
   Engines;
const
   NumChans = 2;
type
   ScalarProd = cellnet(cfg: port in;
                        x: array NumChans of port in;
                        y: array NumChans of port in;
                        s: port out);
var
   k: longint;
   mult: array NumChans of Engines.MultInt;
   sum: Engines.AddSubInt;
begin
   for k := 0 to len(mult) - 1 do
      new(mult[k]);
      delegate(x[k], mult[k].input[0]]);
      delegate(y[k], mult[k].input[1]]);
```

```
    end;
    new(sum);
    delegate(cfg, sum.cfgInp);
    connect(mult[0].output, sum.input[0]);
    connect(mult[1].output, sum.input[1]);
    delegate(sum.output, s);
end MultiChannelFilter;

var
    ...
begin
    ...
end System;
```

The type **ScalarProd** can now be used as a component in a system design of for creating another level of hierarchy together with other existing cell and cellnet types.

Capability parameters of cell can be specified in three ways:

1. Within cell type definition:

```
type
    UartTx = cell{Engine,Baudrate=921600,RtsCtsEnable=0}(input: port in
    );
    end UartTx;
```

2. Within variable definition:

```
var
    uartTx{Id=0,Baudrate=115200,RtsCtsEnable=0}: Engines.UartTx;
```

3. At the time of cell instantiation:

```
new(uartTx{Id=0,Baudrate=115200,RtsCtsEnable=0});
```

Capability parameters are mapped into hardware properties of corresponding hardware component.

When using soft-processor cells (e.g. TRM) programmer can access the ports from program code using send and receive operations. Send and receive operations can be *blocking* and *non-blocking*.

A blocking receive operation is expressed using operator `"?"`. For example: `inputCmp ? cmd` : in this case the cell activity is blocked until a data item becomes available from the port `inputCmp`. When data is available it is written into variable `cmd`.

A non-blocking receive is expressed using operator `"??"`. If data is available on the specified port, the operator receives the data to a specified memory location and returns a boolean status `true`. In case of data absence the operator returns `false` and does not lead to blocking of the cell's activity.

In a similar way the language defines blocking send (operator "!") and non-blocking send (operator "!!") operations.

The example below shows usage of send and receive operation within soft-processor cell:

```
cellnet SimpleSystem;
import
   Engines;
type
   Controller = cell{Arch="TRM",CodeMemorySize=4096,DataMemorySize=4096}(
                  input: port out;
                  output: port in;
                  );
   var
      x,y: longint;

      procedure process(x: longint);
      begin
         ...
      end process;
   begin

      loop
         input ? x; (* blocking receive *)
         y := process(x);
         output ! y; (* blocking send *)
      end;

   end Controller;
var
   controller: Controller;
   engine: Engines.CustomEngine;
begin
   new(controller);
   new(engine);
   connect(controller.output,engine.input,16);
   connect(engine.output,controller.input,16);
end SimpleSystem.
```

3.2.2 Data Types

A data type must be assigned to each program identifier during its declaration. *ActiveCells* contains basic types, composite types and special port types. Basic data types names, sizes, range values and descriptions are shown in Table 1.

Type name	Size	Range	Description
shortint[1]	8 bits (1 byte)	$-128 \ldots +127$	signed integer number
integer[1]	16 bits (2 byte)	$-32768 \ldots +32767$	signed integer number
longint	32 bits (4 bytes)	$-2'147'483'648 \ldots +2'147'483'647$	signed integer number
hugeint	64 bits (8 bytes)	$-2^{63} \ldots +2^{63}-1$	signed integer number
real[2]	32 bits (4 bytes)	$-3.4028^{38} \ldots +3.4028^{38}$	single precision floating-point number
set	32 bits (4 bytes)	$\{\} \ldots \{0..31\}$	a set of integers in 0..31
char[1]	8 bits (1 byte)	$-128 \ldots +127$	character
boolean[1]	8 bits (1 byte)	false or true	logical type

Table 1: Basic Data Types

Examples of basic data types declaration and assignments are shown below:

```
...
var
   a,b,c: longint;
   x,y: real;
   s1,s2,s3: set;
   c1,c2: char;
   b: boolean;
begin
   a := 12345;
   b := longint(32475); (* explicit longint conversion *)
   c := 0xFF; (* hex number *)

   x := 5.67;
   y := 6E7;

   s1 := {}; (* empty set *)
   s2 := {0,3,10};
   s2 := {0,5..12,31};

   c1 := 'a';
   c2 := chr(55);

   b := true;
end;
...
```

Composite data types are shown in Table 2.

[1]Due to the compiler current limitations shortint, integer, char and boolean data types are stored and addressed as 32-bit number in TRM soft-processor memory.

[2]By default real data type will be simulated by TRM soft-processor runtime. For native floating-point support TRM with FPU should be instantiated. The longreal data type is not implemented.

Type name	Description	Possible compositions
`array`	Collection of elements of same type. Elements are selected via index.	`array of` [Type]; `array 10 of` [Type]; `array N of` [Type]; `array 10 of array 10 of` [Type];
`record`	Collection of named fields of arbitrary type	Person = **record** name: `array 32 of char`; age: `longint`; **end**;
`pointer`	Address of record or array	`pointer to` Person; `pointer to array of` [Type]; `pointer to record` x: `real`; y: `longint`; **end**;
procedure type	Pointer to procedure	`var write: procedure (ch: char);`

Table 2: Composite Data Types

An array is a collection of elements all of the same type. Array elements are selected via an index. Arrays are indexed with integers. First element has index 0.

A record is a collection of named fields of arbitrary type.

A pointer variable contains the address of an array or a record, or it has the value `nil` that means that pointer does not point to any array or record. Pointer variable should allocated with **new** operator at run time specifying size, for example **new**`(p,32)`. Pointer `p` is an address of variable in memory to obtain value of this variable operation `p^` is used. The fact that a pointer is the address of heap memory block is hidden by the language: it is an internal implementation detail. This is different from pointer in the C language, which are explicitly addresses. Code like:

```
p: pointer to Something;
...
p := 01045fCH;
```

is forbidden: the programmer does not see the address part of the pointer.

Procedure type is a pointer to procedure that can be assigned to any procedure with the same interface during run time.

Examples of composite data type declarations and assignments are shown below:

```
...
type
   Person = record
      name: array 32 of char;
      age: longint;
   end;
var
   a: array 10 of longint;
   x: array 10 of array 10 of real;
```

```
    c: pointer to array of char;
    b: array 10 of boolean;
    i,j: longint;
    p: Person;
begin

    a[0] := 123; (*assign 123 to first element of "a" with index "0"*)
    a[9] := 0xFF; (*assign 0xFF to last element of "a" with index "9"*)

    i:=1;
    a[i] := 44; (*assign 44 to second element of "a" with index "i=1"*)

    x[1,2] := 12.3; (*assign 12.3 to element of array x with
                 indexes 1,2*)

    i:=5; j:=7;
    x[i,j] := a[0];   (*assign value of first element in array "a" to
                 element of array x with indexes 5,7*)

    new(c,16); (*allocate array of size of 16 elements*)
    c[0] := 'a';
    c[15] := chr(55);

    b[0] := true;

    p.name := "Mike";
    p.age := 30;

end;
...
```

Special data types called Ports are shown in Table 3.

Type name	Description
`port in`	AXI4-Stream input port
`port out`	AXI4-Stream output port

Table 3: Ports

Ports can be two types: input and output. Data can be written into output port and be read from input port.

3.2.3 Expressions

Expressions describe the computation of values and consist of operators and operands. Expressions are shown in Table 4.

Expression type	Operators	Result type
Arithmetic	`+, -, *, /, div, mod`	numeric
Boolean	`&, or, ~`	`boolean`
Relational	`=, #, <, <=, >, >=, in`	`boolean`
Set	`+, -, *, /`	`set`

Table 4: Expressions

Operator	Operation	Example
`[]`	array access	`a[2] := 5;`
`.`	access to elements in objects and modules	`obj.x := 3;`
`^`	access to pointer data	`p^ := 4;`
`(type)`	type casting	`obj(ObjectType).x := 3;`
`is`	type check	`if obj is ObjectType then ...`

Table 5: Additional operators that used in expressions

Operation with **set** data type are described in Table 6.

Operator	Meaning	Example
+	Union	`{0..7} + {5..9} = {0..9}`
-	Difference (x-y = x*(-y))	`{0..7} - {5..9} = {0..4}`
*	Intersection	`{0..7}*{5..9} = {5..7}`
/	Symmetric difference (x/y = (x-y)+(y-x))	`{0..7}/{5..9} = {0..4, 8..9}`

Table 6: Set operations

3.2.4 Statements

ActiveCells provides elementary statements (assignment, procedure call, return, exit), structured statements for selection (if, case) and iteration (while, repeat, for, loop). Statements with descriptions and example of usage are shown in Table 7.

Statement	Example of usage	Description
Assignment	`k := 1;` `x[k] := 5;` `name := "John";`	Variable x is assigned with value of 1. The element of array x with index 1 is assigned by value of 5. Variable name is assigned with "John" string.
Procedure call	`name := getName(0);` `run(flag);`	"getName()" is called with argument 0, the result is assigned to "name"; run() is called with "flag" argument without a variable return
return	`procedure getName(k: longint):` `  array [64] of char;` `begin` `  if k=0 then return "John"; end;` `  return "Helen";` `end getName;`	procedure getName() returns "John" value if k equal to zero, in other cases it returns "Helen"
if	`if c = 1 then name := "Mike";` `elsif c = 2 then name := "John";` `else name := "Paul";` `end;`	Variable name is assigned with "Mike" value if specified condition is true, i.e. if c is equal to 1 and so on.
case	`case k of` `| 0: name := "John";` `| 1: name := "Paul";` `| 2: name := "Mike";` `else` `  name := "Helen";` `end;`	In case k equals to zero variable name is assigned by "John" string, in case k equals to one variable name is assigned to "Paul" value ans so on, if k equals some other value different from 0, 1 or 2 the variable name is assigned to "Helen" string
while	`k := 0;` `while k<10 do` `  inc(k);` `end;`	Variable k will be incremented while k is lower then 10
loop, **exit**	`loop` `  x[k] := 2*k + 1;` `  inc(k);` `  if k = 10 then` `  exit;` `  end;` `end;`	Elements of array x will be assigned by 2*k+1 values for all k<10
for	`for k:=0 to len(x)-1 do` `  x[k] := 2*k+1;` `end;`	Elements of array x will be assigned 2*k+1 for k=0,1...,len(x)-1
repeat	`k := 0;` `repeat` `  x[k] := 2*k+1;` `  inc(k);` `until x[9] = 0;`	Elements of array x will be assigned 2*k+1 while x[9] different from zero

Table 7: Statements

3.2.5 Procedures

A procedure is an entity that contains executable code. Arguments can be passed to the procedure and result can be returned. An example of procedure is shown below, it performs summation of all elements of input vector, returning result of operation:

```
procedure sum(x: array of real): real;
var
   k: longint;
   s: real;
begin
   for k:=0 to len(x)-1 do
      s := s + x[k];
   end;
   return s;
end sum;
```

Procedures consist of a declaration part, in which constants, types, variables and "nested" procedures, and a statement part (the body), which is executed when the procedure is invoked. The parameters declared in the procedure heading are called formal parameters. They are considered local to the procedure. The parameters specified at the procedure call are termed actual parameters. The procedure call is shown below:

```
...
var
   a: real;
   b: array [10] of real;
begin
   (*array b initializaton*)
   ...
   a := sum(b);
end
...
```

A variable can be passed to a procedure by value: the change of this variable within procedure will change the variable outside:

```
...
procedure operation(var x:real);
begin
   inc(x);
end operation;

var
   a: real;
begin
   a := 1;
   operation(a);
   b := a;  (* a and b are equal 2 *)
end
...
```

A variable can be passed to a procedure as constant: the change of this variable is forbidden:

```
...
procedure operation(const x:real): real;
begin
   return inc(x);
end operation;

var
   a: real;
begin
   a := 1;
   b := operation(a);   (* b equal 2, a equal 1 *)
end
...
```

3.2.6 Modules

Module is a program unit with a clearly defined interface that encapsulates different types, procedures and executable code and generally represents an accomplished functional block, component of more complex program. Module represent a separation of concerns and improve maintainability and reusability of components. Modules are mainly used for soft-processor code encapsulation and abstraction. A general module structure as follows:

```
module SimpleModule;
import (*import section*)
   ...
const (*constant definition section*)
   ...
type (*type definition section*)
   ...

   procedure sum*(); (*procedure (exported)*)
   begin
      ...
   end sum;

   procedure mul(); (* procedure *)
   begin
      ...
   end mul;

   ...
var (*variables declaration section*)
   ...
begin (*module body*)
   ...
end SimpleModule.
```

The exported symbols of a module can be made available to other modules. To make the symbols of module A available in module B, module A has to be imported in module B. To export data type or procedure from module sign "*" is used after exported entity name. Within section **`import`** imported modules are defined.

3.3 Compiler

To compile *ActiveCells* application three steps are generally required:

1. Before compilation of application files of hardware components should be added to the compiler path. This is done by means of command `ActiveCellsComponents.AddPath` where first argument represents the path to components. For example:

   ```
   ActiveCellsComponents.AddPath "ActiveCellsHWLib" ~
   ```

2. To compile project `Compiler.Compile` commands is used. It has following arguments:

   - `-b=TRM`: switches compiler back-end to TRM soft-processor back-end;
   - `--objectFile=Intermediate`: ;
   - `--activeCells`: switches on *ActiveCells* support for Fox Compiler;
   - `FilePathName1.Mod FilePathName2.Mdf ...`: files to compile.

   Example of compilation:

   ```
   Compiler.Compile -b=TRM --objectFile=Intermediate --activeCells
   examples/HelloWorldSpartan3/UartTools.Mod
   examples/HelloWorldSpartan3/HelloWorldSpartan3.Mdf
   ~
   ```

3. To build hardware, generate bitstream, patch it and download to FPGA the `ActiveCellsComponents.BuildHardware` command is used. It has following arguments:

   - `--target="[TargetName]"`: specifies the target;
   - `--outputPath="[OutputPathName]"`: specifies project output directory;
   - `[CellnetName]` : specifies top cellnet name of design that will be built.

   Example of usage:

   ```
   ActiveCellsComponents.BuildHardware --target="Spartan_XC3S200" --
       outputPath="HelloWorldSpartan3" HelloWorldSpartan3 ~
   ```

4 Case Studies

In this section we are describing implementation of digital filter and matrix-vector multiplication using *ActiveCells* . Digital filter is implemented using high-level approach when it is assembled from *Engines* like MultInt, Fifo, Constant etc. This implementation is compared with FirFilter *Engine* implemented solidly in Verilog.

4.1 Digital Signal Filtration

Filtration of digital signals is performed by digital filters. Digital filters can have finite impulse response and infinite impulse response.

FIR (finite impulse response) filter is a digital filter without feedback. Absence of feedback makes impulse filter response to be finite. The equation of the FIR filter is as follows:

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k]. \tag{1}$$

Input $x[n]$ and output $y[n]$ signals are related via the convolution operation. $h[k]$ - filter coefficients. N is a filter length. Block diagram of the FIR filter is shown in Fig. 12.

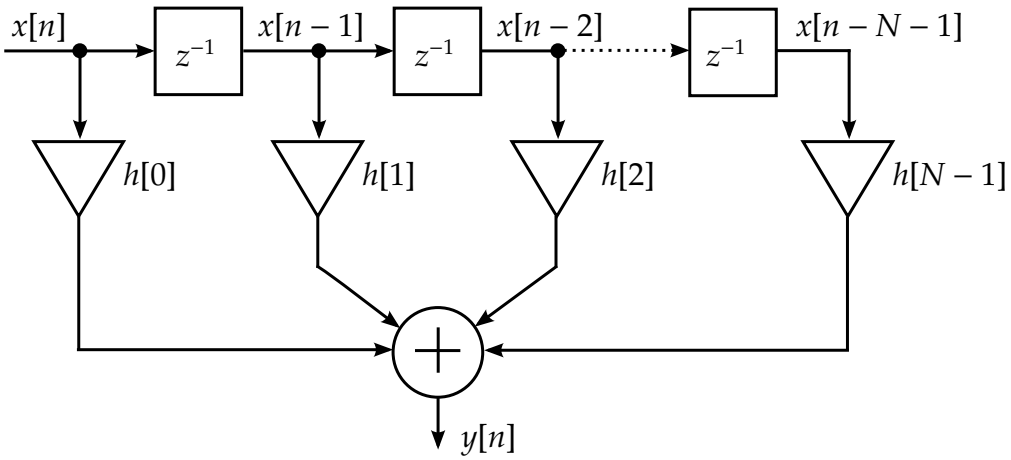


Fig. 12: FIR filter block diagram

Also digital filters can be with infinite impulse response (IIR filters). The advantages of FIR filters compared to IIR filters:

1. FIR filter can be designed with linear phase.
2. FIR filter is simple to implement.

The disadvantages of FIR filters compared to IIR filters:

1. FIR sometimes requires more memory and calculations to achieve a desired response characteristic.

2. Some responses are not practical to implement.

4.1.1 High-level FIR filter implementation

In *ActiveCells* complex components can be built from basic components. As an example consider high-level design of FIR filter described above. As we see from Fig. 12 FIR filter consists of delay, multiplier, summation units.

Listing below shows FIR filter high-level implementation in ActiveCells. As we see cellnet **FirFilter** uses **Fifo**, **StreamRegister**, **Constant**, **MultInt** and **AddSubInt** engines connecting them into filter architecture:

```
FirFilter = cellnet
          (
             input: port in;
             output: port out
          )
var
   k: longint;
   fifo: Engines.Fifo;
   delay: array 2 of Engines.StreamRegister;
   coeff: array 3 of Engines.Constant;
   mult: array 3 of Engines.MulInt;
   add: array 2 of Engines.AddSubInt;
begin
   new(fifo);
   for k:=0 to 1 do
      new(delay[k]{DataWidth=16,Preloaded=1,PreloadValue="0"});
   end;
   new(coeff[0]{DataWidth=16,Value="32'd10"});
   new(coeff[1]{DataWidth=16,Value="32'd20"});
   new(coeff[2]{DataWidth=16,Value="32'd30"});
   for k:=0 to 2 do
      new(mult[k]{InpDataWidth0=16,InpDataWidth1=16,Signed=1,NumStages
=6});
   end;
   for k:=0 to 1 do
      new(add[k]{InpDataWidth0=32,InpDataWidth1=32,InitMode=0});
   end;
   delegate(input, fifo.input);
   connect(fifo.output,delay[0].input);
   connect(fifo.output, mult[0].input[0]);
   connect(coeff[0].output, mult[0].input[1]);
   connect(mult[0].output, add[0].input[1]);
   connect(delay[0].output, delay[1].input);
   connect(delay[0].output, mult[1].input[0]);
   connect(coeff[1].output, mult[1].input[1]);
   connect(mult[1].output, add[0].input[0]);
   connect(add[0].output, add[1].input[1]);
   connect(delay[1].output, mult[2].input[0]);
   connect(coeff[2].output, mult[2].input[1]);
   connect(mult[2].output, add[1].input[0]);
```

```
   delegate(output, add[1].output);
end FirFilter;
```

In *ActiveCells* we can get a visual representation, diagram of **FirFiler** that is shown in Fig. 13.

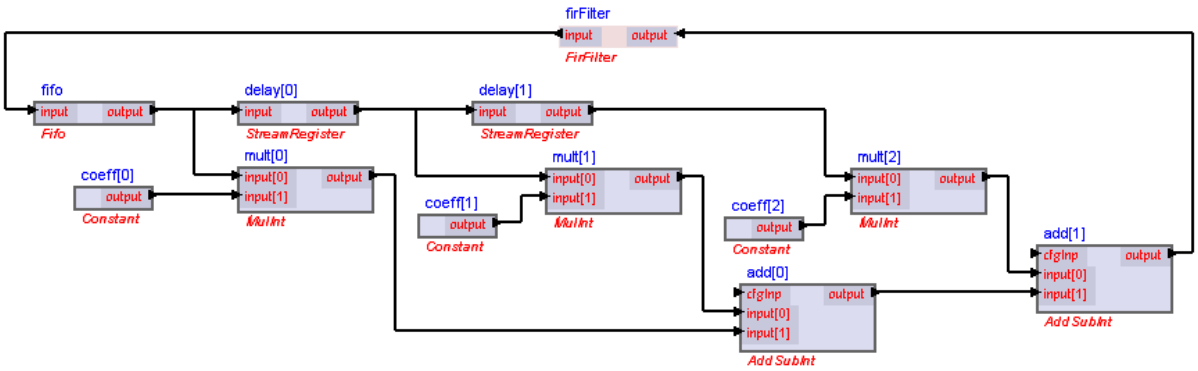


Fig. 13: Fir filter high-level diagram

To test filter we add soft-processor and compile architecture for Spartan-3 Starter Board:

```
cellnet FilterDesignH;
import
   Engines, UartTools, R := StreamReaders, W := StreamWriters;
type
   Controller = cell{Arch="TRM",CodeMemorySize=2048,DataMemorySize=512}(
                  uartOut: port out;
                  uartInp: port in;
                  filterOut: port out;
                  filterInp: port in
                  );
   var
      k,m: longint;
      ch: char;
      x,y: longint;
   begin
      UartTools.init(uartInp,uartOut);
      W.String(UartTools.w,"Testing␣FIR␣Filter␣implemented␣at␣high-level");
    W.Ln(UartTools.w); W.Update(UartTools.w);
      W.String(UartTools.w,"Write␣samples␣to␣filter..."); W.Ln(UartTools.w)
   ; W.Update(UartTools.w);
      filterOut ! -80; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(
   UartTools.w); W.Update(UartTools.w);
      filterOut ! -63; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(
   UartTools.w); W.Update(UartTools.w);
      filterOut ! -46; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(
   UartTools.w); W.Update(UartTools.w);
      filterOut ! -29; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(
   UartTools.w); W.Update(UartTools.w);
```

```
   filterOut ! -12; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(
UartTools.w); W.Update(UartTools.w);
   filterOut ! 5; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(UartTools.
w); W.Update(UartTools.w);
   filterOut ! 22; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(UartTools
.w); W.Update(UartTools.w);
   filterOut ! 39; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(UartTools
.w); W.Update(UartTools.w);
   filterOut ! 56; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(UartTools
.w); W.Update(UartTools.w);
   filterOut ! 73; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(UartTools
.w); W.Update(UartTools.w);
   loop
      uartInp ? x;
      uartOut ! x;
   end;
end Controller;

FirFilter = cellnet
         (
            input: port in;
            output: port out
         )
var
   k: longint;
   fifo: Engines.Fifo;
   delay: array 2 of Engines.StreamRegister;
   coeff: array 3 of Engines.Constant;
   mult: array 3 of Engines.MulInt;
   add: array 2 of Engines.AddSubInt;
begin
   new(fifo);
   for k:=0 to 1 do
      new(delay[k]{DataWidth=16,Preloaded=1,PreloadValue="0"});
   end;
   new(coeff[0]{DataWidth=16,Value="32'd10"});
   new(coeff[1]{DataWidth=16,Value="32'd20"});
   new(coeff[2]{DataWidth=16,Value="32'd30"});
   for k:=0 to 2 do
      new(mult[k]{InpDataWidth0=16,InpDataWidth1=16,Signed=1,NumStages
=6});
   end;
   for k:=0 to 1 do
      new(add[k]{InpDataWidth0=32,InpDataWidth1=32,InitMode=0});
   end;
   delegate(input, fifo.input);
   connect(fifo.output,delay[0].input);
   connect(fifo.output, mult[0].input[0]);
   connect(coeff[0].output, mult[0].input[1]);
   connect(mult[0].output, add[0].input[1]);
   connect(delay[0].output, delay[1].input);
   connect(delay[0].output, mult[1].input[0]);
```

```
      connect(coeff[1].output, mult[1].input[1]);
      connect(mult[1].output, add[0].input[0]);
      connect(add[0].output, add[1].input[1]);
      connect(delay[1].output, mult[2].input[0]);
      connect(coeff[2].output, mult[2].input[1]);
      connect(mult[2].output, add[1].input[0]);
      delegate(output, add[1].output);
   end FirFilter;
var
   controller: Controller;
   firFilter: FirFilter;
   uartTx{Id=0,Baudrate=115200,RtsCtsEnable=0}: Engines.UartTx;
   uartRx{Id=0,Baudrate=115200,RtsCtsEnable=0}: Engines.UartRx;
begin
   new(controller);
   new(uartTx);
   connect(controller.uartOut,uartTx.input,4);
   new(uartRx);
   connect(uartRx.output,controller.uartInp,4);
   new(firFilter);
   connect(controller.filterOut, firFilter.input,1);
   connect(firFilter.output, controller.filterInp,1);
end FilterDesignH.

ActiveCellsComponents.AddPath "ActiveCellsHWLib" ~

Compiler.Compile -b=TRM --objectFile=Intermediate --activeCells
examples/utils/UartTools.Mod
examples/FilterDesign/FilterDesignH.Mdf
~

ActiveCellsComponents.BuildHardware --target="Spartan_XC3S200" --outputPath
   ="FilterDesignHSpartan3" FilterDesignH ~

AcDataFlowModel.build "FilterDesignH" ~

AcDataFlowView.generateView "FilterDesignH.dfm.xml"~

AcDataFlowView.openView "FilterDesignH.dfm.xml"~
```

Top level of obtainig architecture is shown in Fig. 14.

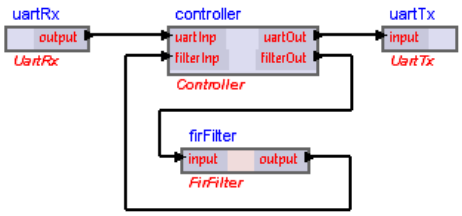


Fig. 14: Top level of FIR filter testing system

After compilation and downloading bit-stream to the target we can see the output in Terminal window:

```
Testing FIR Filter implemented at high-level
Write samples to filter...
-800
-2230
-4120
-3100
-2080
-1060
-40
980
2000
3020
```

4.1.2 Comparison with low-level FIR filter implementation

We compare high-level FIR filter implementation with design that contains FIR filter built in low-level using pure Verilog. In *ActiveCells* FIR filter is represented by **FirFilter** Engine in *ActiveCells Hardware Library* . **FirFilter** could be configured to certain length, pipeline length, output shift, buses length.

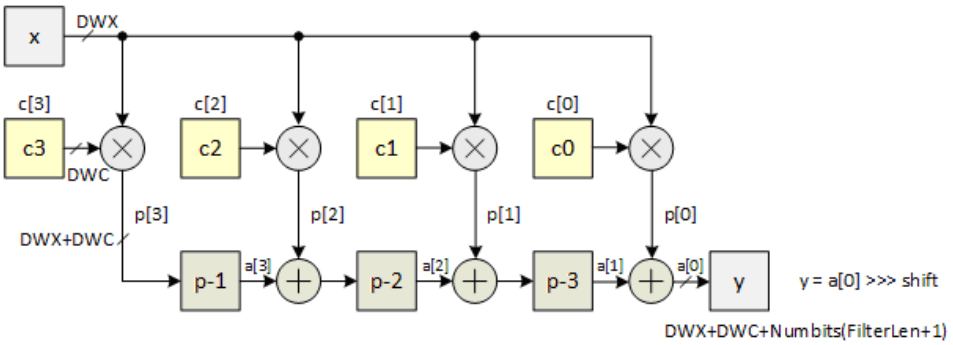


Fig. 15: FirFilter Engine Diagram

This FIR Filter has following properties:

- Has transposed structure
- Has pipline

The testing architecture is shown in listing below:

```
cellnet FilterDesignL;
import
   Engines, UartTools, R := StreamReaders, W := StreamWriters;
type
   Controller = cell{Arch="TRM",CodeMemorySize=2048,DataMemorySize=512}(
```

```
                uartOut: port out;
                uartInp: port in;
                coeffOut: port out;
                shiftOut: port out;
                filterOut: port out;
                filterInp: port in
                );
var
   k,m: longint;
   ch: char;
   x,y: longint;
begin
   UartTools.init(uartInp,uartOut);
   W.String(UartTools.w,"Testing␣FIR␣Filter␣implemented␣at␣low-level");
W.Ln(UartTools.w); W.Update(UartTools.w);
   W.String(UartTools.w,"Write␣coeffitients␣to␣filter..."); W.Ln(
UartTools.w); W.Update(UartTools.w);
   coeffOut ! 10;
   coeffOut ! 20;
   coeffOut ! 30;
   W.String(UartTools.w,"Write␣shift␣to␣filter..."); W.Ln(UartTools.w);
W.Update(UartTools.w);
   shiftOut ! 0;
   W.String(UartTools.w,"Write␣samples␣to␣filter..."); W.Ln(UartTools.w)
; W.Update(UartTools.w);
   filterOut ! -80; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(
UartTools.w); W.Update(UartTools.w);
   filterOut ! -63; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(
UartTools.w); W.Update(UartTools.w);
   filterOut ! -46; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(
UartTools.w); W.Update(UartTools.w);
   filterOut ! -29; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(
UartTools.w); W.Update(UartTools.w);
   filterOut ! -12; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(
UartTools.w); W.Update(UartTools.w);
   filterOut ! 5; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(UartTools.
w); W.Update(UartTools.w);
   filterOut ! 22; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(UartTools
.w); W.Update(UartTools.w);
   filterOut ! 39; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(UartTools
.w); W.Update(UartTools.w);
   filterOut ! 56; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(UartTools
.w); W.Update(UartTools.w);
   filterOut ! 73; filterInp ? y; W.Int(UartTools.w,y,0); W.Ln(UartTools
.w); W.Update(UartTools.w);
   loop
      uartInp ? x;
      uartOut ! x;
   end;
end Controller;
FirFilter = cell{Engine,CoeffWidth=16,InpWidth=16, OutWidth=32,
KernelLength=3, PipeLength=6, InitShift=0}
```

```
                    (coeffInp: port in; shiftInp: port in; input: port in;
     output: port out);
   end FirFilter;
var
   controller: Controller;
   firFilter: FirFilter;
   uartTx{Id=0,Baudrate=115200,RtsCtsEnable=0}: Engines.UartTx;
   uartRx{Id=0,Baudrate=115200,RtsCtsEnable=0}: Engines.UartRx;
begin
   new(controller);
   new(uartTx);
   connect(controller.uartOut,uartTx.input,4);
   new(uartRx);
   connect(uartRx.output,controller.uartInp,4);
   new(firFilter);
   connect(controller.coeffOut, firFilter.coeffInp);
   connect(controller.shiftOut, firFilter.shiftInp);
   connect(controller.filterOut, firFilter.input,1);
   connect(firFilter.output, controller.filterInp,1);
end FilterDesignL.

ActiveCellsComponents.AddPath "ActiveCellsHWLib" ~

Compiler.Compile -b=TRM --objectFile=Intermediate --activeCells
examples/utils/UartTools.Mod
examples/FilterDesign/FilterDesignL.Mdf
~

ActiveCellsComponents.BuildHardware --target="Spartan_XC3S200" --outputPath
   ="FilterDesignLSpartan3" FilterDesignL ~

ActiveCellsComponents.BuildHardware --target="ML505" --outputPath="
   FilterDesignLML505"  FilterDesignL ~


AcDataFlowModel.build "FilterDesignL" ~

AcDataFlowView.generateView "FilterDesignL.dfm.xml"~

AcDataFlowView.openView "FilterDesignL.dfm.xml"
```

Compiling and running this architecture we can see the same output in Terminal Window:

```
Testing FIR Filter implemented at low-level
Write coeffitients to filter...
Write shift to filter...
Write samples to filter...
-800
-2230
-4120
-3100
-2080
```

```
-1060
-40
980
2000
3020
```

It is interesting to compare resources utilization and implementation speed for both cases:

	High-level	**Low-level**
Number of Slice Flip Flops	20%	20%
Number of occupied Slices	72%	58%
Number of RAMB16s	66%	33%
Number of MULT18X18s	25%	25%
Timing constraints	19.7 ns	19.7 ns

Table 8: Resources utilization and speed

As we can see from Table 8 high-level implementation consumes a bit more resources that low-level implementation. However this is not a big price for considerably shorter development time and flexibility that high-level approach provides.

4.2 Matrix-Vector Multiplication

Matrix-Vector Multiplication is important operation of linear algebra and is widely used in various computing algorithms.

If we have matrix **A** with elements $a_{ij},\ i = 1, 2..., M,\ j = 1, 2..., N$ and vector **x** with elements $x_i,\ i = 1, 2..., N$ then matrix-vector multiplication is defined as:

$$\mathbf{b} = \mathbf{A}\mathbf{x} \tag{2}$$

or

$$\begin{bmatrix} b_1 \\ ... \\ b_M \end{bmatrix} = \begin{bmatrix} a_{11} & ... & a_{1N} \\ ... & a_{ij} & ... \\ a_{M1} & ... & a_{MN} \end{bmatrix} \begin{bmatrix} x_1 \\ ... \\ x_N \end{bmatrix} \tag{3}$$

and resulting in new vector **b** with elements $b_i,\ i = 1, 2..., M$.

Each element of vector **b** is a scalar product of corresponding row of matrix **A** and vector **x**:

$$\begin{bmatrix} b_1 \\ ... \\ b_M \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + ... + a_{1N}x_N \\ ... \\ a_{M1}x_1 + ... + a_{MN}x_N \end{bmatrix}. \tag{4}$$

In other words each element of corresponding row of matrix **A** row is multiplied by each element of vector **x** and all these products are summarized to get corresponding

element of vector **b**. To get one element of vector **b** there are N multiplications and $N-1$ summations are needed. To get whole vector **b** there are $M \times N$ multiplications and $M \times (N-1)$ summations are necessary to perform.

Now we are going to implement and test dedicated engine that perform matrix-vector multiplication of martix of size 3×3 and vector of size 3:

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{5}$$

or

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{12}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{bmatrix}. \tag{6}$$

The visual block diagram of matrix-vector multiplication is shown in Fig. 16.

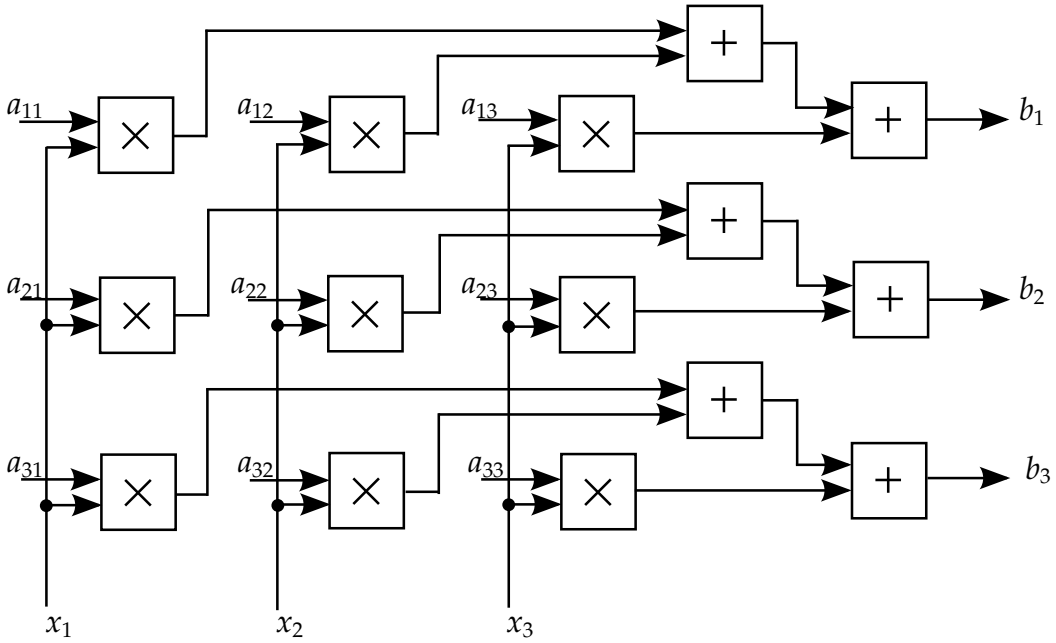


Fig. 16: Block diagram of matrix-vector multiplication

Matrix-vector multiplier uses 9 multiplier units and 6 summation units. The *ActiveCells* high-level implementation is following:

```
MatrixVectorMul3x3 = cellnet
        (
           A: array 3 of array 3 of port in;
           x: array 3 of port in;
           b: array 3 of port out
        )
var
   k,m: longint;
```

```
   fifo: array 3 of Engines.Fifo;
   mult: array 3 of array 3 of Engines.MulInt;
   add: array 3 of array 2 of Engines.AddSubInt;
begin
   for k:=0 to 2 do
      new(fifo[k]);
   end;
   for k:=0 to 2 do
      for m:=0 to 2 do
         new(mult[k,m]{InpDataWidth0=16,InpDataWidth1=16,Signed=1,
NumStages=6});
      end;
   end;
   for k:=0 to 2 do
      for m:=0 to 1 do
         new(add[k,m]{InpDataWidth0=32,InpDataWidth1=32,InitMode=0});
      end;
   end;
   for k:=0 to 2 do
      for m:=0 to 2 do
         delegate(A[k,m], mult[k,m].input[0]);
      end;
   end;
   for k:=0 to 2 do
      delegate(x[k], fifo[k].input);
   end;
   for m:=0 to 2 do
      for k:=0 to 2 do
         connect(fifo[m].output, mult[k,m].input[1]);
      end;
   end;
   connect(mult[0,0].output, add[0,0].input[0]);
   connect(mult[0,1].output, add[0,0].input[1]);
   connect(mult[1,0].output, add[1,0].input[0]);
   connect(mult[1,1].output, add[1,0].input[1]);
   connect(mult[2,0].output, add[2,0].input[0]);
   connect(mult[2,1].output, add[2,0].input[1]);
   connect(add[0,0].output, add[0,1].input[0]);
   connect(mult[0,2].output, add[0,1].input[1]);
   connect(add[1,0].output, add[1,1].input[0]);
   connect(mult[1,2].output, add[1,1].input[1]);
   connect(add[2,0].output, add[2,1].input[0]);
   connect(mult[2,2].output, add[2,1].input[1]);
   delegate(b[0], add[0,1].output);
   delegate(b[1], add[1,1].output);
   delegate(b[2], add[2,1].output);
end MatrixVectorMul3x3;
```

MatrixVectorMul3x3 cellnet diagram generated by *ActiveCells* is shown in Fig. 17.

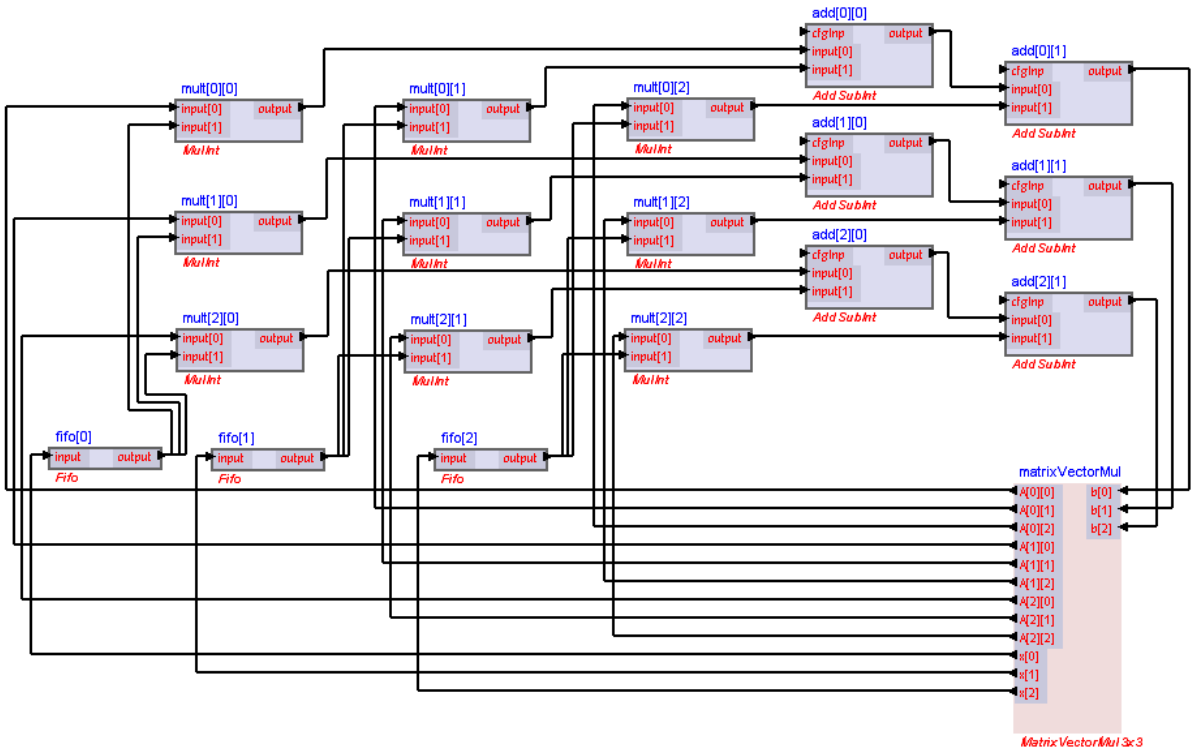


Fig. 17: Matrix Vector Multiplier Viasual Diagram

Now we are going to test MatrixVectorMul3x3 by writing matrix an vector by soft-processor and reading the result of multiplication. The design is implemented on Virtex-5 ML505 FPGA Board. The code is shown below:

```
cellnet MatrixVectorMul;
import
   Engines, UartTools, R := StreamReaders, W := StreamWriters;
type
   Controller = cell{Arch="TRM",CodeMemorySize=2048,DataMemorySize=512}(
                  uartOut: port out;
                  uartInp: port in;
                  A: array 3 of array 3 of port out;
                  x: array 3 of port out;
                  b: array 3 of port in
                  );
   var
      k,m: longint;
      ch: char;
      z: longint;
      res: array 3 of longint;
   begin
      UartTools.init(uartInp,uartOut);
      W.String(UartTools.w,"Testing␣Matrix-Vector␣Multiplication"); W.Ln(
    UartTools.w); W.Update(UartTools.w);
      A[0][0] ! 10;      A[0][1] ! 20;   A[0][2] ! 30;
```

```
   A[1][0] ! 40;     A[1][1] ! 50;  A[1][2] ! 60;
   A[2][0] ! 70;     A[2][1] ! 80;  A[2][2] ! 90;
   x[0] ! 11;
   x[1] ! 22;
   x[2] ! 33;
   b[0] ? res[0]; W.Int(UartTools.w,res[0],0); W.Ln(UartTools.w); W.
Update(UartTools.w);
   b[1] ? res[1]; W.Int(UartTools.w,res[1],0); W.Ln(UartTools.w); W.
Update(UartTools.w);
   b[2] ? res[2]; W.Int(UartTools.w,res[2],0); W.Ln(UartTools.w); W.
Update(UartTools.w);
   loop
      uartInp ? z;
      uartOut ! z;
   end;
end Controller;

MatrixVectorMul3x3 = cellnet
         (
            A: array 3 of array 3 of port in;
            x: array 3 of port in;
            b: array 3 of port out
         )
var
   k,m: longint;
   fifo: array 3 of Engines.Fifo;
   mult: array 3 of array 3 of Engines.MulInt;
   add: array 3 of array 2 of Engines.AddSubInt;
begin
   for k:=0 to 2 do
      new(fifo[k]);
   end;
   for k:=0 to 2 do
      for m:=0 to 2 do
         new(mult[k,m]{InpDataWidth0=16,InpDataWidth1=16,Signed=1,
NumStages=6});
      end;
   end;
   for k:=0 to 2 do
      for m:=0 to 1 do
         new(add[k,m]{InpDataWidth0=32,InpDataWidth1=32,InitMode=0});
      end;
   end;
   for k:=0 to 2 do
      for m:=0 to 2 do
         delegate(A[k,m], mult[k,m].input[0]);
      end;
   end;
   for k:=0 to 2 do
      delegate(x[k], fifo[k].input);
   end;
   for m:=0 to 2 do
```

```
         for k:=0 to 2 do
            connect(fifo[m].output, mult[k,m].input[1]);
         end;
      end;
      connect(mult[0,0].output, add[0,0].input[0]);
      connect(mult[0,1].output, add[0,0].input[1]);
      connect(mult[1,0].output, add[1,0].input[0]);
      connect(mult[1,1].output, add[1,0].input[1]);
      connect(mult[2,0].output, add[2,0].input[0]);
      connect(mult[2,1].output, add[2,0].input[1]);
      connect(add[0,0].output, add[0,1].input[0]);
      connect(mult[0,2].output, add[0,1].input[1]);
      connect(add[1,0].output, add[1,1].input[0]);
      connect(mult[1,2].output, add[1,1].input[1]);
      connect(add[2,0].output, add[2,1].input[0]);
      connect(mult[2,2].output, add[2,1].input[1]);
      delegate(b[0], add[0,1].output);
      delegate(b[1], add[1,1].output);
      delegate(b[2], add[2,1].output);
   end MatrixVectorMul3x3;
var
   k,m: longint;
   controller: Controller;
   matrixVectorMul: MatrixVectorMul3x3;
   uartTx{Id=0,Baudrate=115200,RtsCtsEnable=0}: Engines.UartTx;
   uartRx{Id=0,Baudrate=115200,RtsCtsEnable=0}: Engines.UartRx;
begin
   new(controller);
   new(uartTx);
   connect(controller.uartOut,uartTx.input,4);
   new(uartRx);
   connect(uartRx.output,controller.uartInp,4);
   new(matrixVectorMul);
   for k:=0 to 2 do
      for m:=0 to 2 do
         connect(controller.A[k,m], matrixVectorMul.A[k,m],1);
      end;
   end;
   for k:=0 to 2 do
      connect(controller.x[k], matrixVectorMul.x[k],1);
      connect(matrixVectorMul.b[k], controller.b[k],1);
   end;
end MatrixVectorMul.

ActiveCellsComponents.AddPath "ActiveCellsHWLib" ~

Compiler.Compile -b=TRM --objectFile=Intermediate --activeCells --
   patchSpartan6
examples/utils/UartTools.Mod
examples/MartixVectorMul/MatrixVectorMul.Mdf
~
```

```
ActiveCellsComponents.BuildHardware --target="AVSP6LX75T" --outputPath="
    MatrixVectorMulSpartan6" MatrixVectorMul ~

ActiveCellsComponents.BuildHardware --target="ML505" --outputPath="
    MatrixVectorMulML505"  MatrixVectorMul ~


AcDataFlowModel.build "MatrixVectorMul" ~

AcDataFlowView.generateView "MatrixVectorMul.dfm.xml"~

AcDataFlowView.openView "MatrixVectorMul.dfm.xml"~
```

Top-level diagram of testing hardware architecture is shown in Fig. 18.

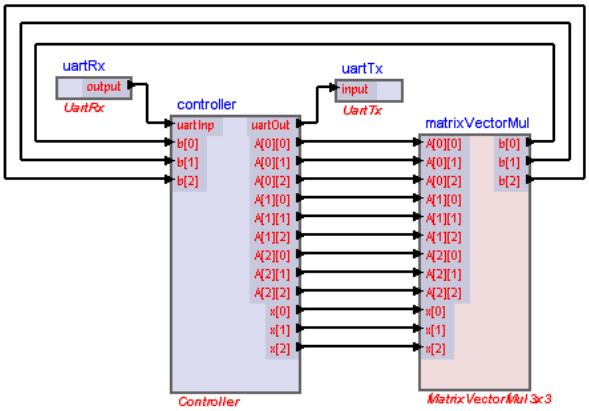


Fig. 18: Top level of Matrix-Vector Multiplier Test

The result of test is shown below:

```
Testing Matrix-Vector Multiplication
1540
3520
5500
```

5 *ActiveCells Hardware Library*

5.1 Introduction

ActiveCells Hardware Library contains a collection of hardware components, which includes processor cores (e.g. TRM) and Engines – specialized hardware components used for arithmetic and memory operations, communication and other dedicated computational tasks (e.g. digital filtering).

Hardware components in *ActiveCells Hardware Library* are implemented using Hardware Description languages (HDL). Currently, all components are written in Verilog. Components can be implemented either manually by an experienced hardware engineer or using automated tools (e.g. using CORE Generator from Xilinx). The interface of the corresponding HDL code has to obey AXI4-Stream communication protocol.

Hardware components are located in `ac/ActiveCellsHWLib` directory of *ActiveCells* . *ActiveCells Hardware Library* includes the following hardware components:

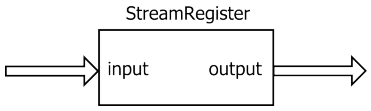
1. **StreamRegister**. AXI4-Stream register.
2. **Constant**. Constant value.
3. **Fifo**. First Input First Output (FIFO) buffer.
4. **MulInt**. Integer multiplier.
5. **AddSubInt**. Integer adder/subtractor.
6. **BitShifter**. Bitwise shift operator.
7. **CompareInt**. Integer comparison operator.
8. **BitwiseBinaryOp**. Binary bitwise operator.
9. **NegateInt**. Integer negate operator.
10. **Switch**. AXI4-Stream switch.
11. **StreamDemux2**. AXI4-Stream demultiplexer with two outputs.
12. **StreamMux2**. AXI4-Stream multiplexer with two inputs.
13. **StreamMerger2**. Merges two AXI4-Stream inputs.
14. **StreamDivider2**. Divides (splits) an AXI4-Stream input into two streams with predefined bit-width.
15. **StreamRepeater2**. Two-channel AXI4-Stream repeater.
16. **Gpi**, **GpiDiff**. General-purpose input.

17. **Gpo**, **GpoDiff**. General-purpose output.
18. **Gpio**. Configurable general-purpose input/output.
19. **Timer**. Configurable timer.
20. **TRM**. RISC Soft-processor.
21. **UartTx**. UART data transmitter.
22. **UartRx**. UART data receiver.
23. **PwmGen1**. Configurable Pulse Width Modulation (PWM) signal generator.
24. **LEDDigits**. LED display driver for Spartan-3 Starter Board.
25. **FirFilter**. Digital FIR filter.

5.2 Description of hardware components

Below we present descriptions of the hardware components available in the *ActiveCells Hardware Library* . Interfaces of those hardware components are defined in the module `ac/src/Engines.Mdf`.

5.2.1 StreamRegister

Diagram	Interface
	`StreamRegister* = cell{Engine,` `  DataWidth=32,` `  Preloaded=0,` `  PreloadValue="0",` `  PersistentOutput=0` `}` `(` `  input: port in;` `  output: port out` `);` `end StreamRegister;`

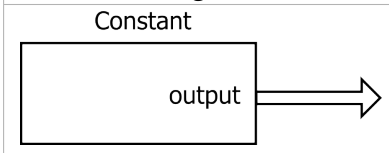
AXI4-Stream register
Ports:

- `input`: input data port
- `output`: output data port

Parameters:

- `DataWidth`: register data width in number of bits
- `Preloaded`: if non-zero the register will be preloaded with a given "PreloadValue" at the reset time
- `PreloadValue`: data value to be used for preloading the register at the reset time
- `PersistentOutput`: if non-zero the register will produce continuous output of the current value (the data at register's input will be always accepted – non-blocking input)

5.2.2 Constant

Diagram	Interface
	`Constant* = cell{Engine,` `  DataWidth=32,` `  Value="32'd0"` `}` `(` `  output: port out` `);` `end Constant;`

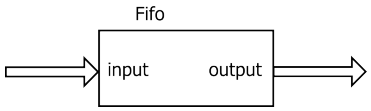
A constant value.
Ports:

- `output`: output data port

Parameters:

- `DataWidth`: data width in number of bits
- `Value`: string with constant value

5.2.3 Fifo

Engine Diagram	Interface Description
	`Fifo* = cell{` `  Engine,` `  DataWidth=32,` `  Length=32` `}` `(` `  input: port in;` `  output: port out` `)` `end Fifo;`

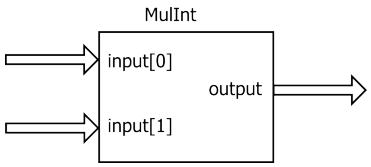
AXI4-Stream First Input First Output (FIFO) buffer.
Ports:

- `input`: data input
- `output`: data output

Parameters:

- `DataWidth`: data width in bits
- `Length`: FIFO length (depth) in number of elements

5.2.4 MulInt

Diagram	Interface
	`MulInt* = cell{Engine,` `  InpDataWidth0=32,` `  InpDataWidth1=32,` `  Signed=1,` `  NumStages=6` `(` `  input: array 2 of port in;` `  output: port out` `)` `end MulInt;`

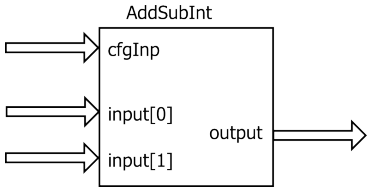
Integer multiplier.
Ports:

- `input[0]`: first argument input
- `input[1]`: second argument input
- `output`: result output

Parameters:

- `InpDataWidth0`: data width of first argument in number of bits
- `InpDataWidth1`: data width of second argument in number of bits
- `Signed`: non-zero for signed multiplication
- `NumStages`: number of multiplication pipeline stages (optional, – if omitted the parameter will be automatically set by *ActiveCells* compiler according to the specified data width parameters)

5.2.5 AddSubInt

Diagram	Interface
	`AddSubInt* = cell{Engine,` `  InpDataWidth0=32,` `  InpDataWidth1=32,` `  Signed=1,` `  InitMode=0` `}` `(` `  cfgInp: port in;` `  input: array 2 of port in;` `  output: port out` `)` `end AddSubInt;`

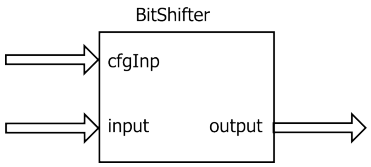
Performs integer addition or subtraction depending on the chosen configuration.
Ports:

- `cfgInp`: single bit configuration input (0 for addition, 1 for subtraction)
- `input[0]`: first argument input
- `input[1]`: second argument input
- `output`: result output

Parameters:

- `InpDataWidth0`: data width of first argument in number of bits
- `InpDataWidth1`: data width of second argument in number of bits
- `Signed`: non-zero for signed operations
- `InitMode`: initial operation mode

5.2.6 BitShifter

Diagram	Interface
	`BitShifter* = cell{` `  Engine,` `  InpDataWidth=32,` `  OutDataWidth=32,` `  InitMode=0,` `  InitShift=0` `}` `(` `  cfgInp: port in;` `  input: port in;` `  output: port out` `)` `end BitShifter;`

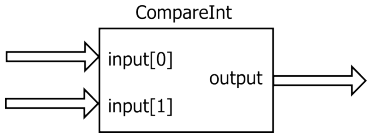
Bitwise shift operator.
Ports:

- `cfgInp`: 32-bit configuration input with 3 least significant bits for configuring the mode (0 for arithmetic shift, 1 for logical shift, 2 for rotation), and 29 most significant bits for configuring signed value of the shift
- `input`: data input
- `output`: result output

Parameters:

- `InpDataWidth`: input data width in number of bits
- `OutDataWidth`: output data width in number of bits
- `InitMode`: initial value of shift in number of bits (positive number for shift left, negative number of shift right)
- `InitShift`: initial mode (0 for arithmetic shift, 1 for logical shift, 2 for rotation)

5.2.7 CompareInt

Diagram	Interface
	`CompareInt* = cell{Engine,` `  InpDataWidth0=32,` `  InpDataWidth1=32,` `  Signed=1` `}` `(` `  cfgInp: port in;` `  input: array 2 of port in;` `  output: port out` `)` `end CompareInt;`

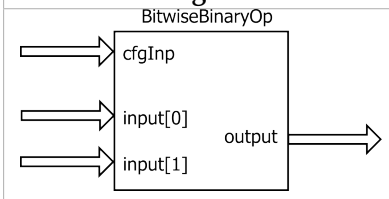
Integer comparison operator.
Ports:

- `cfgInp`: 3-bit configuration input (0 for "=", 1 for "#", 2 for ">", 3 for ">=", 4 for "<", 5 for "<=")
- `input[0]`: first argument
- `input[1]`: second argument
- `output`: operation result

Parameters:

- `InpDataWidth0`: input 0 data width in bits
- `OutDataWidth1`: input 1 data width in bits
- `Signed`: non-zero for signed operation

5.2.8 BitwiseBinaryOp

Diagram	Interface
	`BitwiseBinaryOp* = cell{Engine, DataWidth=32, InitMode=0} ( cfgInp: port in; input: array 2 of port in; output: port out ) end BitwiseBinaryOp;`

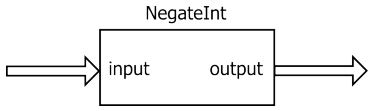
Binary bitwise operator
Ports:

- `cfgInp`: 2-bit configuration input (0 for "AND", 1 for "OR", 2 for "XOR", 3 for "XNOR")
- `input[0]`: first argument input
- `input[1]`: second argument input
- `output`: operation result output

Parameters:

- `DataWidth`: data width in number of bits
- `InitMode`: initial operation mode (0 for "AND")

5.2.9 NegateInt

Diagram	Interface
	`NegateInt* = cell{Engine, DataWidth=32, Signed=1} ( input: port in; output: port out ) end NegateInt;`

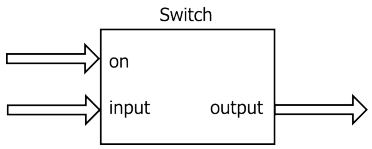
Integer negate operator
Ports:

- `input`: data input
- `output`: result output

Parameters:

- `DataWidth`: data width in number of bits
- `Signed`: non-zero for signed operation

5.2.10 Switch

Diagram	Interface
	`Switch* = cell{Engine,` `  DataWidth=32,` `  InitState=0` `}` `(` `  on: port in;` `  input: port in;` `  output: port out` `)` `end Switch;`

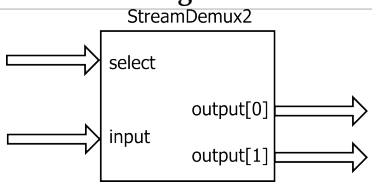
AXI4-Stream switch.
Ports:

- `on`: switch configuration input (0 for ”off”, 1 for ”on”)
- `input`: data input
- `output`: data output

Parameters:

- `DataWidth`: data width in number of bits
- `InitState`: initial state after reset: 0 off, 1 on

5.2.11 StreamDemux2

Diagram	Interface
	`StreamDemux2* = cell{Engine,` `  DataWidth=32,` `  InitSelect=0` `}` `(` `  select: port in;` `  input: port in;` `  output: array 2 of port out` `)` `end StreamDemux2;`

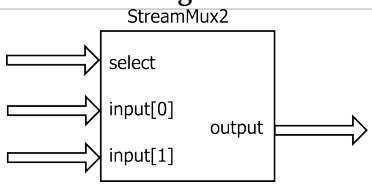
AXI4-Stream demultiplexer with two outputs
Ports:

- `select`: selection input (0 for output[0], 1 for output[1])
- `input`: input to demultiplex
- `output[0]`: output 0
- `output[1]`: output 1

Parameters:

- `DataWidth`: data width in number of bits
- `InitState`: initial output channel selection

5.2.12 StreamMux2

Diagram	Interface
	`StreamMux2* = cell{Engine,` `  DataWidth=32,` `  InitSelect=0` `}` `(` `  select: port in;` `  input: array 2 of port in;` `  output: port out` `)` `end StreamMux2;`

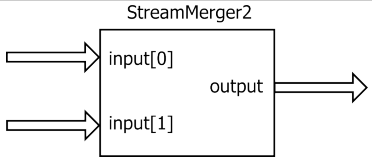
AXI4-Stream multiplexer with two inputs
Ports:

- `select`: selection channel (0 for input[0], 1 for input[1])
- `input[0]`: input 0
- `input[1]`: input 1
- `output`: data output

Parameters:

- `DataWidth`: data width in bits
- `InitState`: initial input channel selection

5.2.13 StreamMerger2

Diagram	Interface
	`StreamMerger2* = cell{Engine, InpDataWidth0=32, InpDataWidth1=32 } ( input: array 2 of port in; output: port out ) end StreamMerger2;`

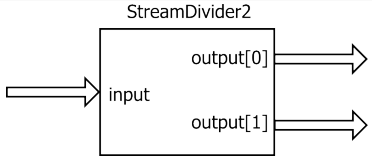
Merges two AXI4-Stream inputs
Ports:

- `input[0]`: input 0: least significant bits of the resulted output
- `input[1]`: input 1: most significant bits of the resulted output
- `output`: merging result

Parameters:

- `InpDataWidth0`: bitness of the first input
- `InpDataWidth1`: bitness of the second input

5.2.14 StreamDivider2

Diagram	Interface
	`StreamDivider2* = cell{Engine, InpDataWidth=64, OutDataWidth0=32 } ( input: port in; output: array 2 of port out ) end StreamDivider2;`

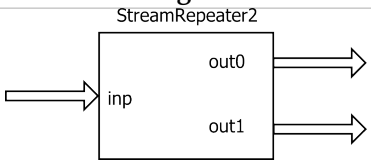
Divides (splits) an AXI4-Stream input into two streams with predefined bit-width
Ports:

- `input`: input stream to divide
- `output[0]`: first output stream (DWO1 least significant bits of the input stream)
- `output[1]`: second output stream (DWI-DWO1 most significant bits of the input stream)

Parameters:

- `InpDataWidth`: bitness of the input to divide
- `OutDataWidth0`: bitness of the first output stream (least significant bits of the input)

5.2.15 StreamRepeater2

Diagram	Interface
	``` StreamRepeater2* = cell{Engine, DataWidth=32 } (inp: port in; out0, out1: port out) end StreamRepeater2; ```

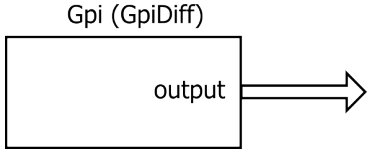
Two-channel AXI4-Stream repeater
Ports:

- `ino`: data input
- `out0`: data output 0
- `out1`: data output 1

Parameters:

- `DataWidth`: input data width in number of bits

5.2.16 Gpi, GpiDiff

Diagram	Interface
	``` Gpi* = cell{Engine, DataWidth=8} (output: port out) end Gpi; GpiDiff* = cell{Engine, DataWidth=8} (output: port out) end GpiDiff; ```

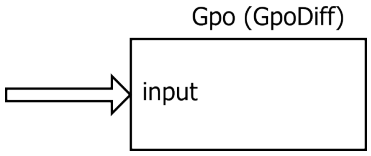
General-purpose input
Ports:

- `output`: port out

Parameters:

- `DataWidth`: number of GPI bits

5.2.17 Gpo, GpoDiff

Diagram	Interface
	`Gpo* = cell{Engine, DataWidth=8} ( input: port in ) end Gpo;` `GpoDiff* = cell{Engine, DataWidth=8} ( input: port in ) end GpoDiff;`

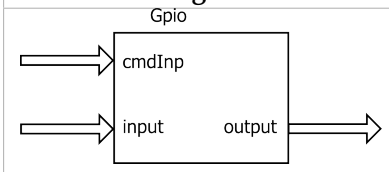
General-purpose output
Ports:

- `input`: input port

Parameters:

- `DataWidth`: number of GPO bits

5.2.18 Gpio

Diagram	Interface
	`Gpio* = cell{Engine, DataWidth=8} ( cmdInp: port in; input: port in; output: port out ) end Gpio;`

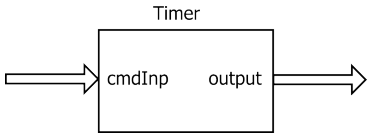
General-purpose input/output.
Ports:

- `cmdInp`: configuration input (most significant bit (DataWidth-th bit) defines the command: 0 - setup IO direction for every GPIO pin (0 for input, 1 for output) , 1 - readout of the state of all GPIO pins; command data is contained in DataWidth least significant bits)
- `input`: data input
- `output`: data output

Parameters:

- `DataWidth`: number of GPIO pins

5.2.19 Timer

Diagram	Interface
	`Timer* = cell{Engine,` `  CounterWidth=32,` `  Inc=1` `}` `(` `  cmdInp: port in;` `  output: port out` `)` `end Timer;`

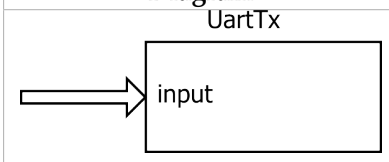
Configurable timer.
Ports:

- `cmdInp`: timer command input; 0 for resetting the timer counter, 1 for sampling the timer counter
- `output`: timer counter output; data becomes available every time when sampling command has been issued

Parameters:

- `CounterWidth`: width of the timer counter in bits
- `Inc`: 1 for incrementing counter, 0 for decrementing counter

5.2.20 UartTx

Diagram	Interface
	`UartTx* = cell{Engine}` `(` `  input: port in` `);` `end UartTx;`

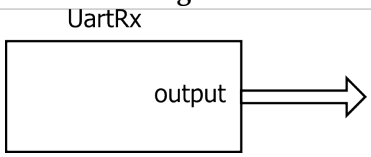
UART data transmitter
Ports:

- `output`: received data output (8 bit wide)

Parameters:

- `Baudrate`: data rate in baud
- `RtsCtsEnable`: 1 to enable RTS/CTS signaling, 0 to disable it

5.2.21 UartRx

Diagram	Interface
	`UartRx* = cell{Engine}` `(` `  output: port out` `);` `end UartRx;`

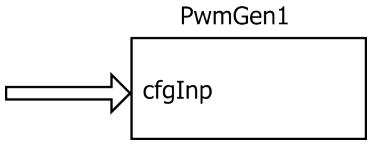
UART data receiver
Ports:

- `output`: received data output (8 bit wide)

Parameters:

- `Baudrate`: data rate in baud
- `RtsCtsEnable`: 1 to enable RTS/CTS signaling, 0 to disable it

5.2.22 PwmGen1

Diagram	Interface
	`PwmGen1* = cell{` `  Engine,` `  MaxPeriod=536870911,` `  InitPeriod=2,` `  InitPulseWidth=1,` `  InitPhase=0,` `  InitEnable=0` `}` `(` `  cfgInp: port in` `);` `end PwmGen1;`

Configurable Pulse Width Modulation (PWM) signal generator
Ports:

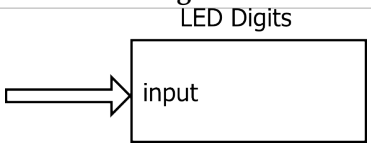
- `cfgInp`: configuration
  least significant bit is used to enable(high)/disable(low) the output; bits 1 and 2 are used for specifying a configuration command:
  - 0 - set period
  - 1 - set pulse width
  - 2 - set phase

  other bits specify the value to be set

Parameters:

- `MaxPeriod`: maximal period in clock cycles
- `InitPeriod`: initial value of the period in clock cycles
- `InitPulseWidth`: initial value of the pulse width in clock cycles
- `InitPhase`: initial phase (value of the time counter) in clock cycles
- `InitEnable`: non-zero if pulse generation is enabled at the start-up

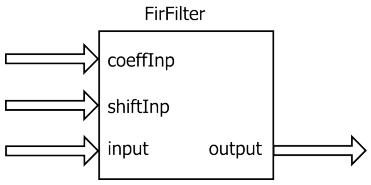
5.2.23 LEDDigits

Diagram	Interface
	`LEDDigits* = cell{Engine}` `(` `  input: port in` `);` `end LEDDigits;`

LED display driver for Spartan-3 Starter Board.
Ports:

- `input`: data input

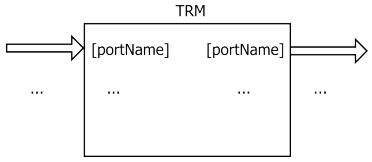
5.2.24 FirFilter

Diagram	Interface
	`FirFilter = cell{Engine,` `  CoeffWidth=16,` `  InpWidth=16,` `  OutWidth=32,` `  KernelLength=3,` `  PipeLength=6,` `  InitShift=0}` `(coeffInp: port in;` `  shiftInp: port in;` `  input: port in;` `  output: port out);` `end FirFilter;`

Digital FIR Filter
Ports:

- `coeffInp`: coefficients input
- `shiftInp`: shift value input
- `input`: data input
- `output`: data output

5.2.25 TRM

Diagram	Interface
	`[ProcessorInstanceName] = cell{Arch="TRM", CodeMemorySize=4096, DataMemorySize=4096 } ( [portName1]: port in; ... [portName2]: port out; ... ); var begin end [CPUName];`

TRM (Tiny Register Machine) soft-processor - a very simple RISC processor initially created by Niklaus Wirth. ActiveCells uses the second version of TRM processor called "TRM-2".

Parameters:

- `CodeMemorySize`: code memory size
- `DataMemorySize`: data memory size

5.3 Usage of Hardware Components

Hardware Components in *ActiveCells* current version are consisted of soft-processor (TRM) and dedicated component (*Engines*). As it was shown, TRM can be programmed in high-language Oberon, also the TRM program can configure *Engines* during runtime.

Among *Engines* there are components that have terminal ports that must be assigned to the FPGA I/O ports. For example, such a components are UartTx, UartRx, Gpio, etc. Other components like MulInt, Fifo, etc. don't have ports that must be assigned to FPGA I/O ports, these components are used in connection with TRM or with other components. Some *Engines* can be configured at the time of FPGA architecture synthesis, others can be configured in run-time.

5.3.1 Gpio Usage Example

```
cellnet GpioExample;
import
   Engines;
type
   Controller = cell{Arch="TRM",CodeMemorySize=1024,DataMemorySize=1024}
      (
         gpioCmdOut: port out;
         gpio: port out
      );
```

```
    var
        cfg, outVal, inpVal: set;
    begin
        cfg := {};
        gpioCmdOut ! cfg; (* configure All Gpio as inputs *)
        cfg := {21};
        gpioCmdOut ! cfg; (* send command to read all ports *)
        gpioInp ? inpVal; (* read all ports states *)
    end Controller;
var
    controller: Controller;
    gpio{DataWidth=10}: Engines.Gpio;
begin
    new(controller); new(gpio);
    connect(controller.gpioCmdOut,gpio.cmdInp,0);
    connect(controller.gpioOut,gpio.input,0);
    connect(gpio.output,controller.gpioInp,0);
end GpioExample.
```

5.4 FPGA Development Boards

Currently *ActiveCells* supports Xilinx FPGA's only. However *ActiveCells* supports any FPGA board with Xilinx FPGA chip on it.

To make quick start and simplify development at initial stage *ActiveCells* contains already designed configurations and demo projects for a several FPGA boards. These boards are shown in Table 9.

In *ActiveCells* each board has alias name (**ActiveCells Target Name** in Table 9) that should be specified in "`--target=`" parameter of command "`ActiveCellsComponents.BuildHardware`". For example, if we are implementing design for Avnet Spartan-6 FPGA LX75T board the command call should be as follow:

```
ActiveCellsComponents.BuildHardware --target="AVSP6LX75T"
--outputPath="HelloWorldAvnetSpartan6" HelloWorldAvnetSpartan6 ~
```

Also in this command call the third parameter "`HelloWorldAvnetSpartan6`" is a name of top cellnet (and name of file contains this cellnet). The second parameter "`--outputPath="HelloWorldAvnetSpartan6`" specifies the output directory where the design implementation will be stored.

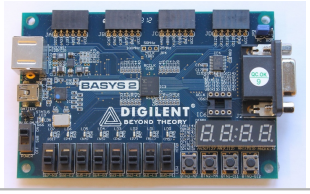
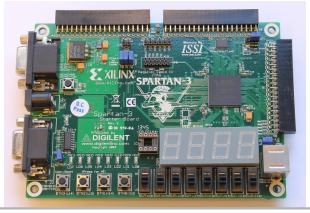
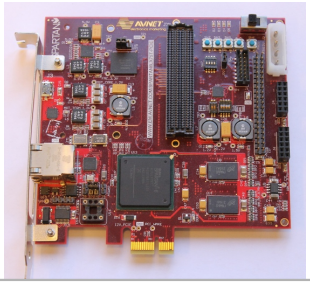
Board name	Board picture	ActiveCells Target Name
Digilent Basys2 Spartan-3E FPGA Board		Basys2Spartan3E
Digilent Spartan-3 Starter Board (with XC3S200 Spartan-3 FPGA)		Spartan_XC3S200
Avnet Spartan-6 FPGA LX75T Development Kit		AVSP6LX75T
ML505 FPGA Board (with Virtex-5)		ML505

Table 9: Supported FPGA boards

5.5 Extension of ActiveCells Hardware Library

ActiveCells provides flexible instruments for extension. *ActiveCells Hardware Library* can be expanded with custom hardware components. Any development board with Xilinx FPGA can be added into the *ActiveCells* .

Custom hardware components (*Engines*) should be written in Verilog or VHDL language. *Engines* HDL modules must have ports with AXI4-Stream interface in order to provide interconnections with other *ActiveCells* components. Every hardware component in the library is described by a corresponding *Hardware Component Specification* (HCS) file. Support of a new FPGA development board is provided by a corresponding

(*Target Specification*) (TS) file, which contains all necessary information about a specific FPGA board.

5.5.1 Hardware Component Specification File

HCS file is used to describe the mapping between the high-level interface of an *ActiveCells* component with the interface of the corresponding HDL hardware implementation. In the *ActiveCells* such mapping is described using XML. HCS file has the following structure:

All descriptions are contained within XML tags `<component> ... </component>`. The tag `<component ... >` has following attributes:

1. `type="HdlComponent"` tells that this is a description of an HDL component;
2. `name` defines the name of the hardware component, for example `name="UartTx"`;
3. `description` defines the description of the component, for example `description="UART␣transmitter"`;
4. `isa` defines the type of the Instruction Set Architecture (ISA), for example `isa="TRM "` is used in the case of the Tiny Register Machine (TRM) architecture; for engines `isa="none"`;

Within tags `<component> ... </component>` following tags are contained:

1. Tags `<supported> ... /<supported>` specify supported FPGA chips for this component. They contain tag `<element ... />` with the following attributes:
   (a) `type="StringValue"` specifies string attribute;
   (b) `value="*"` specifies supported FPGA chip, for example: `value="*XC6S*"` means that component is supported only by Spartan-6 FPGA.
2. Tags `<ports> ... /<ports>` contain component ports descriptions. Port is descried within tags `<element> ... </element>` of within tag `<element .. />`. Tag `<element .. />` has following attributes:
   (a) `type` specifies port type. Could have following values:
      i. `type="ClockHdlPort"` specifies clock port;
      ii. `type="HdlPort"` specifies hardware port ;
      iii. `type="AXI4StreamPort"` specifies AXI4-Stream port;
      iv. `type="TerminalHdlPort"` specifies GPIO hardware port;
   (b) `name` specifies port name in *ActiveCells* interface, for example: `name="input"`;
   (c) `mapped` specifies hardware definition port name, for example: `mapped="tdata"`;

(d) `optional` specifies whether the port must be connected somewhere: `optional ="false"` or not: `optional="true"`;

(e) `direction` specifies port direction. Could be `"in"`,`"out"` or `"inout"`, for example: `direction="in"`;

(f) `width` specifies port width, for example: `width="32"`;

(g) `signalPolarity` is specified only for reset port and sets reset polarity. Could be `"true"` or `"false"`.

For example, the description of component system reset port has following notation:

```
<element type="HdlPort" name="aresetn" mapped="systemReset" direction=
    "in" width="1" signalPolarity="false"/>
```

AXI4-Stream port is described within `<element> ... </element>` tags in following way:

```
<element type="AXI4StreamPort" name="input" description="" direction="in" width="32">
   <tdata name="inp_tdata" mapped="tdata" description="" direction="in" width="32"/>
   <tvalid name="inp_tvalid" mapped="tvalid" description="" direction="in" width="1"/>
   <tready name="inp_tready" mapped="tready" description="" direction="out" width="1"/>
</element>
```

Above code defines mapping of *ActiveCells* high-level port (defined by attribute `name` in `<element ... >` tag) to hardware AXI4-Stream ports (defined by attribute `name` in `<tdata .../>`, `<tvalid .../>`, `<tready .. >` tags).

System clock port is described within `<element> ... </element>` tags in following way:

```
<element type="ClockHdlPort" name="aclk" mapped="systemClock" direction="in" width="1">
   <clockFrequencyConstraints>
      <element type="IntegerRangeValue" value="1:"/>
   </clockFrequencyConstraints>
   <clockDutyCycleConstraints>
      <element type="RealValue" value="0.5"/>
   </clockDutyCycleConstraints>
</element>
```

For system clock the constraints could be specified by tags `<clockFrequencyConstraints >` and `<clockDutyCycleConstraints>`.

3. Tags `<parameters> ... /<parameters>` contain component parameters descriptions. Parameter is described within tags `<element> ... /<element>`. Tag `<element ... >` has following attributes:

   (a) `type="HdlParameter"` specifies that the hardware parameter is defined;

   (b) `name` specifies parameter name;

   (c) `mapped` specifies hardware name of parameter;

   (d) `description` specifies description of parameter.

   within tags `<element> ... /<element>` the parameter value is specified by tag `<value ...>` that has following attributes:

(a) `type` specifies type of attribute. For example: `type="RealValue"`, `type="IntegerValue"`;

(b) `value` specifies parameter value. The constrained could be added to parameter to specify permissible values, for example:

```
<constraints>
   <element type="IntegerRangeValue" value="0:1"/>
</constraints>
```

4. Tags `<dependencies> ... /<dependencies>` contain component dependencies descriptions. Each dependency is defined within `<element.../>` tag that has following attributes:

   (a) `type="HdlDependency"` specifies that the hardware dependency is defined;

   (b) `fileName` specifies name of hardware description file that the component is dependent on, for example: `fileName="UartTx.v"`.

Hardware Component Specification File has ".achc.xml" extension and name the same as hardware definition file.

5.5.2 Target Specification File

At the time of *ActiveCells Project* deployment into a target device (FPGA board), *ActiveCells Compiler* requires the information about device I/O pins. Compiler then maps input/output hardware ports of components used in design into hardware pins of FPGA chip. The TSF (*Target Specification File*) contains necessary descriptions in XML.

All descriptions are contained within tags `<device> ... </device>`. The tag `<device ... >` has following attributes:

1. `type="TargetDevice"` specifies type of device;
2. `name` defines name of target device (FPGA Development Board), for example: `name="ML505"`.

Device name is used by command `ActiveCellsComponents.BuildHardware` to specify target name, for example:

```
ActiveCellsComponents.BuildHardware --target="ML505".
```

Within `<device> ... </device>` following tags are contained:

1. Tag `<pldPart .../>` is contained descriptions of FPGA ship. It has following attributes:

   (a) `type` specifies Programmable Logic Device (PLD) part, should have value `type="PldPart"`;

(b) `vendor` specifies FPGA vendor. Currently `vendor="Xilinx"` is the only supported value;

(c) `family` specifies FPGA family. There is used an agreement to write name with capital letter following the number without space, for example: `"Spartan3"`, `"Spartan6"`, `"Virtex5"`, `"Virtex6"`, `"Kintex7"`. For Zynq-7000 the family name is `"Zynq"`;

(d) `device` specifies FPGA device name. Examples: `"XC3S200"`, `"XC5VLX50T"`, `"XC6SLX25"`, `"XC6SLX45"`, `"XC6VLX240T"`, `"XC7Z020"`;

(e) `package` specifies FPGA package. Examples: `"FTG256"`, `"CLG484"`, `"FF1136"`;

(f) `speedGrade` specifies FPGA speed grade. Example: `"-1"`, `"-2"`, `"-3"`;

(g) `cfgChainPos` specifies position of the PLD in the JTAG chain. For example, for Spartan XC3S200 development board `cfgChainPos="1"`.

2. Tags `<systemClock> ... </systemClock>` contain the description of the system clock. Tag `<systemClock>` has the following attributes:

(a) `type` defines the type of the clock. It can be either `"DerivedClock"` or `"ExternalClock"`. Attribute `type = "ExternalClock"` specifies that system clock is taken directly from external clock that feeds dedicated FPGA pin. Attribute `type = "DerivedClock"` specifies that system clock is derived from external clock with specified `mulRatio` and `divRatio` attributes. This is implemented by placing frequency synthesizer "behind the scene" in the project;

(b) `name` defines the clock name. For example `name = "systemClock"` ;

(c) `description` defines the description of clock. For example `description="system␣clock␣used␣to␣drive␣PLD"`;

(d) `frequency` (specified only for `type="ExternalClock"`) defines signal clock frequency. For example `frequency="200000000"`;

(e) `dutyCycle` (specified only for `type="ExternalClock"`) defines signal duty cycle. For example `dutyCycle="50"`.

(f) `mulRatio` (specified only for `type="DerivedClock"`) defines multiply ratio for frequency synthesizer. For example `mulRatio="4"`;

(g) `divRatio` (specified only for `type="DerivedClock"`)defines division ratio for frequency synthesizer. For example `divRatio="8"`.

Within `<systemClock ...> ... </systemClock>` tags input port should be defined to specify FPGA clock pin that drives system clock. For this tags `<inputPort ...> ... </inputPort>` are used that have following attributes:

(a) `type` specifies port type. Can be `"TerminalHdlPort"` or `"TerminalHdlPort"`;

(b) `name` defines the port name. For example `name="systemClockSrc"`;

(c) `direction` defines port direction. Should be `direction="in"`;

(d) `width` defined port width. For example `width="1"`.

Within `<inputPort ...> ... </inputPort>` tags pins should be specified, for example `<pins> <element name="systemClockSrc" loc="T9"/> </pins>` where `name` defines port name, `loc` defines name of FPGA pin.

When system clock is derived from external clock within system clock description tags `<systemClock type="DerivedClock" ... > ... </systemClock>` the external clock description should be specified like `<inputClock type="ExternalClock" ...> ... </inputClock>`

3. Tags `<systemReset> ... </systemReset>` contain the description of the system reset. Tag `<systemReset>` has the following attributes:

   (a) `type` ;

   (b) `name` ;

   (c) `description` ;

   (d) `signalPolarity` ;

   (e) `direction` ;

   (f) `width` .

   Within tags `<systemReset> ... </systemReset>` reset port pins are specified by tags `<pins> ... </pins>` that are contained tag `<element>` whith following attributes:

   (a) `name` ;

   (b) `loc` ;

   (c) `ioStandard` ;

   (d) `pullUp` ;

   (e) `pullDown` .

4. Within tags `<terminalPorts> ... </terminalPorts>` descriptions of terminal ports are contained. Each terminal port is described within `<element> ... </element>` tags. Tag `<element ... >` has following attributes:

   (a) `type` specifies port type, should be `type="TerminalHdlPort"`;

   (b) `name` specifies port name, for example `name="UartTxd0"`;

   (c) `description` specifies port description;

   (d) `direction` specifies port direction, could be `"in"`, `"out"`, `"inout"`;

   (e) `width` specifies port width, for example `width="8"`.

Within tags `<element> ... </element>` pins are specified by tags `<pins> ... </pins>` that are contained tag `<element>` with following attributes:

(a) `name` specifies name of port, for example `name="UartTxd0"`;

(b) `loc` specifies FPGA pin name, for example `loc="Y32"`;

(c) `ioStandard` specifies I/O standard, for example `ioStandard="LVCMOS33"`;

(d) `pullUp` specifies pull up, could has values `"false"`, `"true"`;

(e) `pullDown` specifies pull down, could has values `"false"`, `"true"`.

Target Specification File has "[Name].achc.xml" name, where [Name] is custom name of development board.

6 History of ActiveCells

1. 2007. The idea of FPGA-optimized CPU designed at Microsoft Research belongs to Chuck Thacker [**?**]. His CPU has 32-bit RISC architecture and targeted for educational purposes, however it can run real tasks and has a performance of 60 million instruction per second.

2. 2010. Second step made Niklaus Wirth at ETH Zurich by design 3 generations of Tiny Registers Mashines (TRMs) and high-level programming language (Oberon) compiler [1]. Also he build peripherals like UART, DMA and the hole system consisted from multiply cores (12) using single FPGA. The performance of Wirth system has introduced by "Supercomputer in a Pocket" project delivering the power for medical data processing. TRM-3 is a very minimalistic 32-bit RISC core occupied modest number of FPGA resources.

3. 2011. Next step was made by F. Friedrich, L. Liu and J. Gutknecht from ETH Zurich with contribution of Alexey Morozov and Patrick Hunziker from University Hospital of Basel and introducing Active Cells [2]. Active Cells allows to conveniently construct multi-core systems with different computer architectures, ranging from homogeneous many-core architectures to networks of heterogeneous general purpose processor cores or signal processing engines. They create FPGA hardware library implemented and a compiler provide a platform for prototyping and constructing distributed systems on a chip [2].

4. 2012-2014. Further contributions were made by Alexey Morozov, Felix Friedrich and Patrick Hunziker. They introduced a modular programming model and a suited development tool-chain that integrate the concepts of processors, component networks and customized task engines. Networks are defined and processors and engines are interconnected at a high level resulting in very short development, debugging and testing cycle times. This approach contributes to high-level development of complex systems on FPGAs that combine high-performance and robust functionality with a short time-to-deployment and, consequently, time-to-market.

Abbreviations

FPGA	Field-programmable gate array
ASIC	Application-specific integrated circuit
ISA	Instruction Set Architecture
RISC	Reduced instruction set computing
CISC	Complex instruction set computing
TRM	Tiny register machine
CPU	Central processing unit
PLD	Programmable logic device
I/O	Input/Output
RAM	Random-access memory
SRAM	Static random-access memory
UART	Universal asynchronous receiver/transmitter
LED	Light-emitting diode
GPIO	General-purpose input/output
VGA	Video Graphics Array
JTAG	Joint Test Action Group
USB	Universal Serial Bus
COM	Communication port
FIFO	First In First Out
FPU	Floating-point unit
AXI	Advanced eXtensible Interface
PWM	Pulse width modulator
DSP	Digital signal processing
FIR	Finite impulse response
IIR	Infinite impulse response
OS	Operating system
PC	Personal computer
XML	Extensible Markup Language
PET	Programmer's Editing Tool
TS	Target Specification
HCS	Hardware Component Specification
HDL	Hardware definition language

References

[1] N. Wirth *"Experiments in Computer System Design"* Technical Report, August. 2010 http://www.inf.ethz.ch/personal/wirth/FPGA-relatedWork/ComputerSystemDesign.pdf

[2] F. Friedrich, L. Liu, J. Gutknecht *"Active Cells - A Computing Model for Rapid Construction of On-Chip Multi-Core Systems"* http://nativesystems.inf.ethz.ch/pub/Main/FelixFriedrichPublications/ActiveCells.pdf

[3] F. Friedrich, J. Gutknecht, O. Morozov, and P. Hunziker *"A Mathematical Programming Language Extension for Multilinear Algebra"* http://computational.ch/publications/MathOberon2007.pdf

[4] O. Morozov *Spline- and tensor-based signal reconstruction: from structure analysis to high-performance algorithms to multiplatform implementations and medical applications*. Diss. PhD thesis, University of Basel, 2014

[5] H. Mossenbock. *Object-Oriented Programming in Oberon-2*. Second Edition. Springer-Verlag Berlin Heidelberg 1993, 1994 http://ssw.jku.at/Research/Books/Oberon2.pdf

[6] S. Smith. *Digital Signal Processing. A Practical Guide for Engineers and Scientists*

[7] U. Meyer-Baese *Digital Signal Processing with Field Programmable Gate Arrays*. Third Edition. Springer 2007.