Master Thesis

A Boot-Manager Core for a Processor Network on FPGA

Author(s):
Dillier, Andreas

Publication Date:
2016

Permanent Link:
https://doi.org/10.3929/ethz-a-010669154

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
A Boot-Manager Core for a Processor Network on FPGA

Master Thesis

Supervisors: Felix Friedrich, Alexey Morozov, Prof. Thomas Gross

Andreas Dillier
May 1, 2016
Abstract

The ActiveCells framework is a framework for simple processor network instantiation on FPGA. In ActiveCells, the program code of said processors currently has to be included into the hardware description files for the FPGA. The current tools available to change the code post-synthesis are not portable.

In this thesis, a portable solution is implemented, and the framework altered, such that the processors can be initialized independently to said FPGA configuration files.

The hardware description of the processors is altered in such a way to allow this post-configuration initialization.

The developed system shows reasonable performance and a much better performance than currently available tools.
## Contents

1 **Introduction**  
   1.1 ActiveCells  
   1.1.1 Engines  
   1.1.2 TRM  
   1.1.3 Cellnet  
   1.2 Problems to Tackle  

2 **Environment**  
   2.1 FPGA  
   2.2 Xilinx ISE  
   2.3 ActiveCells and A2  

3 **High-Level Design: Kernel or Hardware**  
   3.1 Kernel Approach  
   3.2 Hardware Approach  

4 **Hardware Changes to the TRM**  
   4.1 Data Memory  
   4.1.1 Initialization State  
   4.2 Instruction Memory  
   4.3 TRM Level Changes  
   4.3.1 Double Boot Port TRM  
   4.3.2 Single Boot Port TRM  
   4.3.3 No Separate Boot Port  

5 **Boot Core Design**  

6 **Data Stream Generation**  
   6.1 FPGA with Onboard ARM CPU  
   6.2 Patching from ROM  

7 **Performance**  
   7.1 Resource Consumption  
   7.2 Timing  

8 **User Guide**  
   8.1 TRM Definition  
   8.2 Cellnet Definition  
   8.3 Setting up the Boot Core and ARM Processor  
   8.4 Hardware Synthesis and Mapping on the Board  
   8.5 Creation of the Initialization Module  
   8.6 Initializing the Board  

9 **Conclusion**  

3
10 Further Work
10.1 TRM Reconfiguration .......................... 35
10.2 Better Tool-chain Integration .................. 35
10.3 Using Existing Topology ......................... 35
10.4 More External Interfaces ....................... 35

11 Appendix
A Full Example Module Codes ..................... 37
B Initial Task Description ......................... 43
C Declaration of Originality ....................... 46
List of Figures
1 The data flows through the TRM thanks to the kernel code . . . 11
2 Reserved sections for instruction and data memory . . . . . . 12
3 The three used states . . . . . . . . . . . . . . . . . . . . . . . . 16
4 The two memory modules each have separate Axi4 ports . . . 19
5 The example boot tree for larger hierarchies . . . . . . . . . . 19
6 Parallel initialization by different modules . . . . . . . . . . . . 20
7 The TRM has one port for boot data and internally splits the
  signal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
8 The boot core provides a single configuration interface . . . . 23
9 The ARM based interface . . . . . . . . . . . . . . . . . . . . . . . 26
10 The PROM based interface . . . . . . . . . . . . . . . . . . . . . . 27

List of Tables
1 Key features of the ZedBoard . . . . . . . . . . . . . . . . . . . . 9
2 Key features of the programmable logic . . . . . . . . . . . . . . 9
3 Space utilization of the TRM modules . . . . . . . . . . . . . . . 28
4 Space utilization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5 Time taken for initialization . . . . . . . . . . . . . . . . . . . . . . . . . 30

Listings
1 A simple TRM definition . . . . . . . . . . . . . . . . . . . . . . . . . 6
2 A simple cellnet definition . . . . . . . . . . . . . . . . . . . . . . . . 7
3 Pseudo code for the kernel . . . . . . . . . . . . . . . . . . . . . . . . . 11
4 The new memory module interface. DW denotes the data width,
  in this case 32 (bit). . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
5 The Xilinx friendly memory . . . . . . . . . . . . . . . . . . . . . . . 14
6 The instruction memory interface. DW denotes the data width,
  in this case 36 (bits). . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
7 The initialisation state machine. The two bit register instrNum
  denotes the current state . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
8 Updated TRM reset generation . . . . . . . . . . . . . . . . . . . . . 18
9 The simple signal demultiplexer . . . . . . . . . . . . . . . . . . . . . 21
10 Example boot core code for a cellnet with three TRM . . . . . . 24
11 Boot up data protocol . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
12 An example TRM definition . . . . . . . . . . . . . . . . . . . . . . . 31
13 An example cellnet definition . . . . . . . . . . . . . . . . . . . . . . 31
14 Controller and boot core definition . . . . . . . . . . . . . . . . . . 32
15 ExampleNetwork.Mdf . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
16 Zynq.Examplenetwork.Mod . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
1 Introduction

The goal of this thesis is to extend the ActiveCells framework[1] with a memory initialization method different and independent from the Xilinx tools provided, namely the Verilog readmem function[3] and the data2mem[6] functionality.

1.1 ActiveCells

The ActiveCells framework has been developed at ETH Zürich starting in 2010. The main idea of the ActiveCells computational model is, that a programmer can easily setup an FPGA (field programmable gate array) with a processor network in high-level code without having to bother with hardware description languages or FPGA configuration. ActiveCells features a broad hardware component library to simplify development.

1.1.1 Engines

Engines in the ActiveCells framework are pure hardware components, that can be used. Examples of engines are FIFO-queues, communication channels or on-board hardware drivers, such as LED controllers or switch inputs.

1.1.2 TRM

The processors used in the ActiveCells framework are so called "Tiny RISC machines", hereafter called TRM[2]. These TRMs use 18 bit instructions and have a separate data and instruction memory (i.e. they use Harvard architecture[4], which means the instruction memory is not writable through the processor itself). The programmer defines the code to be run on the processors which is then transferred to the FPGA with the Verilog readmem function in the memory modules.

```
Adder = CELL {Processor="TRM"} (in : PORT IN ; out : PORT OUT)

VAR i : LONGINT ;
BEGIN
  LOOP
    i <<= in ;
    i := i + 1 ;
    out <<= i ;
  END ;
END Adder ;
```

Listing 1: A simple TRM definition.

Listing 1 shows how a simple TRM can be programmed. The TRM continuously executes the code inside the outer loop. In this example, the TRM has one Axi4[5] in port and one Axi4 out port. The TRM keeps reading values from the in port, if available, increments the received value and sends it through the out port. Communication is blocking, so if there is no new value available, the TRM is blocked until a value is available. The same applies to the out port, if there is no receiver for the value, the TRM is blocked until there is one. The receiver depends on the connection type in the cellnet (next section) and can also be a FIFO bufer, in which case the TRM is only blocked if the buffer is at full capacity.
1.1.3 Cellnet

The programmer can define what components to use and how they are connected. This top-level topology is called a cellnet.

Components of a cellnet can either be engines or processors.

```
CN = CELLNET
VAR
c : Controller;
systemClock : Engines.SystemClock;
systemReset : Engines.SystemReset;
trm : Adder;
BEGIN
(* instantiate entities *)
NEW(c);
NEW(trm);
(* FIFO connection *)
CONNECT(c.out, trm.in, 256);
(* direct connection *)
CONNECT(trm.out, c.in);
(* clock and reset *)
NEW(systemClock);
CONNECT(c.clock0, systemClock.input);
NEW(systemReset{InputPolarity =0});
CONNECT(c.reset0, systemReset.input);
END CN;
```

Listing 2: A simple cellnet definition.

Listing 2 shows how a simple cellnet can be programmed using the before defined Adder TRM. The controller in this case actually is an ARM processor running outside the FPGA, however, it could just as well be another TRM or some kind of actuator with sensor that sends data and acts on the processed data.

Another noteworthy detail are the CONNECT statements. The connection from the controller to the TRM uses a FIFO buffer of length 256, denoted by the additional argument, the reverse direction is an unbuffered direct connection.

The framework takes these programmer definitions and automatically creates a corresponding Xilinx ISE project with all the necessary Verilog modules. Using the Xilinx ISE, the project can be mapped on an FPGA and then used.

1.2 Problems to Tackle

The current ActiveCells implementation has some problems regarding code and data initialization of the processors. For the `readmem` function, the code and data files have to be provided to the Xilinx ISE ahead of synthesis in order to include them in the generated FPGA programming file. This means, that even small changes in code lead to a relatively large computational and temporal overhead, because the cellnet has to be recompiled and the ISE project has to perform all steps again necessary to create a bitfile which can be fed to the FPGA.

One existing workaround is the `data2mem` functionality provided by Xilinx. This tool updates the memory sections written by the `readmem` function.
directly in the generated bitfile. However, this functionality is not available for other vendors and tends to scale relatively poorly.

In this master thesis, exactly this problem will be tackled. The thesis explores ways to reduce the dependency on vendor specific tools and implements a portable code and data initialization scheme that can be used.
2 Environment

In this section, a short overview of the used environment is given.

2.1 FPGA

The FPGA used, or rather, the SoC (system on chip) used, was a Zynq®
ZedBoard™.

Although this board was used, the approach implemented is of course not
limited to this approach, and can be used for any FPGA supported by Active-
Cells.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Dual-core ARM® Cortex™-A9 MPCore™ with CoreSight™</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>DDR3</td>
<td>512MB</td>
</tr>
<tr>
<td>Quad-SPI Flash</td>
<td>256 Mb</td>
</tr>
<tr>
<td>SD-card</td>
<td>4 GB</td>
</tr>
<tr>
<td>Programmable Logic</td>
<td></td>
</tr>
<tr>
<td>Logic Cells</td>
<td>85k</td>
</tr>
<tr>
<td>Block RAM</td>
<td>4.9Mb</td>
</tr>
</tbody>
</table>

Table 1: Key features of the ZedBoard.

<table>
<thead>
<tr>
<th>Slice Registers</th>
<th>106400</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice LUTs</td>
<td>53200</td>
</tr>
<tr>
<td>Slices total</td>
<td>13300</td>
</tr>
<tr>
<td>RAMB36E1/FIFO36E1</td>
<td>140</td>
</tr>
<tr>
<td>RAMB18E1/FIFO18E1</td>
<td>280</td>
</tr>
</tbody>
</table>

Table 2: Key features of the programmable logic.

2.2 Xilinx ISE

The Xilinx ISE design suite[7] contains all tools necessary to configure an FPGA
and map Verilog code onto the FPGA. The version used was 14.7.

The tools create hardware schematics from Verilog code, map the schematics
onto a specified hardware board and generate a configuration file that can be
used to configure the FPGA.

The tools are also used to connect on-board pins to the Verilog modules
(outside-world interface) and map the SPI flash memory.

2.3 ActiveCells and A2

Thanks to the used board being a SoC with integrated ARM processor, the
already developed ActiveCells extensions[10] can be used to facilitate commu-
ication between the ARM processor and the programmed logic on the FPGA.

1Full Specification: [9]
Additionally, the ARM processor can easily be programmed with high-level Oberon code thanks to the existing work.

With these tools, it is rather easy to develop code on a computer and interface to the board, since the operating system on the board provides a high-level interface. Code can be loaded onto the board in a few simple commands and then executed on the ARM processor. Since the ARM processor is also connected to the FPGA, the written Oberon code can directly communicate with the TRMs on the board.
3 High-Level Design: Kernel or Hardware

As a general design choice, two approaches were thinkable to allow memory configuration, an approach using some kernel code on the TRMs or an approach where the configuration happens on hardware level, invisible to the TRM.

3.1 Kernel Approach

In this approach, the TRM is instantiated with a short code section, called the kernel. The kernel approach reuses most existing hardware, with a little catch (Harvard architecture can not write instruction memory). The TRM starts executing this kernel code when booted.

The core of the kernel code repeatedly reads the incoming data from the Axi4 stream interface and stores the data in the appropriate place (Listing 3). The required kernel code section would not be too large and the kernel approach is very extensible, up to a point where an entire operating system could run on the TRM.

```
1 i := address_after_kernel_code;
2 while i <= instr_mem_size do
3   dat := receive();
4   instr_mem[i] := dat;
5   i := i + 1;
6 end
7 i := address_after_channel_addrs;
8 while i <= data_mem_size do
9   dat := receive();
10  data_mem[i] := dat;
11  i := i + 1;
12 end
```

Listing 3: Pseudo code for the kernel.

There are a few problems with the approach though. Firstly, the instruction memory can not be accessed in the regular Harvard architecture used in the TRM. This problem can be solved by introducing a special instruction to the TRM, which can only be used in kernel mode. This also means, a new kernel mode would have to be supported by the TRM.

Axi4 Input

```
TRM
IM
Kernel Code
DM
```

Figure 1: The data flows through the TRM thanks to the kernel code.
Secondly, as seen in the pseudo code of Listing 3, the TRM has to be aware of the kernel code, kernel code size and the channel addresses where the data is received, in order to properly write the instruction and data memory sections. This is not too bad, it just means the kernel code has to have the proper offsets included, which can be done during compilation and linking.

Additionally, the risk exists that the TRM accidentally overwrites crucial code and data sections, such as the kernel code itself. Special care has to be taken to prevent this from happening.

The most crucial flaw of this approach though is the fact, that the kernel code is replicated across all TRM nodes in the cellnet. The code and data memory sizes are already quite limited on the FPGA, and even though the kernel code could be done in a couple dozen instructions, the overhead over the whole cellnet adds up rapidly. Additionally, the TRM are not designed to be full-fledged systems with operating system code. The TRM are designed to be light-weight configurable processing nodes.

Since a hardware based approach does neither have to modify the TRM logic nor fill up the precious memory with clutter only used on boot-up, we decided to not implement the kernel approach.

Figure 2: Reserved sections for instruction and data memory
3.2 Hardware Approach

The hardware-based approach basically implements the basic algorithm described in the above kernel section. The difference is, that the reading and writing of the memory sections is done through additional hardware. On a high-level view, the memory is simply extended by another write interface that is used for initialization. The TRM cannot influence this process.

This is the approach we took and the next chapter discusses the modifications done in detail.
4 Hardware Changes to the TRM

In order to support the memory configuration at boot-up, the hardware had to be modified. The modifications mostly concern the memory modules, but small changes also had to be done to the TRM module.

4.1 Data Memory

The changes to the data memory encompass an additional Axi4 receive port and an initialization state.

4.1.1 Initialization State

The memory module starts in the initialization state and keeps the corresponding TRM reset signal on low (i.e. keeping the TRM in reset state). In this state, the module listens on the added Axi4 port for incoming word sized data chunks. If a data word is available, it is written to memory.

```
module DM
(
  //clock and reset signals
  input aclk,
  input arstn,
  //TRM write signals
  input [31:0] wrAdr,
  input [DW−1:0] wrDat,
  input wrEnb,
  //TRM read signals
  input [31:0] rdAdr,
  output reg [DW−1:0] rdDat,
  //TRM reset
  output resetTRM,
  // axi4 write interface
  input write_boot_tvalid,
  output write_boot_tready,
  input [DW−1:0] write_boot_tdata
);
```

Listing 4: The new memory module interface. DW denotes the data width, in this case 32 (bit).

There are many possible protocols one could use. The implemented data stream protocol is the following: incoming data is written to increasing addresses, starting from address 0, the first location in memory, and initialization is complete when the whole memory is written.

At the end of initialization, the Axi4 port is disabled and the TRM reset signal is set to high.

```
//concurrent write enable, address and data based on state
wire writeEnable;
assign writeEnable = TRM ? wrEnb : (write_boot_tvalid &
  write_boot_tready);
wire [31:0]writeAddr;
assign writeAddr = TRM ? wrAdr : write_address;
```

14
Listing 5: The Xilinx friendly memory.

The above version in Listing 5 shows how this protocol is implemented in hardware with two states for the memory, initialization (TRM register is 0) and TRM mode (TRM register is 1). Depending on the state, a different write interface is used. A previous version of the code with more explicit state division had the issue that the Xilinx ISE could not map the module properly. Instead of correctly using a dual port BRAM component available on the board, an expensive triple port memory was mapped in programmable logic. This is why in Listing 5, the two write inputs are manually assigned to a single write interface to the actual memory based on the state we are in.

4.2 Instruction Memory

The changes to instruction memory are largely the same as with data memory. The main difference arises from the fact, that instructions in the ActiveCells TRM implementation are 18 bits, and they are stored in memory in pairs, totaling a memory word size of 36 bits. The standard width of integers and thus the Axi4 port is 32 bit however. In order to solve this, the initialization state of the memory has its own three states.
Listing 6: The instruction memory interface. DW denotes the data width, in this case 36 (bits).

The three states work as follows (Figure 3):

1. The first instruction is received and written to a register.
2. The second instruction is received and written to another register, the Axi4 port is disabled.
3. The two received 32 bit values in the registers are truncated to 18 bit, concatenated to form the 36 bit word and written to memory. The Axi4 port is enabled again.

These three states alternate until the whole memory is written, just as with the data memory.

Figure 3: The three used states
//concatenate the registers
assign memLine = {inst2, inst1};

always @ (posedge aclk) begin
  //TRM mode
  if (TRM) begin
    //regular TRM code
    ... end
  
  //initial mode
  if (~TRM) begin
    //read instr 1
    if (instrNum == 2'b00) begin
      //data is ready
      if (write_boot_tvalid & write_boot_tready) begin
        inst1 <= write_boot_tdata[17:0];
        instrNum <= 2'b01;
      end
    end
    //read instr 2
    if (instrNum == 2'b01) begin
      if (write_boot_tvalid & write_boot_tready) begin
        inst2 <= write_boot_tdata[17:0];
        instrNum <= 2'b10;
        //disable Ax4 port
        writeReady <= 1'b0;
      end
    end
    //write line to memory and increase addr
    if (instrNum == 2'b10) begin
      mem[write_address] <= memLine;
      //switch mode if at end
      if (write_address == (Size -1)) begin
        TRM <= 1'b1;
        write_address <= 0;
        writeReady <= 1'b0;
      end else begin
        write_address <= write_address + 1;
        writeReady <= 1'b1;
        instrNum <= 2'b00;
      end end
    end
  end
end

Listing 7: The initialisation state machine. The two bit register instrNum denotes the current state.
The instruction memory did not have the problem of the Xilinx tools not being able to use BRAM for the actual memory block, because the TRM does not have a write interface to the memory. Therefore, the module has only one write and one read interface per default, which enables the tools to use the 2-I/O-port BRAM on the board automatically.

### 4.3 TRM Level Changes

Without modification of the TRM, the modules participating in initialization would have to know more about the memory systems than necessary and directly interface to them. In order to abstract these changes from these modules and provide a unified access scheme through the top-level TRM module, two new TRMs were designed.

The changes those designs have in common is a small addition to the reset mechanism to incorporate the signals received from the memories.

```vhdl
1 // new wire definitions to plug into the memories
2 wire rst;
3 wire dmrst;
4 wire imrst;
5
6 // the new reset signal 0 if any of the memories or external reset
7   are 0 (active high reset)
8 assign rst = dmrst & arst & imrst;
```

Listing 8: Updated TRM reset generation.

In Listing 8 we can see, that the default reset input is anded with the two memory reset signals. This means, whenever one of the reset signals is low, the whole TRM reset is low, which keeps the TRM in reset mode and stalled. Note that the TRM uses an active high reset mechanic, meaning the TRM has to receive a constant high signal to work. The reason is, in short, to avoid problems with power-on voltage fluctuations.
4.3.1 Double Boot Port TRM

In this variation, the TRM was simply extended with two additional Axi4 Ports and wired to the memories. This provides the outside world an interface to connect to either the data memory or the instruction memory separately.

![Diagram](image)

Figure 4: The two memory modules each have separate Axi4 ports.

This implementation is simple, but requires a lot of port connections for the module that provides the memory configuration data. Considering the current limitation of 16 in and 16 out ports on the TRMs, only 8 TRMs could be handled by a single boot core (see the boot core chapter). Having more than that would require a second boot core or a tree of boot cores for initialisation (Figure 5).

![Diagram](image)

Figure 5: The example boot tree for larger hierarchies.
This excessive use of connections also consumes more resources on the FPGA than necessary and should be avoided.

On the other hand, the data and instruction memory configuration could be done in parallel by different modules by simply interfacing to only one type of memory. If space is not an issue, but configuration time is highly critical, this double port TRM can be used.

![Parallel initialization by different modules.](image)

### 4.3.2 Single Boot Port TRM

In this variation, the TRM has only one Axi4 port for both memories. In order to make this design work, a simple additional module was developed to split the signal accordingly.

![The TRM has one port for boot data and internally splits the signal.](image)
The boot demultiplexer module prioritizes the data memory over the instruction memory. As long as the data memory requests more data, the received data is routed to the data memory module. As soon as the data memory switches its Axi4 ready signal to low, the data is routed to the instruction memory. Listing 9 shows the simple implementation for this demultiplexer.

```verilog
module BootDemux

    // single input Axi4 port
    input [31:0] boot_tdata,
    input boot_tvalid,
    output boot_tready,

    // data memory output port
    output [31:0] dat_tdata,
    input dat_tvalid,
    output dat_tready,

    // instruction memory output port
    output [31:0] inst_tdata,
    input inst_tvalid,
    output inst_tready

); //

wire [32:0] data;
wire valid;

// ready whenever client is ready
assign boot_tready = dat_tready | inst_tready;

// write to data first until not ready anymore
assign dat_tvalid = dat_tready & boot_tvalid;

// write im when data is not ready anymore
assign inst_tvalid = ~dat_tready) & boot_tvalid;

assign dat_tdata = boot_tdata;
assign inst_tdata = boot_tdata;
endmodule
```

Listing 9: The simple signal demultiplexer.

The order of data memory first, instruction memory second, is enforced by the fact, that the instruction memory implementation keeps switching its Axi4 ready signal due to the 3-state architecture used, where the ready flag of the memory switches during initialization. This would cause elements to end up in data memory during instruction memory initialization.

The TRM module itself connects the single Axi4 boot port to the demultiplexer module and the demultiplexer output signals to the memories.
4.3.3 No Separate Boot Port

Another TRM version with no separate Axi4 port for the boot signals could also be thought of, but was not further investigated. This approach would require a substantial changes to how the Axi4 ports on TRMs behave. One of the available input ports would have to be reserved for memory configuration.

Furthermore, there has to be a smart multiplexer or some sort of signal selection, in order to retrieve data on said input channel. This could be done using a demultiplexer inside the TRM, that is also wired to the reset signals of the memories, and selects data from the reserved channel, whenever the memories keep the TRM in reset. As soon as configuration is over, the memories do no longer send the reset signal and the multiplexer would switch to regular TRM operation.
5 Boot Core Design

In order to facilitate the boot process across all nodes in the topology, an additional node can be used, the so-called boot core. The boot core has to be aware of the topology on the FPGA and acts as a single interface for boot configuration of memories.

The current implementation is a specialization of a regular TRM (with fixed data and instruction memory, deployed in the traditional way).

The boot core is responsible for taking inputs from a single source and routing them to the other TRMs on the board that require configuration. The simplest implementation simply takes an Axi4 data stream and sends it to each connected TRM in a predefined order. The input could be any source, that is capable of sending an Axi4 data stream in the used boot configuration protocol (here the continuous stream of data entries followed by instructions for each TRM).

![Diagram of boot core and TRMs](image)

Figure 8: The boot core provides a single configuration interface.

More advanced schemes with TRM IDs and sending an ID to the boot core first, in order to decide the configuration order could easily be implemented due to the fact that the boot core is a fully functioning TRM and the code can be configured as with the other TRMs.

Although the boot core functions with the old technology, the data and instruction memory, once set up, is not expected to change, as long as the cellnet topology is not changed. And if the cellnet topology is changed, the synthesis and mapping in the ISE has to be redone anyways.
Listing 10: Example boot core code for a cellnet with three TRM.

Listing 10 shows a simple example of a boot core implementation. The example cellnet has three TRMs to be configured with equal memory sizes.

As already mentioned in the section "Double Boot Port TRM", a more complex topology of boot ports can be used. This applies to the case where more output ports are needed than supported (currently 16). The boot cores can be arranged in a tree to still provide a single interface to the outside world for configuration.

Using more than one boot core can also be considered for larger cellnets, in order to achieve a more localized mapping on the FPGA. Having a localized mapping can have benefits in overall FPGA resource consumption in spite of the bigger overhead for multiple boot cores.
6 Data Stream Generation

The setup with a boot core and all TRMs connected properly to said boot core provides an interface to initialize all memory modules of the TRMs on the FPGA with a simple 32-bit Axi4 data stream. The protocol used in this setup is the following:

```
foreach trm from 0 to numTRM (ordered)
    dm := data memory size;
    im := instruction memory size;
    for i from 0 to dm-1
        send 32 bit data item
    end
    for i from 0 to (2*im)-1
        send 18 bit instruction treated as 32 bit number
    end
end
```

Listing 11: Boot up data protocol.

The code and data that has to be sent in this protocol is readily available in .code and .data files after compilation.

There are many possibilities to generate such a stream from these files and send it to the boot core. In the following, two cases are discussed, the case where we have an operating system running on an external CPU and one case, where the data resides in flash memory.

6.1 FPGA with Onboard ARM CPU

On the SoC (System on Chip) of the Zynq chip, the board contains an additional CPU that can communicate with the FPGA. The ActiveCell framework is able to utilize this CPU and the provided Axi4 interface from the CPU to the FPGA. This facilitates communication from the developer machine to the board immensely. Thanks to this, the developer can write Oberon code that is then loaded on the CPU and executed. This code writing, compiling, linking and loading on the board can be done very quickly, compared to synthesizing the whole hardware over and over again.

This means, we can use this CPU to connect to the boot core and send the stream from the CPU. From the code and data files generated at compilation, a special initialization module is generated, that contains all necessary functions and orderings, such that a single function call can be done to send the whole stream at the start of operation (Figure 9).
6.2 Patching from ROM

If there is no CPU on the board and there is no possibility to run an operating system on the board, the data for the stream has to be placed somewhere else. Most FPGA have some kind of SPI[8] accessed flash memory or programmable read-only memory (PROM). This memory is most commonly used to store the FPGA layout generated by the Xilinx tools. However, it can also be used to store some user data.\(^2\)

In order to use this PROM to store and read the data stream, Verilog modules have to be written, which know how to read the PROM at the required locations through SPI. Then, the read data has to be translated into a valid 32-bit Axi4 word, and then simply sent on to the boot core. These driver modules have the disadvantage, that they can not be written fully portable, since different vendors each use their own implementations and specialties in hardware and the protocols to be used.

\(^2\)I want to thank Paul Reed for his help and insights to the PROM area. Without him, a lot of time would have gone to waste.
Unfortunately, the ZedBoard used does not allow direct access to the SPI PROM from the FPGA, and therefore this proof of concept could not be implemented for this thesis.
7 Performance

The main performance criteria for this work are how fast the new implementation is and how it compares to the existing Xilinx data2mem function and how high the price of the increased speed is in hardware.

7.1 Resource Consumption

Since the approach used features additional hardware components and also modifies existing hardware modules to be a bit more complex, the difference from the new system compared to the previous system is of interest.

Table 4 shows the key FPGA resources used for the traditional ActiveCells cellnet and a modified cellnet using a boot core and memory initialization. The topology used is a simple TRM ring, where the used TRM each have one in and one out port.

The exception is the eight-TRM complex layout. In this cellnet topology, each TRM is connected to each other TRM in a clique.

It can be seen, that the overhead in the ring scenario is relatively big. This is only logical, considering that half of the communication channels are used for the connection to the boot core.

In the example with the complex topology, we can see, that the overhead from the configuration setup is approximately constant from the simple topology using the same number of TRM. This shows, that the configuration overhead is largely independent from the used cellnet and the number of TRM in the cellnet to be configured is the key factor for the overhead.

<table>
<thead>
<tr>
<th></th>
<th>Traditional</th>
<th>New</th>
<th>BootCore</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>344</td>
<td>388</td>
<td>271</td>
</tr>
<tr>
<td>Slice Reg</td>
<td>180</td>
<td>247</td>
<td>177</td>
</tr>
<tr>
<td>LUT</td>
<td>681</td>
<td>741</td>
<td>681</td>
</tr>
<tr>
<td>LUTRAM</td>
<td>57</td>
<td>57</td>
<td>63</td>
</tr>
<tr>
<td>BRAM/FIFO</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3: Space utilization of the TRM modules.

Table 3 shows the module level utilization of a traditional, unmodified TRM, the developed single port configurable TRM and the boot core.
<table>
<thead>
<tr>
<th>TRM</th>
<th>Traditional (Abs.)</th>
<th>Traditional (%)</th>
<th>New (Abs.)</th>
<th>New (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 TRM</td>
<td>Slice Registers</td>
<td>275</td>
<td>1</td>
<td>577</td>
</tr>
<tr>
<td></td>
<td>Slice LUT</td>
<td>832</td>
<td>1</td>
<td>1575</td>
</tr>
<tr>
<td></td>
<td>Occupied Slices</td>
<td>423</td>
<td>3</td>
<td>686</td>
</tr>
<tr>
<td></td>
<td>LUT Flip-Flop</td>
<td>956</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAMB36E1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RAMB18E1</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>DSP48E1</td>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2 TRM</td>
<td>Slice Registers</td>
<td>468</td>
<td>1</td>
<td>868</td>
</tr>
<tr>
<td></td>
<td>Slice LUT</td>
<td>1547</td>
<td>2</td>
<td>2321</td>
</tr>
<tr>
<td></td>
<td>Occupied Slices</td>
<td>578</td>
<td>4</td>
<td>673</td>
</tr>
<tr>
<td></td>
<td>LUT Flip-Flop</td>
<td>1710</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAMB36E1</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RAMB18E1</td>
<td>3</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>DSP48E1</td>
<td>8</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>4 TRM</td>
<td>Slice Registers</td>
<td>860</td>
<td>1</td>
<td>1448</td>
</tr>
<tr>
<td></td>
<td>Slice LUT</td>
<td>2974</td>
<td>5</td>
<td>3860</td>
</tr>
<tr>
<td></td>
<td>Occupied Slices</td>
<td>1156</td>
<td>8</td>
<td>1603</td>
</tr>
<tr>
<td></td>
<td>LUT Flip-Flop</td>
<td>3236</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAMB36E1</td>
<td>8</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RAMB18E1</td>
<td>5</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>DSP48E1</td>
<td>16</td>
<td>7</td>
<td>20</td>
</tr>
<tr>
<td>8 TRM</td>
<td>Slice Registers</td>
<td>1804</td>
<td>1</td>
<td>2608</td>
</tr>
<tr>
<td></td>
<td>Slice LUT</td>
<td>5125</td>
<td>9</td>
<td>3860</td>
</tr>
<tr>
<td></td>
<td>Occupied Slices</td>
<td>2047</td>
<td>15</td>
<td>2363</td>
</tr>
<tr>
<td></td>
<td>LUT Flip-Flop</td>
<td>5630</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAMB36E1</td>
<td>16</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RAMB18E1</td>
<td>9</td>
<td>3</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>DSP48E1</td>
<td>32</td>
<td>14</td>
<td>36</td>
</tr>
<tr>
<td>8 TRM Complex Layout</td>
<td>Slice Registers</td>
<td>3008</td>
<td>2</td>
<td>3891</td>
</tr>
<tr>
<td></td>
<td>Slice LUT</td>
<td>8511</td>
<td>15</td>
<td>9735</td>
</tr>
<tr>
<td></td>
<td>Occupied Slices</td>
<td>3283</td>
<td>24</td>
<td>3920</td>
</tr>
<tr>
<td></td>
<td>LUT Flip-Flop</td>
<td>8950</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAMB36E1</td>
<td>16</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RAMB18E1</td>
<td>80</td>
<td>28</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>DSP48E1</td>
<td>32</td>
<td>14</td>
<td>36</td>
</tr>
<tr>
<td>16 TRM</td>
<td>Slice Registers</td>
<td>3532</td>
<td>3</td>
<td>4928</td>
</tr>
<tr>
<td></td>
<td>Slice LUT</td>
<td>10107</td>
<td>18</td>
<td>11342</td>
</tr>
<tr>
<td></td>
<td>Occupied Slices</td>
<td>4042</td>
<td>30</td>
<td>4456</td>
</tr>
<tr>
<td></td>
<td>LUT Flip-Flop</td>
<td>11218</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RAMB36E1</td>
<td>32</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RAMB18E1</td>
<td>17</td>
<td>6</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>DSP48E1</td>
<td>64</td>
<td>29</td>
<td>68</td>
</tr>
</tbody>
</table>

Table 4: Space utilization.
7.2 Timing

The other important factor for performance is how long initialization of the TRM takes after boot up and how this compares to existing technologies.

The existing technology that is available on all platforms is synthesizing a programming file for the FPGA. This method requires several minutes each time it is used. The method does not scale with the amount of change that was implemented.

If available, the Xilinx data2mem functionality already used in the Active-Cells framework is substantially faster than that and more importantly, scales with the number of changes performed. The method took between 550 and 600 milliseconds for each memory block that had to be changed.

Using a boot core and the single boot port TRM is dependent on the clock frequency used on the FPGA. In our case, a 83.333332 MHz frequency was used.

The number of cycles used in the data memory is expected to be one per memory write. The number of cycles expected in the instruction memory is three per instruction memory line written (that is, three cycles for two data items received). Furthermore, the boot core takes approximately 100 cycles in the innermost loop to receive a data item from the external source and send it to the TRM. Since these things happen pipelined, the cost of the boot core is expected to be the factor deciding the time needed.

Assuming a 512 line data and instruction memory, the total data items to be processed is 1536 for each TRM (512 lines for the data memory and 1024 single instructions for the instruction memory, later merged to double instruction words).

This yields an estimate of roughly 1.84 ms to configure one single TRM.

<table>
<thead>
<tr>
<th>With Boot Core (ms)</th>
<th>Without Boot Core (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 TRM</td>
<td>2</td>
</tr>
<tr>
<td>2 TRM</td>
<td>4</td>
</tr>
<tr>
<td>4 TRM</td>
<td>7</td>
</tr>
<tr>
<td>8 TRM</td>
<td>16</td>
</tr>
<tr>
<td>16 TRM</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 5: Time taken for initialization.

Table 5 shows the time measured for various numbers of TRMs. As can be seen, the expectation is met and initialization scales linearly with the number of TRMs used.

It can also be seen, that the boot core is indeed the largest portion of the time needed, since the tests performed without boot core and a direct connection from the ARM controller node to the TRMs are significantly faster.

Compared to the data2mem functionality, the time used on each TRM was cut from over one second (two memory modules at 550-600 milliseconds) to less than two milliseconds.
8 User Guide

In order to use the developed system, the following steps have to be performed.

8.1 TRM Definition

Just like in the traditional system, the TRMs to be used have to be defined and specified. This includes the number and types of ports to use and the code.

```
ExampleTRM = CELL { Processor="SingleTRM" } (in, boot: PORT IN; out: PORT OUT)
VAR i1: LONGINT;
BEGIN
  LOOP
    i1 := i1 + 1;
    out := i1;
  END;
END ExampleTRM;
```

Listing 12: An example TRM definition.

As can be seen in Listing 12, if the TRM is to be initialized using the new single boot port TRM, the processor type "SingleTRM" has to be used, and the special in port called "boot" has to be defined.

In order to use the dual-boot-port TRM, the processor type "DualTRM" has to be used. On this TRM, the special in ports called "bootData" and "bootInstr" have to be used to connect to the data or instruction memory respectively.

8.2 Cellnet Definition

The next step to probably be performed is to define the cellnet from said TRM definitions.

```
ExampleCN = CELLNET
VAR
  c: Controller;
  systemClock: Engines.SystemClock;
  systemReset: Engines.SystemReset;
  trm1: ExampleTRM;
  trm2: ExampleTRM;
  booter: BootTrmCell;
BEGIN
  (* instantiate entities *)
  NEW(c);
  NEW(trm1);
  NEW(trm2);
  NEW(booter);

  (* wire up the trm according to plan *)
  CONNECT(c.out, trm1.in, 256);
  CONNECT(trm1.out, trm2.in, 256);
  CONNECT(trm2.out, c.in, 256);

  (* wire up the boot core *)
  CONNECT(c.outboot, booter.in, 256);
  CONNECT(booter.out[0], trm1.boot, 256);
  CONNECT(booter.out[1], trm2.boot, 256);
```

31
The framework takes care of connecting the boot core outputs to the actual hardware port instead of creating a logical port. This is done because in the SingleTRM definition in the ActiveCells hardware library, the hardware port is called "boot" and the framework will automatically try to connect ports with the exact name to this port.

### 8.3 Setting up the Boot Core and ARM Processor

The two special nodes, the controller and the boot core, have to be specified as well. In the case of the controller, the interface definition has to be done in this section, as in the traditional system. The main point here is, that the controller has an out port connection to the boot core.

The boot core is simply the code from the boot core chapter adapted to this example.

```
Controller = CELL {Processor="ZynqPs7"}
(out, outboot: PORT OUT; in: PORT IN; clock0 {
  Frequency=100000000}: PORT OUT; reset0: PORT OUT)
BEGIN
  (* code has to be fed to ARM runtime *)
END Controller;

BootTRmCell = CELL {Processor="BootTRM"}(in: PORT IN; out: ARRAY 2 OF PORT OUT)
VAR data, addr, imsize, dmsize, totsize, trm, numtrm: LONGINT;
BEGIN
  LOOP
    (* loop through all data memory and write *)
    imsize := 512;
    dmsize := 512;
    totsize := imsize + imsize + dmsize;
    numtrm := 2;
    trm := 0;
    WHILE trm < numtrm DO
      addr := 0;
      WHILE addr < totsize DO
        data << in;
        out[trm] << data;
        addr := addr + 1;
      END;
      trm := trm + 1;
    END;
  END;
END BootTRmCell;
```

Listing 14: Controller and boot core definition.
8.4 Hardware Synthesis and Mapping on the Board

The next step is to compile the produced module. If all goes well, the next step is to create the Xilinx ISE project files.

A point to remember is, that the compiled boot core data and instruction files are by default not put into the Xilinx project folder. These files for the boot core should be renamed to BootTrmCell.code0 and BootTrmCell.data respectively, and copied to the now created Xilinx ISE project folder.

Now it is time to fire up the ISE itself and let the tools produce a bit file. The process properties of the implementation step have to be altered first. The "Other Ngdbuild Command Line Options" have to be deleted. Now, we can generate the programming file.

As before, the bit file can then be transformed into a bin file using the Xilinx tool promgen. This is the file we use later to initialize the FPGA hardware.

8.5 Creation of the Initialization Module

While the Xilinx tools work, the initialization module can be created. In order to do this, the OberonCodeConverter.exe was created. The tool will ask for the cellnet name and then some TRM names. The .code and .data files of the needed TRMs should be in the same folder. The order of the TRM names matters, since the one typed first is the first to be initialized. In the example above, trm1 should be entered before trm2, because the boot core starts sending to trm1.

After everything has been entered, the file {cellnetname}Init.Mod will be generated and is ready for inclusion in the project.

8.6 Initializing the Board

The module for the ARM processor has to include the generated module and call the initialization method.

At this point, the code can be loaded onto the ARM processor with the usual commands and the initialization can be done. From then on out, operation is as usual.

The full code of the example, including commands to be executed can be found in the appendix.
9 Conclusion

In this thesis, the ActiveCells framework was extended by specialized TRM in order to support initialization of instruction and data memory with a post-synthesis mechanism. This has been shown to be of value for the sake of quick development and error correction. The approach shown is portable across vendors, unlike some of the existing tools.

In order to achieve this, we have seen how the memory can be modified to support such a mechanism in hardware, and the possibility of a software based approach (kernel code) has been discussed.

To facilitate patching multiple machines, a boot core TRM has been proposed, that provides a clean interface to the implemented initialization mechanism.

Performance measurements show promising results when compared to the non-portable Xilinx data2mem function. However, the additional hardware cost can be substantial. Even worse is the fact, that most of this mapped hardware can currently not be further used after configuration.
10 Further Work

10.1 TRM Reconfiguration
A possible extension would be, to allow reprogramming a TRM at runtime. The steps required to make this work could be, in a simple case, a reset signal to the data and instruction memory of the corresponding TRM, in order to reset them into initialization state.

With this change, the memories would keep the TRM in reset mode again, and expect data values on their Axi4 ports.

10.2 Better Tool-chain Integration
The system developed in its current state is not yet quite in the state, where the programmer can write some code and get everything working with one button press. A lot of steps that are currently performed manually can be automated and integrated into the ActiveCells framework.

For example, automation could be achieved by creating the boot TRM code and data files with the correct name in the Xilinx ISE project folder directly, creating an initialization module automatically from code as a compiler flag or hiding the call to the initialization module from the user.

Even providing a simple standard boot core if no manual version is provided could be automated.

10.3 Using Existing Topology
Another idea could be to try to reduce the hardware cost of the initialization mechanism. An idea is to try to reuse regular cellnet TRM connections and route the data stream through the cellnet on existing channels, where possible.

10.4 More External Interfaces
The system developed relies on some form of external storage to retrieve the data and code files from. More driver modules can be developed for various boards and provided in a usable hardware library. Examples are the discussed PROM storage, SD card storage or USB connections.
References


11 Appendix

A Full Example Module Codes

```plaintext
MODULE ExampleNetwork;
IMPORT Engines;

TYPE
  BootTrmCell = CELL {Processor="BootTRM"}(in: PORT IN; out: ARRAY 3 OF PORT OUT)

VAR data, addr, imsize, dmsize, totalsize, trm, numtrm: LONGINT ;
BEGIN
  LOOP (* loop through all data memory and write *)
    imsize := 512;
    dmsize := 512;
    totalsize := imsize + imsize + dmsize;
    numtrm := 3;

    trm := 0;
    WHILE trm < numtrm DO
      addr := 0;
      WHILE addr < totalsize DO
        data << in;
        out[trm] << data;
        addr := addr + 1;
      END;
      trm := trm + 1;
    END;
  END;
END BootTrmCell;

AddingCell1 = CELL {Processor="SingleTRM"} (in, boot: PORT IN; out: PORT OUT)
VAR i1: LONGINT;
BEGIN
  LOOP
    i1 << in;
    i1 := i1 + 1;
    out << i1;
  END;
END AddingCell1;

AddingCell2 = CELL {Processor="SingleTRM"} (in, boot: PORT IN; out: PORT OUT)
VAR i1: LONGINT;
BEGIN
  LOOP
    i1 << in;
    i1 := i1 + 2;
    out << i1;
  END;
END AddingCell2;

AddingCell3 = CELL {Processor="SingleTRM"} (in1, in2, boot: PORT IN; out: PORT OUT)
VAR i1, i2, i3: LONGINT;
BEGIN
  LOOP
```

37
53 \quad i1 \ll i1;
54 \quad i2 \ll i2;
55 \quad i3 := i1 + i2;
56 \quad \text{out} \ll i3;
57 \text{END};
58 \text{END AddingCell3};
59
60 \text{Controller} = \text{CELL} \{ \text{Processor}="\text{ZynqPs7}" \}
61 \{ \text{out1, out2, outboot} : \text{PORT OUT}; \text{in} : \text{PORT IN};
62 \quad \text{clock0} \{ \text{Frequency}=100000000 \} : \text{PORT OUT}; \text{reset0} : \text{PORT OUT} \}
63 \text{BEGIN}
64 \quad (* \text{code has to be fed to ARM runtime} *)
65 \text{END Controller};
66
67 \text{ExampleCN} = \text{CELLNET}
68 \text{VAR}
69 \quad c : \text{Controller};
70 \quad \text{systemClock} : \text{Engines.SystemClock};
71 \quad \text{systemReset} : \text{Engines.SystemReset};
72 \quad \text{trm1} : \text{AddingCell1};
73 \quad \text{trm2} : \text{AddingCell2};
74 \quad \text{trm3} : \text{AddingCell3};
75 \quad \text{booter} : \text{BoostrapCell};
76 \text{BEGIN}
77 \quad (* \text{ instantiate entities } *)
78 \quad \text{NEW}(c);
79 \quad \text{NEW}(\text{trm1});
80 \quad \text{NEW}(\text{trm2});
81 \quad \text{NEW}(\text{trm3});
82 \quad \text{NEW}(\text{booter});
83 \quad (* \text{ regular trm to arm connection } *)
84 \quad \text{CONNECT}(c.\text{out1}, \text{trm1.in}, 1);
85 \quad \text{CONNECT}(c.\text{out2}, \text{trm2.in}, 1);
86 \quad \text{CONNECT}(\text{trm3.out}, c.\text{in}, 1);
87 \quad (* \text{TRM connections } *)
88 \quad \text{CONNECT}(\text{trm1.out}, \text{trm3.in1}, 1);
89 \quad \text{CONNECT}(\text{trm2.out}, \text{trm3.in2}, 1);
90 \quad (* \text{ arm to memory connection } *)
91 \quad \text{CONNECT}(c.\text{outboot}, \text{booter.in}, 1);
92 \quad \text{CONNECT}(\text{booter.out}[0], \text{trm1.boot}, 1);
93 \quad \text{CONNECT}(\text{booter.out}[1], \text{trm2.boot}, 1);
94 \quad \text{CONNECT}(\text{booter.out}[2], \text{trm3.boot}, 1);
95 \quad (* \text{ regular clock and reset } *)
96 \quad \text{NEW}(\text{systemClock});
97 \quad \text{CONNECT}(c.\text{clock0}, \text{systemClock.input});
98 \quad \text{NEW}(\text{systemReset[InputPolarity}=0]);
99 \quad \text{CONNECT}(c.\text{reset0}, \text{systemReset.input});
100 \text{END ExampleCN};
101 \text{END ExampleNetwork}.
102 \quad (* \text{ install ActiveCells3 hardware library } -- \text{ redo if changes applied to library itself, e.g. changes in TRM component } *)
103 \text{SystemTools.DoCommands}
104 \text{SystemTools.FreeDownTo AcHdlBackend }$
Listing 15: ExampleNetwork.Mdf

```
MODULE Test;

IMPORT
  SYSTEM, PsConfig, AcAxisIo, Kernel, Trace, Commands,
  ExampleCNInit;

VAR
  out1, out2, outboot: AcAxisIo.Output;
in: AcAxisIo.Input;

PROCEDURE ResetPl;
VAR t: LONGIN;
BEGIN
  ASSERT(PsConfig.SetPlResets([0, 1, 2, 3], res));
  t := Kernel.GetTicks();
  WHILE Kernel.GetTicks() - t < 1 DO END;
  ASSERT(PsConfig.SetPlResets([], res));
END ResetPl;

PROCEDURE InitMod;
VAR
  res: LONGIN;
  freq: HUGEINT;
BEGIN
  freq := PsConfig.GetPlClkFrequency(PsConfig.IoPll, res);
  Trace.String("IO PLL frequency is "); Trace.Int(freq, 0);
  Trace.StringLn(" Hz");
  Trace.String("Initial FPGA clock 0 frequency is "); Trace.
  Int(PsConfig.GetPcIclkFrequency(0, res), 0);
  Trace.StringLn(" Hz");
  ASSERT(PsConfig.SetPlResets([0, 1, 2, 3], res));
IF PsConfig.SetPlClock(0, PsConfig.IoPll, 12, 1, res) THEN
```

---

| AcHdlBackend. AddPathToLibrary "AC3HWL" |
| AcHdlBackend. LoadSpecs AC3HWL/Specifications.txt |
| AcXilinx.SetIseBinPath "c:/Xilinx/14.7/ISE_DS/ISE/bin/nt" |
| SystemTools.Show "ActiveCells library has been installed" |
| |
| (* Compile the code *) |
| SystemTools.DoCommands |
| Compiler.Compile -p=TRM |
| TRM/TRMRuntime.Mod |
| AxisChannels.Mod |
| AC3HWL/Engines.Mdf |
| Code/ExampleNetwork/ExampleNetwork.Mdf |
| (* Compilation of the code for hdl code generation *) |
| Compiler.Compile -p=Win32G —cellsAreObjects -e -i |
| AC3HWL/Engines.Mdf |
| Code/ExampleNetwork/ExampleNetwork.Mdf |
| |
| (* generate ISE project *) |
| SystemTools.DoCommands |
| SystemTools.Free ExampleNetwork |
| AcHdlBackend.Build —target="ZedBoard" -p="ISE" —outputPath=" |
| AC3P projectn" —g "ExampleNetwork.ExampleCN" |

---

| MODULE Test;
| IMPORT
|  SYSTEM, PsConfig, AcAxisIo, Kernel, Trace, Commands,
|  ExampleCNInit;
| VAR
|  out1, out2, outboot: AcAxisIo.Output;
in: AcAxisIo.Input;

PROCEDURE ResetPl;
VAR t: LONGIN;
BEGIN
  ASSERT(PsConfig.SetPlResets([0, 1, 2, 3], res));
  t := Kernel.GetTicks();
  WHILE Kernel.GetTicks() - t < 1 DO END;
  ASSERT(PsConfig.SetPlResets([], res));
END ResetPl;

PROCEDURE InitMod;
VAR
  res: LONGIN;
  freq: HUGEINT;
BEGIN
  freq := PsConfig.GetPlClkFrequency(PsConfig.IoPll, res);
  Trace.String("IO PLL frequency is "); Trace.Int(freq, 0);
  Trace.StringLn(" Hz");
  Trace.String("Initial FPGA clock 0 frequency is "); Trace.
  Int(PsConfig.GetPcIclkFrequency(0, res), 0);
  Trace.StringLn(" Hz");
  ASSERT(PsConfig.SetPlResets([0, 1, 2, 3], res));
IF PsConfig.SetPlClock(0, PsConfig.IoPll, 12, 1, res) THEN
```
Trace.String("FPGA clock 0 frequency has been changed to "); Trace.Int(PsConfig.GetPIMClockFrequency(0, res), 0); Trace.StringLn(" Hz"); ELSE Trace.String("Error while setting FPGA clock 0 frequency, res="); Trace.Int(res, 0); Trace.Ln;
END;

out1 := AcAxisIo.GetOutput(0, 0); (* trm1 *)
outboot := AcAxisIo.GetOutput(0, 2); (* boot *)
ton2 := AcAxisIo.GetOutput(0, 1); (* trm2 *)
in := AcAxisIo.GetInput(0, 0); (* trm3 *)

ResetPI; (* Reset programmin logic *)
END InitMod;

PROCEDURE InitTRM*(context : Commands.Context);
BEGIN
ExampleCNInit.InitTrm*(outboot);
END InitTRM;

PROCEDURE Test*(context : Commands.Context);
VAR x2, x1, y : LONGINT;
BEGIN
  ASSERT(context.arg.GetInteger(x1, FALSE));
  ASSERT(context.arg.GetInteger(x2, FALSE));
  Trace.String("x1="); Trace.Int(x1, 0); Trace.Ln;
  Trace.String("x2="); Trace.Int(x2, 0); Trace.Ln;
  WHILE ~out1.ready DO END;
  ASSERT(out1.ready);
  Trace.String("sending1"); Trace.Ln;
  out1 := x1;
  Trace.String("sent1"); Trace.Ln;
  WHILE ~out2.ready DO END;
  ASSERT(out2.ready);
  Trace.String("sending2"); Trace.Ln;
  out2 := x2;
  Trace.String("sent2"); Trace.Ln;
  WHILE ~in.available DO END;
  ASSERT(in.available);
  y := in;
  Trace.String("y="); Trace.Int(y, 0); Trace.Ln;
END Test;

PROCEDURE Test2*(context : Commands.Context);
VAR x2, x1, y : LONGINT;
BEGIN
  x1 := 0;
  x2 := 0;
WHILE x1 < 256 DO
  Trace.String("x1="); Trace.Int(x1,0); Trace.Ln;
  Trace.String("x2="); Trace.Int(x1,0); Trace.Ln;
  WHILE ~out1.ready DO END;
  Trace.String("sending1"); Trace.Ln;
  out1 := x1;
  Trace.String("sent1"); Trace.Ln;
  WHILE ~out2.ready DO END;
  Trace.String("sending2"); Trace.Ln;
  out2 := x1;
  Trace.String("sent2"); Trace.Ln;
  WHILE ~in.available DO END;
  y := in;
  Trace.String("y="); Trace.Int(y,0); Trace.Ln;
  x1 := x1 + 1;
END;
END Test2;

VAR
  i, t, res: LONGINT;
  baudrate, d: LONGINT;
  ch: CHAR;
BEGIN
  InitMod;
  Trace.StringLn("TestRing started!");
  END Test.

(* generate binary image file from the FPGA bitstream (.bit file
   generated in ISE) *)
 promgen -p bin -data_width 32 -b -u 0x0 examplecn.bit -w

(* ZYBO & ZedBoard *)
FSTools.CreateFile -c -r Replacements.tmp
BootConfig.UartInputClockHz = 50000000;
BootConfig.KernelOutputUart = 1;
BootConfig.CpuNb = 2;

(* Compile the ARM code with included ExampleCNI t Init.Mod *)
SystemTools.DoCommands

Release.Build -b -fARM.Release.Tool -o='--replacements=
  Replacements.tmp --notInitLocals --traceModule=Trace' --only="
  Kernel" ZyngA2
Release.Build -b -fARM.Release.Tool -o='--replacements=
  Replacements.tmp --traceModule=Trace' --only="System Shell"
ZyngA2

Compiler.Compile -b ARM --traceModule=Trace --objectFile=Generic --
  newObjectFile = replacements=Replacements.tmp
  ActiveCells/Zynq/Zynq.AcAxislo.Mod
  Zynq.PsConfig.Mod
  CodeConversion/ExampleCNIInit.Mod
  Code/ExampleNetwork/Zynq.ExampleNetwork.Mod

(* Link the ARM code *)
Listing 16: Zynq.Examplenet.work.Mod

```bash
StaticLinker.Link --fileName=Test.Bin --displacement=100000H -a
Runtime Initializer Platform FPE64 ARMRuntime Trace BootConfig Uart
    Machine Heaps Modules
Objects Kernel KernelLog Plugins Streams Pipes Commands Reals Clock
    Dates Strings Files Disks
Reflection TrapWriters Traps Locks Options Timer Shell
    ShellController

AcAxisIo
PsConfig
ExampleCNIInit
Test
    ~

(* start the server over which we interface to the board *)
TFTPServer.Start ~

(* scan for the port on which the board is *)
V24.Scan

(* connect to the board, the first argument depends on the V24.Scan
    result *)
WMV24Component.Open 3 115200 8 1 none ~

(*
Now a window should be opened if the board is connected properly
The following commands can be copied and pasted into this window
*)
( commands to be executed on the board to load the module *)
set source TFTP 10.3.34.150
load examplecn.bin fpga
load Test.bin memory 100000H
start 100000H

(*
After these commands were executed in the console, the modules are
loaded.
The following command will send all necessary code and data to the
    TRMs:
Test.InitTRM

After this step, you can use:
Test.Test <arg1> <arg2>
Test.Test2
*)
```
B Initial Task Description
A Boot-Manager Core for a Processor Network on FPGA

Master Thesis of

Andreas Dillier, 10-918-613, dilliera@student.ethz.ch

1.11.2015 - 1.5.2016

1. Assignment of Tasks

Goal of this thesis is to implement a boot-manager core on an FPGA that can configure processors on a peer-to-peer network on the FPGA.

In the Active Cells peer-to-peer model, each and every cell’s memory has to be configured during compile time by special tools provided by the FPGA vendors (Data2Mem in the case of Xilinx). The toolchain is thus dependent on the existence of data patching mechanisms and, moreover, deployment of code to the FPGA always requires a full reconfiguration of the FPGA. With this thesis this dependency will be removed with a boot manager engine that patches the data and instruction memory of all processors in the Active Cells network.

The following tasks should be finished in this Master Thesis project:

Mandatory Tasks

1. Implement a Boot-manager (BM) core based on a TRM communicating with an ARM on a Xilinx Zynq based ZED board. Communication interface with the ARM processor is AXI4 stream.
2. Define master and slave scenarios between the BM and the instruction stream providing interface. Minimally: ARM(Master)-BM(Slave), Flash(Slave)-BM(Master)
3. Implement communication for ARM-BM scenario.
4. Design and implement a memory configuration mechanism.
   - Start with the traditional AXI4-Stream port-port model (simplest approach). Modify the existing (Harvard Architecture) Processor TRM such that the memory of a TRM is exclusively written by the configuring engine.
   - Define a protocol between BM and TRM for uploading instruction and data memory code. (Halt TRM / Reset PC / Program Memory / Start).

Optional Extensions

1. Implement an SPI <-> BM interface for communicating with SPI (e.g. flash storage) devices or external adapters in order to run the same mechanism on an FPGA without an ARM interface.
2. Integrate the boot-manager mechanism into the Active Cells toolchain.
3. Explore different ways to access instruction memory on the TRMs, direct memory access from the BM versus TRM self-reconfiguration (access to instruction memory plus interface to BM)
4. Based on the results of task 4, explore different possibilities for interconnect topologies (bus / token ring?)
5. Partial Reconfiguration of the FPGA from a Boot-manager device.
A development environment based on A2 comprising the Active Cells Toolchain and build tools for the configuration of a Xilinx Zynq FPGA will serve as a starting point. The Communication between the ARM microprocessor and the Programming Logic based on AXI4 stream interfaces is already fully functional. A Xilinx Zynq ZED evaluation platform will be provided and Xilinx toolchain licenses will be made available as far as required.

2. Administrative Tasks

Provide a work schedule within the first two weeks of the project comprising potential risks and mitigation strategies.

Talk about the progress of the project regularly with the supervisors. However, the work itself and finding solutions to the problems regarding your tasks is the student’s responsibility.

The thesis itself should be written in English and must contain at least:

- The original task description
- Description and Analysis of the Problems
- Description, Motivation and Implementation of the solution
- Instruction Manual for a programmer / user
- Summary and Conclusion
- Outlook

Please hand in three bound copies of the thesis and make digital versions thereof available (Word, Latex or PDF). The program sources (source code und executables) must be provided in electronic form.

Professor: T. Gross

Supervision: Felix Friedrich, ETH Zürich, Alexey Morozov, HighDim GmbH, Basel.
C Declaration of Originality
Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

A Boot-Manager Core for a Processor Network on FPGA

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):
Dillier

First name(s):
Andreas

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Emmen, 01.05.2016

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.