Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze

Computer Engineering and
Networks Laboratory

Department of Information Technology and Electrical Engineering

Spring Term 2014

Resource-efficient Dynamic Partial Reconfiguration on FPGAs

Semester Project

Andreas Traber
atraber@student.ethz.ch

June 2014

Supervisors: Dr. Markus Happe, markus.happe@tik.ee.ethz.ch
Ariane Trammell, ariane.trammell@tik.ee.ethz.ch

Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

Abstract

The ReconOS project provides an environment for reconfigurable computing on an FPGA. Partial reconfiguration is used to change hardware accelerators on the FPGA during runtime. For this purpose fast reconfiguration speeds are important. During this semester project a new hardware core was developed that is much faster than the currently used solution and works independent of the CPU. The new core uses the ReconOS interfaces to communicate with the CPU and the main memory.

The capabilities of the Internal Configuration Access Port (ICAP) interface that is used for FPGA reconfiguration were examined and a proof-of-concept implementation of a state saving and restoring method for reconfigurable regions was developed. The proposed method only needs the bitstream that was used for partial reconfiguration. It is able to extract all information that is needed to capture the current state of a reconfigurable region from the bitstream.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ASIC	Application-Specific Integrated Circuit
CLB	Configurable Logic Block
CRC	Cyclic Redundancy Check
FAR	Frame Address Register
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FSL	Fast Simplex Link
FSM	Finite-State Machine
GSR	Global Set/Reset
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
LFSR	Linear Feedback Shift Register
LSB	Least Significant Bit
LUT	Lookup Table
RAM	Random Access Memory
RTL	Register-Transfer Level
SRAM	Static Random Access Memory

Chapter 1

Introduction

Modern Field-Programmable Gate Arrays (FPGAs) can contain many complex logic blocks, it is even possible to implement an entire computing system on an FPGA. The Virtex-6 FPGA from Xilinx that was used for this thesis supports dynamic reconfiguration during runtime, thus the computing environment can be adapted during execution so that it fits its current workload best. The reconfiguration port that is used on the FPGA is called Internal Configuration Access Port (ICAP). The ReconOS project provides such a computing system and uses a proprietary core to access the ICAP interface.

It is important that a reconfiguration is done as fast as possible because workloads can change any minute. The current solution used in ReconOS is rather slow and blocks the CPU of the system during reconfigurations. During this semester thesis a new core was developed that is faster and works independently of the CPU.

Additionally the capabilities of the ICAP interface were examined. The ICAP interface not only allows to reconfigure the FPGA on the fly, but one can also read configuration data from it.

This report is divided into several parts. In Chapter 2 background about FPGAs, ICAP and the ReconOS project is given. In Chapter 3 the hardware design that was chosen for the new ICAP controller is explained. In Chapter 4 the software design for the new core is shown. Chapter 5 contains the performance evaluation and resource utilization of the new ICAP controller. Chapter 6 draws a conclusion about this project and finally in Appendix A several issues are discussed that were encountered during the course of this project. In Appendix D a How-to can be found that explains the toolflow that was used for this project.

Chapter 2

Preliminaries / Background

2.1. FPGA Background

An FPGA is an Integrated Circuit (IC) that can be reconfigured by the customer after manufacturing. This allows a customer to implement any logic he desires in an FPGA as long as the FPGA is big enough to hold it. Due to the fact that an FPGA is reconfigurable it must be much bigger and more complex than an Application-Specific Integrated Circuit (ASIC) that implements the same logic. An FPGA is thus more expensive than an ASIC for very large quantities. FPGAs are thus mostly used for small volume production, prototyping and for applications where the ability to change the configuration after production is key.

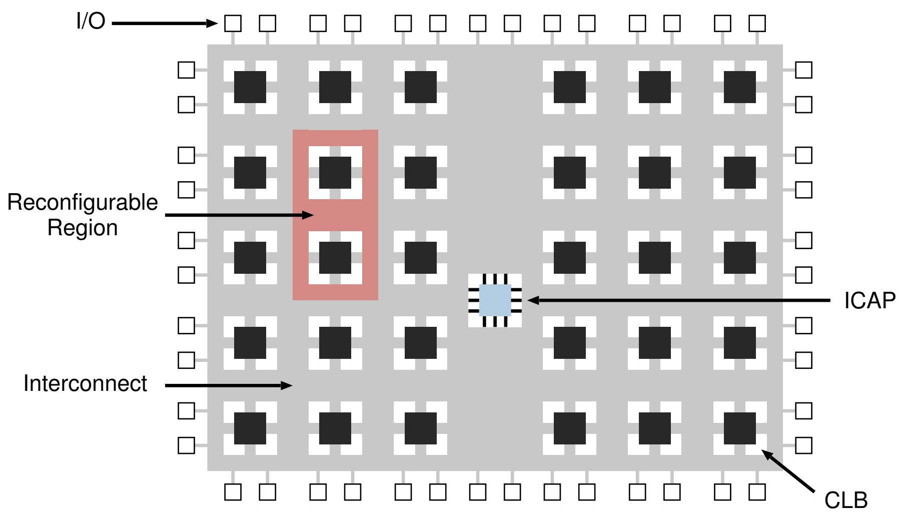


Figure 2.1.: FPGA Overview.

Figure 2.1 shows the general architecture of an FPGA. It contains thousands of so called Configurable Logic Blocks (CLBs) which are built of Lookup Tables (LUTs), flip-flops and multiplexers that allow for basic routing inside a CLB. Those CLBs are placed in regular intervals inside the FPGA and are connected with each other by interconnect logic that can be arbitrarily configured. The FPGA also contains I/O-pins which allows the logic inside to communicate with the outside world.

Most applications need some amount of Random Access Memory (RAM) for data storage which is why most FPGAs contain block RAMs. Block RAMs are typically small dual port RAMs that are placed at regular intervals inside the FPGA.

The FPGA that was used for this project was the XC6VLX240T from the Xilinx Virtex-6 Family and contains 18′840 CLBs and 416 block RAMs with a capacity of 36 Kb each [1].

Most modern FPGAs store their configuration data in Static Random Access Memory (SRAM) which makes it possible to implement a feature called partial reconfigurability. As the FPGA is based on SRAM it is possible to update only a portion of the SRAM with new content. This effectively allows us to reconfigure a region of the FPGA while the rest of it is still running.

2.1.1. Internal Configuration Access Port (ICAP)

Modern Xilinx FPGAs contain a hardware module that is called ICAP that allows a user design which is running inside the FPGA to access the configuration logic. The ICAP interface allows loading of new configuration data as well as a readback of the current configuration stored in the FPGA. Together with the partial reconfigurability feature it is thus possible to update the functionality of the currently running design at runtime from within the FPGA.

The ICAP can be configured to have an 8, 16 or 32 bit wide interface. The width of this interface directly influences the reconfiguration speed, as for an 8 bit wide interface we need 4 times as long to write new configuration data as for a 32 bit wide interface. The interface can be clocked with a frequency of up to 100 MHz which gives us a maximum throughput of 400 MB/s, though it has been shown that the interface can be overclocked to achieve even higher throughputs [2].

2.1.2. Bitstream Format

An FPGA is configured by a so called bitstream that is generated by the design tools of the FPGA vendor. The bitstream contains all configuration data necessary to run a design on an FPGA.

Partial reconfiguration generates bitstreams that only contain the configuration data that must be changed in comparison to a full bitstream. After loading a full bitstream first, one can load partial bitstreams that only affect the configuration of the specified reconfigurable region.

The bitstream is a sequence of commands that are interpreted by the configuration logic of the FPGA. A bitstream always starts with a synchronization sequence that tells the configuration logic the desired bus width (8, 16 or 32 bits) and contains a special synchronization word (`0xAA995566`). Similarly a bitstream always ends with a desynchronization sequence.

After the device has been synchronized, one can send configuration commands to it. Commands are sent in two steps. First a read or write request to a register is sent to ICAP. After this request the data that should be written to the register is sent or data is read from the register. Requests are sent in packets, each packet is one 32-bit word long and contains the configuration register plus the number of data words that should be read or written to this register. The most important registers and its uses are explained in Table 2.1. A more detailed explanation of bitstreams and the configuration logic is available in [3].

Table 2.1.: Configuration Registers.

Register	**Description**
FAR	Frame Address Register, contains the frame address that is used by FDRI and FDRO
CRC	Cyclic Redundancy Check Register, is used for integrity checks of the bit-stream
CMD	Command Register, commands are executed after writing to this register
FDRI	Frame Data Input Register, configuration data is written to it
FDRO	Frame Data Output Register, configuration data is read from it

Configuration data that controls the state of LUTs, flip-flops and block RAM is written in the form of configuration frames where each frame has a length of 81× 32-bit. The frames are written to the FDRI register after the address of the frame has been set in the Frame Address Register (FAR). Figure 2.2 shows the structure of such a bitstream. The frame address is automatically incremented when multiple frames are written with one write request. A full bitstream usually contains only one write command after which the configuration for the whole FPGA follows.

2.2. ReconOS Project

The ReconOS project provides an operating system for reconfigurable computing [4]. It runs on modern FPGAs such as the Virtex-6 that is used in this thesis. Reconfigurable

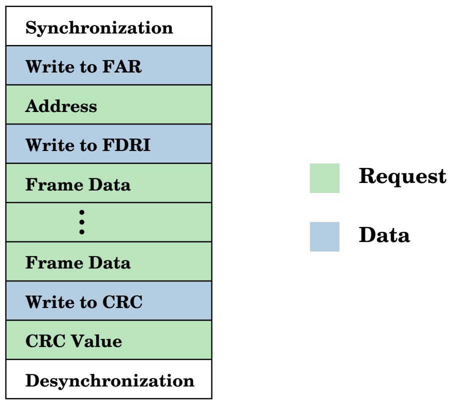


Figure 2.2.: Bitstream Structure.

computing in this content means that it uses partial reconfiguration to dynamically configure hardware accelerators on the FPGA. ReconOS calls those accelerators hardware threads as they have similar capabilities to threads in the software environment. This system also contains a traditional CPU which manages the whole system. As the Virtex-6 FPGA does not contain a dedicated CPU, ReconOS uses a softcore MicroBlaze CPU from Xilinx. This CPU runs a modified version of Linux that includes support for the interfaces used by hardware threads.

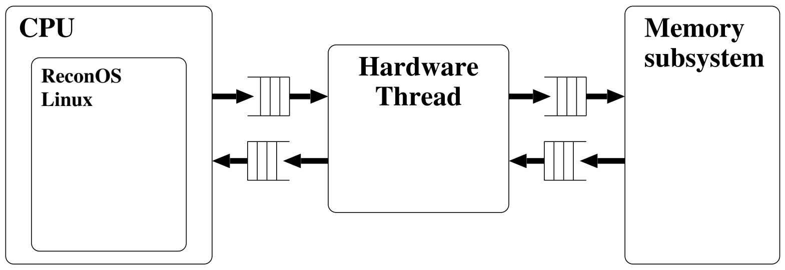


Figure 2.3.: ReconOS Hardware Thread Interfaces.

Every hardware thread has an operating system interface which is used to communicate with the CPU and a memory interface which is used to communicate directly with the main memory, see also Figure 2.3. These interfaces are basically First In First Out (FIFO) queues running in both directions. It is possible to disable the FIFO queues to prevent erroneous data to be inserted into them during reconfigurations.

Until now ReconOS used the XPS_HWICAP core to perform reconfigurations which has severe drawbacks that will be explained below.

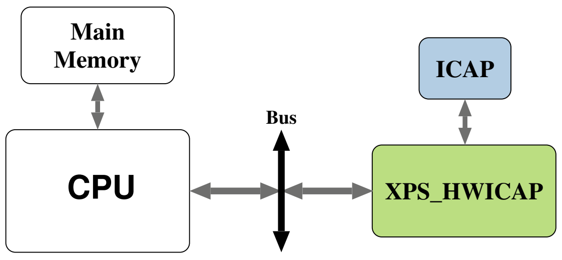


Figure 2.4.: XPS_HWICAP Core.

2.3. Xilinx ICAP Controller: XPS_HWICAP

Xilinx provides an implementation of a core that is able to use the ICAP interface which is called XPS_HWICAP. This core is intended to be used together with the MicroBlaze CPU and uses a system bus to communicate with the CPU, see Figure 2.4. In order to do a partial reconfiguration, the CPU must first fetch the bitstream from the filesystem and load it into main memory, then it has to write word after word of the bitstream to the bus. This procedure is very inefficient and slow, and effectively blocks the CPU during reconfigurations.

It has been shown by Delorme et al. [5] and Liu et al. [6] that using direct memory access instead of burdening the CPU with the transfer achieves much lower reconfiguration times and also offloads the CPU. The goal for this semester thesis was thus to develop a hardware thread for the ReconOS system that does reconfigurations without the assistance of the CPU.

2.4. Related Work

Related work has shown that the maximum transfer speed of the ICAP interface given by Xilinx can be exceeded by overclocking the ICAP interface [2]. Overclocking is highly device specific as not all devices can support the same speed and it makes routing more complicated which is why overclocking was not considered as an option for this semester thesis.

It has been shown by Liu et al. [6] and Bonamy et al. [7] that bitstream compression can be used to decrease the size of the data that must be transferred to the ICAP controller. Compression ratios of up to 75% have been reported. Compression algorithms are fairly complex and take up a huge amount of space on the FPGA. Due to the limited time scope of a semester thesis it was not possible to include bitstream compression for the new ICAP core.

Liu et al. [8] have used block RAMs to cache bitstreams on the ICAP controller prior to writing them to the ICAP interface. This avoids a potential memory interface bottle neck as the partial reconfiguration bitstream does not have to be transferred to the ICAP core first. For the new ICAP core bitstream caching was not done as this wastes a huge amount of block RAMs. Also this would only be possible for applications where it is known in advance which bitstream is needed next, so that it is possible to put it into the cache.

It was demonstrated by [9, 10] that state capturing and restoring is possible by reading back configuration data over ICAP. Jozwik et al. [9] have used a Virtex-4 FPGA to do state capturing and restoring, however they needed additional combinational logic inside the reconfigurable regions to be able to restore the state of flip-flops in reconfigurable regions.

Morales-Villanueva and Gordon-Ross [10] have shown that state capturing and restoring is also possible on a Virtex-5, but they did not include the restoration of block RAMs in their work. They have also demonstrated that it is possible to relocate reconfigurable regions, i.e. restore the state from the reconfigurable region 1 in region 2.

Chapter 3

Hardware Design

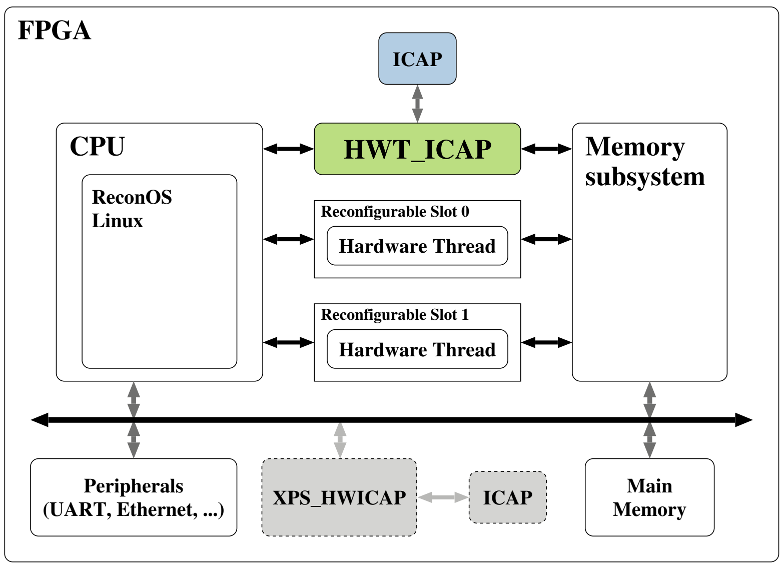


Figure 3.1.: HWT_ICAP Hardware Thread.

To solve the problems that the XPS_HWICAP core has, I have implemented a new core that uses the interfaces of the ReconOS project to communicate with the operating system and the main memory. Figure 3.1 shows the integration of the new HWT_ICAP core into the ReconOS system. In contrast to the XPS_HWICAP core, this new core has direct access to main memory which allows it to work independently of the CPU.

3.1. Partial FPGA Reconfiguration

With the new HWT_ICAP core only the code in Algorithm 3.1 needs to be run on the CPU to perform a partial reconfiguration:

Algorithm 3.1: Performing a Reconfiguration using HWT_ICAP

```
Input  : Address and size of the bitstream in main memory
Output: Success / Error
1 // send address of bitfile in main memory to HWT_ICAP
2 mbox_put (mbox_out, address);
3 // send length of bitfile (in bytes) in main memory to HWT_ICAP
4 mbox_put (mbox_out, size);
5 // wait for response from HWT_ICAP
6 ret = mbox_get (mbox_in);
```

3.1.1. Single Buffering

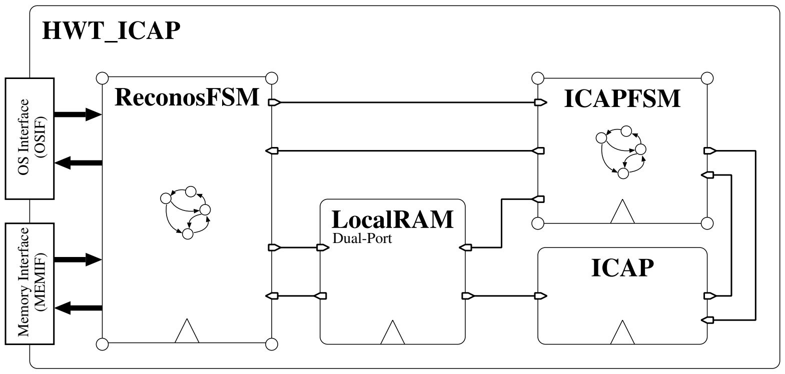


Figure 3.2.: HWT_ICAP Block Diagram.

Figure 3.2 shows the block diagram of HWT_ICAP. The ICAP interface is connected to a dual port RAM which is controlled by a separate Finite-State Machine (FSM) that manages the transfer of the bitstream from the local RAM to the ICAP interface. This RAM is filled by a second FSM that manages communication with the operating system and the main memory. As already mentioned, the ICAP interface supports 8, 16 and 32 bit wide data. Since the ReconOS memory interface is 32 bit wide, the 32 bit wide ICAP interface has been used in this core.

The local RAM is not large enough to hold a complete partial bitstream, thus the RAM is filled by the second FSM and its content is then sent to the ICAP interface, then the RAM is filled again and so on, until the complete bitstream has been transferred to the ICAP interface.

This approach is simple and works well, but its efficiency can be improved, as either data is transferred from main memory to the local RAM, or data is transferred from the local RAM to ICAP, but it is not possible to perform both transfers at the same time.

3.1.2. Double Buffering

To make the transfer of a bitstream from main memory to ICAP more efficient, we can transfer data from main memory to the local RAM and data from the local RAM to ICAP at the same time.

This can be done by using double buffering. In this scheme the FSMs always work on only half of the local RAM. The ICAP FSM works on the lower half of the memory, while the other FSM works on the upper half and vice-versa when both have completed their transfers. There are no unnecessary idle times anymore, both FSMs are active whenever possible.

3.1.3. Without RAMs

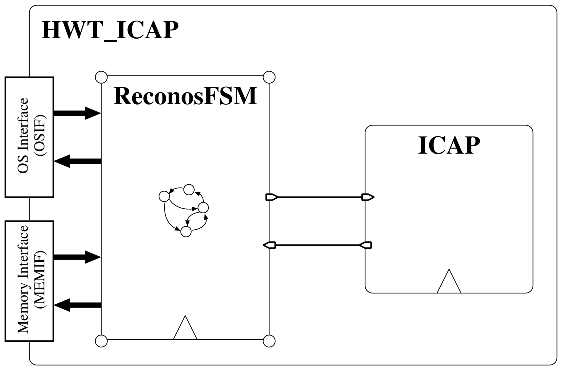


Figure 3.3.: HWT_ICAP Block Diagram without local RAM.

To write to the ICAP interface one only needs three signals: Clock, chip enable and the input data. Chip enable and the data have to be supplied synchronous to the clock signal, thus the chip enable signal can be set when there is valid data to write to the ICAP interface. This interface is similar to a RAM, except that there is no address input. So if it is possible to read from main memory in a sequential way, so that data is read from incremental addresses, then the local RAM can be bypassed entirely, the signals that would go to the local RAM are instead connected directly to the ICAP interface, see Figure 3.3. With this scheme, the entire bitstream can be transferred in one go, making it the most efficient way to transfer data from main memory to ICAP using the ReconOS memory interface. It is the most efficient way because the ReconOS memory interface is always busy and thus the memory bandwidth is used in an optimal way.

3.1.4. Failure Handling

Performing a reconfiguration on an FPGA while it is running is a critical task as any error in the configuration data might upset the whole FPGA and lets the system crash. To mitigate those errors, bitstreams for Xilinx FPGAs contain Cyclic Redundancy Check (CRC) checks along with the configuration data. The configuration logic inside the FPGA continuously calculates the CRC value of the configuration data and checks it against the values stored in the bitstream. If an error is detected, an error flag is raised by the ICAP interface, making it possible for a user to handle this exception and load a different bitstream.

The HWT_ICAP core reacts immediately to this error flag and stops writing data to the ICAP interface. It then resets the CRC register and informs the operating system about the error condition. This procedure ensures that the ICAP interface is always in a valid state even when errors occur.

According to Xilinx CRC errors are unlikely if the bitstream is stored locally [11]. A more complex recovery strategy than just loading a different bitstream is thus probably not needed.

3.2. FPGA Configuration Readback

The ICAP interface not only allows to write new configuration data to the FPGA, it also allows reading back the currently active configuration data. A readback of configuration data can for example be used to check for flipped bits in the configuration. Flipped bits in the configuration memory are caused by noise, e.g. radiation. If a flipped bit is detected during a readback, the bitstream used for programming could be downloaded a second time and thus the flipped bit can be corrected.

Reading from the ICAP interface is not as easy as writing to it, details of specific issues with this interface can be found in Section A.2. Reading must be done in two steps and needs a local RAM on the ICAP core. Configuration data is read from the ICAP interface, stored in the local RAM and then transferred to main memory using the ReconOS interface. As the local RAM is much smaller as a typical configuration frame, this process is repeated until the specified number of words is transferred.

Algorithm 3.2 shows how data can be read from the ICAP interface using the HWT_ICAP core. The Least Significant Bits (LSBs) of the buffer address and buffer size are always zero, as buffers must be word-aligned in order for the ReconOS memory interface to work correctly. As those bits are always zero, we can use those two bits to encode additional information in the messages. If the LSB of the second message is `0`, data is written to ICAP. If the LSB is `1`, data is read from ICAP. Although reading alone is simple, it will not work without first telling the ICAP interface what to read. See Section 4.2.2 for details how this can be done.

Algorithm 3.2: Reading using HWT_ICAP

```
Input   : Address and size of the readback buffer in main memory
// send address of buffer in main memory to HWT_ICAP
mbox_put (mbox_out, address);
// send length of buffer (in bytes) in main memory to HWT_ICAP
mbox_put (mbox_out, size | 0x1);
// wait for response from HWT_ICAP
ret = mbox_get (mbox_in);
```

Note that we can not only read the configuration data back from the device, but also the current state [3, 10, 9]. This allows us to capture the current state of a reconfigurable partition, store it in main memory for some time and restore it later. With this feature we can do preemptive hardware scheduling similar to preemptive scheduling on CPUs. For example we can start a time consuming calculation on a hardware thread. If we need a calculation from a different hardware thread right now, we can capture the first thread and replace it with the new one. This thread can then do its job. After it is done, we can restore the previous thread which can continue its calculation without even knowing, that it was stopped in between. Another example is two periodic threads that need to run once every second but only need a short time for their calculations. By continuously swapping between those two threads, both can do their jobs. Without preemptive hardware scheduling both threads would need their own dedicated area on the FPGA. This area is wasted when hardware threads are idle, so hardware scheduling is beneficial.

3.3. Restore Registers

To restore a hardware thread to a previously captured state, one needs to write the captured configuration data to the device and then perform a Global Set/Reset (GSR). GSR is a special signal that runs through the whole FPGA and resets every flip-flop in a given region to its initial / captured state. More details on how this works are given in Section 4.2.4.

According to Xilinx [3] there are two ways to perform a GSR. The first is to send a GRESTORE command over the ICAP interface, the second is to toggle the GSR signal manually on the STARTUP_VIRTEX6 primitive. It turned out that sending a GRESTORE command over ICAP does not work, see Section A.3 for more details about this issue. So in order to be able to perform a GSR, the new core includes the capability to toggle the GSR signal manually on the STARTUP_VIRTEX6 primitive.

Algorithm 3.3 shows what needs to be sent to the thread such that the HWT_ICAP core performs a GSR. Once again we are using the LSB to encode information. If the

LSBs of the first message is ’1’, a GSR is performed.

Algorithm 3.3: Performing a GSR using HWT_ICAP

```
// send GSR message to HWT_ICAP
mbox_put (mbox_out, 0x1);
// wait for response from HWT_ICAP
ret = mbox_get (mbox_in);
```

Chapter 4

Software Design

4.1. Partial FPGA Reconfiguration

The HWT_ICAP core is controlled by software running on the MicroBlaze CPU inside the FPGA, thus there is also software required to perform reconfigurations.

The partial bitstreams are typically stored on a large storage device, in our case a compact flash card that is plugged into the FPGA development board. Since the HWT_ICAP core can only access main memory, the bitstream must first be loaded into the main memory by the CPU. It is then checked in software if the bitstream looks valid, i.e. this means searching for the synchronization sequence. Every valid bitstream contains it in the first few words, if a file does not contain this sequence, it is either not a bitstream or it is in the wrong format.

The Xilinx `bitgen` tool can generate bitstream files in multiple formats. Reconfiguration using ICAP needs files with the `.bin` suffix, while configuration using JTAG needs files with the `.bit` suffix [12]. JTAG configuration files contain a proprietary header that must be stripped first if they should be used for reconfiguration with ICAP.

When a reconfiguration should be performed, the following steps have to be executed:

1. Activate the reset signal leading to the reconfigurable region.
   This also resets the operating system and memory interfaces.
2. Instruct the HWT_ICAP core to write the partial bitstream to ICAP.
3. Deactivate the reset signal.
   The reconfigured region is now in its initial state due to the reset signal and all interfaces are cleared and ready for this new core.

The reason why the reset signal has to be set before the reconfiguration is started is that otherwise erroneous data could be written to the interfaces leading to the operating system or to the memory during the reconfiguration step.

4.2. FPGA Configuration Readback

4.2.1. Bitstream Parsing

To perform a readback of the active configuration data on the FPGA, one needs to know which frames should be read. This information is not that easy to get, as it depends on the reconfigurable region being used. The best way I found to get this data was to parse the bitstream used for programming and extract this information from it. Specifically the frame addresses and the number of frames that were written in every write request need to be extracted, see Figure 4.1.

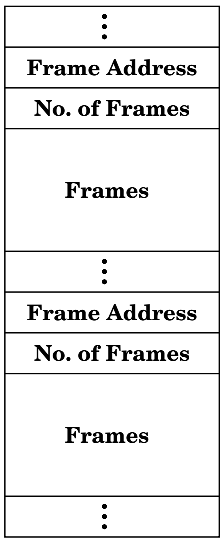


Figure 4.1.: Configuration Bitstream.

By reading back exactly the same frames as were previously written to the device, one can then compare those frames and see if there are any differences. If there are, then those hint to a potential error in the FPGA configuration, but this does not necessarily have to be the case. Some bits in the configuration frames correspond to user memory or null memory that should not be compared against the original bitstream. The `bitgen` tool can generate mask files that contain the bit locations that should/should not be compared [3].

4.2.2. Reading from ICAP

Reading a specific configuration frame from the ICAP interface is quite complicated and needs several steps:

1. Synchronize the ICAP interface (if it is not already synchronized)
2. Set the FAR to the frame that should be read
3. Send a read configuration packet for the FDRO register to ICAP.
   Also the number of words that should be read back must be specified, set this number to the number of words you want to read plus 82 words.
4. Read from ICAP the number of words you specified above
5. Desynchronize the ICAP interface (unless it is needed soon)

The reason why one needs to read 82 more words is that the ICAP interface outputs a dummy frame first to flush its internal frame buffer, so the first 82 words have to be discarded. It is also important to read exactly the number of words that were announced to ICAP as otherwise the ICAP interface is in a unknown state and will not accept new configuration commands.

4.2.3. Capture Current State

To be able to read the current state from the FPGA, a few additional steps are required. The Virtex-6 FPGA supports a command called GCAPTURE that stores the current state of all flip-flops in hidden registers. This hidden registers contain the initial values of all flip-flops, they are used during the initial configuration of the FPGA to set them to the state specified in the Register-Transfer Level (RTL) model. The GCAPTURE command must be sent over ICAP to the device which then replaces the initial values of the flip-flops with the newly captured values [3].

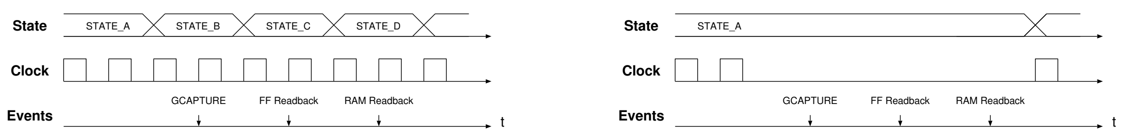
Per default this command operates on the whole FPGA which might not always be what we want. It is possible to constrain this command to only act on a subset of the FPGA, e.g. a reconfigurable region. Xilinx allows a user to set a property called RESET_AFTER_RECONFIG for partial reconfigurable regions which is used to tell the synthesis tools that a user wants his design to start in a known state after a reconfiguration. Internally this is done by a GSR event which is constrained to a subset of the FPGA.

The configuration memory of a Virtex-6 FPGA is divided into three sections [3]. The first section contains configuration data for CLBs, I/O and clocks. The second section contains the block RAM contents. The third section is called CFG_CLB and Xilinx provides no documentation for what it is used, they just say that a normal bitstream does not contain configuration data for this section.

By comparing partial bitstreams that were generated using the same netlist but once with the RESET_AFTER_RECONFIG property set and once without it, I discovered that the Xilinx tools insert one additional configuration frame into the partial bitstream if this property is set. Its frame address is `0x00400000` and thus it belongs to the CFG_CLB section of the FPGA. A comparison of different partial bitstreams with the RESET_AFTER_RECONFIG property set revealed that the content of this configuration frames is different for different reconfigurable areas. Also at the end of partial bitstreams with RESET_AFTER_RECONFIG set, the Xilinx tools inserted a GRESTORE command. I then tried to constrain the effects of GCAPTURE and GSR by writing this special configuration frames to the device prior to capturing some data and prior to GSR. With this special frame those two commands were then limited to just the area that I was interested in.

Although the GRESTORE command does not work, we can still extract the content of the CFG_CLB section from this bitstream and use it to constrain the GCAPTURE and the GSR commands.

After having extracted the frame addresses and the CFG_CLB section from the bitstream we have all information that is needed to perform a readback of the partial region. Only one last issue remains, namely the region has to be disconnected from the clock signal. Figure 4.2 shows why clock gating is needed for a successful readback. If no clock gating is done, then the logic inside the region continues to work and changes its state while the readback is performed. This is not a problem for the current state of flip-flops as this state is saved in the hidden registers and cannot be altered by the logic inside the region. But it is a problem for the state of block RAMs as the content of those RAMs is not saved in a hidden RAM, but must be read back from the RAM directly, thus the state of the RAM can potentially change between the start of the readback and its end. To avoid this problem we can simply stop the clock and the state of the reconfigurable partition stays the same for the complete readback. Details how clock gating was introduced to partial reconfiguration regions can be found in Section A.5.



(a) Without Clock Gating.

(b) With Clock Gating.

Figure 4.2.: GCAPTURE needs Clock Gating.

4.2.4. Restore Previous State

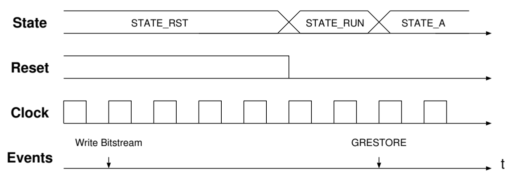
After having captured and read back a state from the FPGA, this data must be formatted so that it can be once again sent back to ICAP in order to be able to restore a previous state. The easiest way to do this is to copy the original partial bitstream that was used to reconfigure this partial region and replace all frames it contains with the frames that were read back. Frames that belong to block RAMs need some special handling which is explained in Section A.4. Also the original bitstream contained CRC checks that are now invalid. This CRC checks can be replaced by NOOP commands, the FPGA accepts bitstreams even when no CRC values are supplied, but it is not possible to simply remove those commands as was shown by Liu et al. [6].

As we need the RESET_AFTER_RECONFIG property anyway for capturing the state, our bitstream already contains the CFG_CLB section which we need to perform a GSR event, thus the bitstream is ready to be written to ICAP.

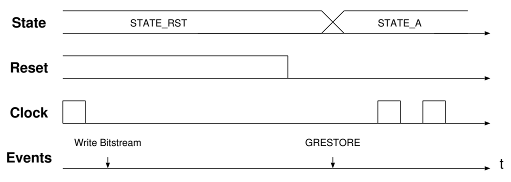
To do a restore the following steps need to be done:

1. Activate the reset signal and stop the clock for the reconfigurable partition
2. Write the bitstream to ICAP
3. Deactivate the reset signal
4. Perform GSR
5. Start the clock

The reset signal is needed to shutdown the interfaces to the OS and memory during reconfiguration. GSR is needed to return the captured state and clock gating needs to be performed as otherwise we might end up in an invalid state, see also Figure 4.3. Similar to the capturing of the current state, the RAM is not restored using GSR, so we need to ensure that it is not changed until the flip-flops are in their previous state too.



(a) Without Clock Gating.



(b) With Clock Gating.

Figure 4.3.: GRESTORE needs Clock Gating.

Chapter 5

Results

5.1. Test Setup

5.1.1. Overview

To evaluate the HWT_ICAP core and its associated software, a ReconOS setup with two reconfigurable slots was used.

The first slot contained either a simple adder or a subtracter. These two hardware threads were mainly intended to verify that writing to ICAP using HWT_ICAP works reliably. These hardware threads are explained in Section 5.1.2.

The second slot contained either a ”dummy” multiplier or a Linear Feedback Shift Register (LFSR). Those threads were intended to test the capturing and restoring of states. Both threads perform functions that run for a long time or even forever, which allows the testing of cycle-true state restoration. If not all sequential elements contain data that belongs to the same cycle after restoration, it will not work. The multiplier is explained in Section 5.1.3 and the LFSR in Section 5.1.4.

5.1.2. Adder / Subtracter Threads

This hardware thread contains 4× 32-bit registers and a local RAM. Three of four registers can be set via messages over the operating system interface, all of them can be read via messages. It is also possible to copy data from main memory to the hardware thread and vice-versa. The data in the local RAM is not modified by this thread.

The third register is the target register of an addition / subtraction of the first two registers and can be used to check which of the two different modules is currently loaded in the reconfigurable slot.

The fourth register and the local RAM are only included to be able to test state capturing and restoring, as data in those storage elements is not touched by any calculation of the thread, so it is easy to check if a restoration was successful.

5.1.3. Dummy Multiplier Thread

This hardware thread was mainly created to check if state restoring is cycle true and that no data was lost. Its architecture is similar to the ADD / SUB thread as it also contains 4 registers and a local RAM. The RAM is again not modified by this thread.

This thread performs a multiplication of the values in the first two registers. It does this by a naive multiplication algorithm which was chosen to be slow, so that this thread is active for a long time. It does multiplications by a loop in which the value in the first register is decremented in each iteration and the value in the second register is added to the result register. It does this until the value in the first register has reached zero, then the multiplication is finished.

The multiplication result is available in the third register, while the fourth register contains only one bit which says if the multiplication is still running or if it has already finished. By choosing the value in the first register a user can control how many cycles this thread needs to run in order to calculate its result.

5.1.4. Linear Feedback Shift Register Thread

This hardware thread runs several 16-bit LFSRs. The hardware actually contains only one real shift register which is shared between all LFSRs. To do this the LFSR values are stored in a block RAM and only inserted in the hardware shift register for one cycle before being put back into the RAM, see Figure 5.1. The number of LFSRs that can be run on this hardware thread is thus only limited by the amount of local RAM, for the evaluation four LFSRs were used. It would also be possible to have several LFSRs running on this hardware thread without using a RAM, but a hardware thread was needed that actually needs the RAM contents which was why this thread has such a special architecture.

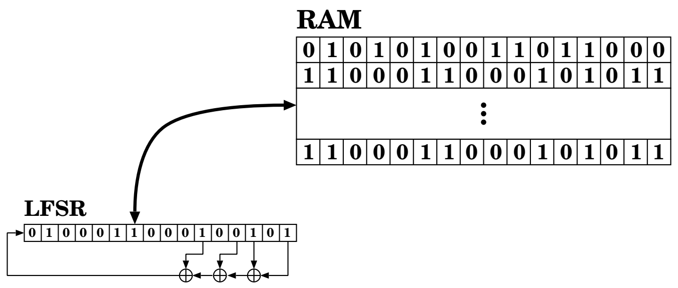


Figure 5.1.: LFSR Hardware Thread.

Using the OS interface it is possible to set a start value for each LFSR. The thread counts the number of cycles the LFSR was run since it was last loaded with a new value. By reading the current value of the cycle counter and all values of the LFSRs, it is possible to check if the LFSR is in the correct state.

5.2. Functional Verification

Functional verification of HWT_ICAP was done by first configuring the ADD and MUL hardware threads on the FPGA. It is then tested if those two hardware threads work correctly. After that the ADD thread is replaced by the SUB thread and the MUL thread by the LFSR thread. Again their correct functionality is tested. This process was then repeated one hundred times. The reconfiguration always worked correctly and the hardware threads showed no issues after being reconfigured, so I assume that the new HWT_ICAP core performs reconfigurations reliably.

The functional verification of state capturing and restoring was done in several steps, as state capturing and restoring was separately tested for each hardware thread.

The ADD and SUB threads were tested by first configuring them on the FPGA and setting their registers and local RAM to a defined state. Then they were captured and the registers on the FPGA were overwritten. After that another thread was loaded on that reconfigurable slot, i.e. the SUB thread if the ADD thread was captured and vice-versa. Now the previously captured state was restored and it was checked if the values in the registers and the local RAM match the values that were set when the thread was captured.

For the MUL and LFSR hardware threads a similar procedure was used, but this time the threads were active when they were captured. For the MUL thread this means that a lengthy multiplication is started, while for LFSR thread this means that some known values are loaded before it was captured. So after state restoration it was necessary to wait for the completion of the calculation of the MUL thread before it was possible to check if its result is indeed correct. For the LFSR thread the current shift register values and the number of cycles that it was run, were read and compared to a simulation model of the shift register.

The state capturing and restoration was done several hundred times for all hardware threads. Of all those experiments about 1% has failed, because a segmentation fault occurred, the hardware thread did not respond to a message sent to it or the Linux kernel has crashed. I was not able to find the cause of these problems. As they are not deterministic, it is difficult to solve them. Also it is unclear where these problems come from, as they can come from one of the hardware threads, from the software I have written, the ReconOS project and even the Linux kernel.

5.3. Performance Evaluation

5.3.1. Reconfiguration

Performance of the XPS_HWICAP core and the new HWT_ICAP core were measured by reconfiguring a slot one hundred times. The time is measured that was needed to write the partial bitstream to ICAP and the corresponding bandwidth was calculated. Figure 5.2 shows a comparison of three different bitstream sizes across different RAM sizes and implementation options for the HWT_ICAP core. Note that the size of the local RAM does not seem to be important as only small variations can be seen for the double buffering case. Also the implementation that uses no local RAM is not significantly faster than the double buffering implementations. All this indicates that the core has already hit the memory bandwidth limit of the ReconOS system.

Since the local RAM of the HWT_ICAP core will be mapped to a block RAM on the FPGA anyway, a RAM size smaller than 2 kilobytes does not save any resources as the smallest block RAM available on a Virtex-6 has a size of 18 kilobits.

We can see that double buffering achieves a significant performance gain of about 20% against single buffering with the same RAM size.

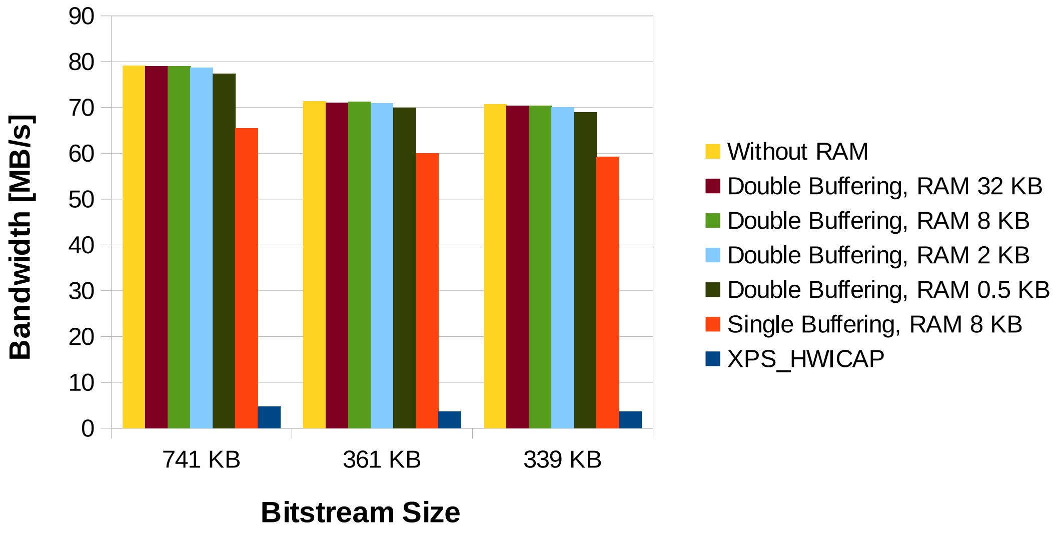


Figure 5.2.: HWT_ICAP Write Performance.

Table 5.1 shows a comparison between the old XPS_HWICAP core and the new HWT_ICAP core using double buffering and 8 KB of local RAM. The new core is about $16 - 19$ times faster as the old one, depending on the bitstream size.

Table 5.1.: XPS_HWICAP and HWT_ICAP Write Performance.

ICAP Core	Bitstream Size	Reconfiguration Time	Bandwidth
XPS_HWICAP	741 KB	150.7 ms	4.8 MB/s
HWT_ICAP	741 KB	9.2 ms	79.0 MB/s
XPS_HWICAP	361 KB	95.4 ms	3.7 MB/s
HWT_ICAP	361 KB	4.9 ms	71.3 MB/s
XPS_HWICAP	339 KB	91.6 ms	3.6 MB/s
HWT_ICAP	339 KB	4.7 ms	70.4 MB/s

5.3.2. Readback / State Restoration

Readback

The readback performance was measured by reading back typical numbers of partial configuration frames, Table 5.2 and Figure 5.3 show the results. One can see that the performance is higher for a larger number of frames, so there seems to be a considerable overhead that influences the performance negatively.

The HWT_ICAP core with 8 KB RAM seems to be a bit faster than the one with 32 KB of local RAM. This effect is related to the way the ReconOS memory interface is constructed. The memory interface is built from two FIFO queues, one in the direction of the main memory and one in the opposite direction. The FIFO can hold up to 128 elements, thus it is much smaller than the local RAM of HWT_ICAP. As we are only doing single buffering for readbacks, the FIFO is going to be full nearly all the time when data is transferred from the local RAM to the main memory, but it will soon be empty when data is read from ICAP and stored in the local RAM. For a small memory the amount of time that the FIFO is empty is smaller than for a larger memory, thus a smaller local RAM makes single buffering faster for readbacks. Double buffering for readbacks would probably improve this situation.

Table 5.2.: HWT_ICAP Read Performance.

RAM Size	Total Size	#Frames	Readback Time	Bandwidth
32 KB	120 KB	381	2277 us	51.7 MB/s
8 KB	120 KB	381	2256 us	52.2 MB/s
32 KB	98 KB	309	2000 us	47.7 MB/s
8 KB	98 KB	309	1968 us	48.5 MB/s
32 KB	41 KB	129	1396 us	28.6 MB/s
8 KB	41 KB	129	1377 us	29.0 MB/s

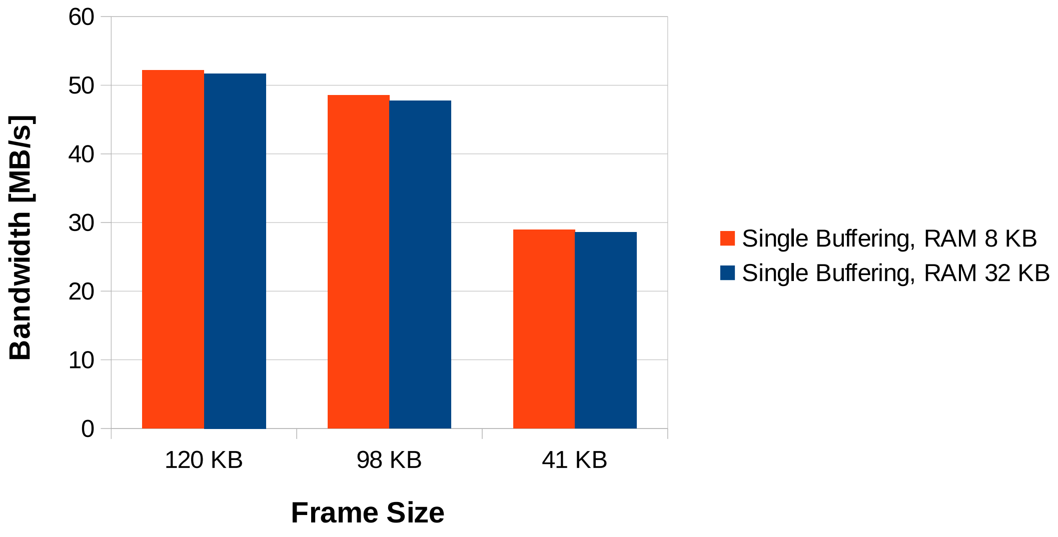


Figure 5.3.: HWT_ICAP Read Performance.

State Saving & Restoring

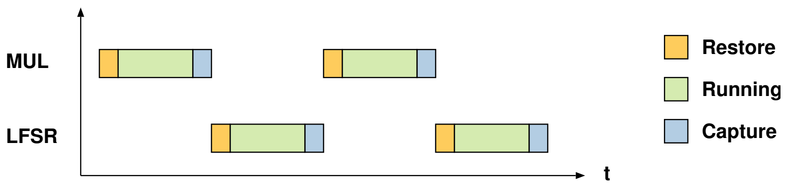


Figure 5.4.: Hardware Thread Swapping.

The time it needs to do capturing and restoration of a hardware thread was measured to estimate how much overhead one can expect when preemptive hardware scheduling is used. For this measurements the two hardware threads LFSR and MUL were scheduled on the same reconfigurable slot and each only got a fraction of the hardware time. After a random time the currently running thread was captured and the other one restored, see Figure 5.4. For this measurement only double buffering for writing to the ICAP interface and single buffering for reading were considered. It was not possible to use XPS_HWICAP for readbacks due to bugs in the Linux kernel driver, thus it is not possible to compare it to HWT_ICAP here.

Table 5.3 shows the results of the measurements. The column normalized time is intended as an indicator, so that different bitstream sizes can be compared. It is calculated by dividing the total time by the bitstream size. The reason why the larger bitstream yields a much lower normalized time is that readback time also depends on the size and number of frames in the bitstream. As multiple configuration frames that are stored at continuous addresses in the FPGA configuration memory can be read with only one request to ICAP, readback is much more efficient for cases where less requests are needed. In this case both the 741 KB and the 361 KB bitstreams can be read back with two requests each. If two bitstreams would be compared that have the same size but one needs twice the number of requests, then the bitstream with fewer requests would yield a much lower capturing time because the overhead for a readback is reduced.

Table 5.3.: HWT_ICAP State Restoration Performance.

Ram Size	**Bitstream Size**	**Capturing**	**Restoring**	**Total**	**Normalized**
32 KB	741 KB	16.1 ms	9.6 ms	25.7 ms	34.7 μs/KB
8 KB	741 KB	16.0 ms	9.7 ms	25.7 ms	34.7 μs/KB
32 KB	361 KB	10.4 ms	5.5 ms	15.9 ms	43.9 μs/KB
8 KB	361 KB	10.3 ms	5.5 ms	15.8 ms	43.7 μs/KB

By comparing Table 5.1 with Table 5.3 it can be seen that state restoration takes about three times as long as a reconfiguration of a bitstream with the same size.

5.4. FPGA Resource Usage

Table 5.4 shows the FPGA resources used by the different variants of HWT_ICAP and XPS_HWICAP. The table also includes values for the Fifo32 which is used for the memory interface of a hardware thread, Fast Simplex Link (FSL) which is used for the OS interface of a hardware thread and the PLB bus which is used by XPS_HWICAP to communicate with the CPU. These values are estimates from synthesis, so post map results would probably look slightly different.

It can be seen that the size of the local RAM does only influence the number of BRAMs used for HWT_ICAP. The single buffering case needs significantly less resources than the double buffering case, this is misleading and comes from the fact that the single buffering core does not support readbacks and therefor it is much simpler. Of all HWT_ICAP variants the implementation without RAMs is the most efficient in terms of resources which is not surprising. Therefor if only reconfiguration and no readbacks from ICAP are needed, the HWT_ICAP core without RAM should be used as it is the fastest and smallest core available.

It is difficult to compare the XPS_HWICAP core with the HWT_ICAP core directly as both cores need additional interfaces to perform their duties and those interfaces have to be considered in the comparison. For HWT_ICAP the Fifo32 and FSL interfaces have to be added to the resource usage of HWT_ICAP, while for the XPS_HWICAP core the

PLB bus must be considered. The PLB bus is not only used by XPS_HWICAP but also by other peripherals which makes it difficult to put a number on the amount of resources consumed by XPS_HWICAP. If the interfaces are neglected, then all HWT_ICAP variants with less than 8 KB RAM use strictly less resources than XPS_HWICAP does. If the interfaces are included, the picture is not so clear anymore. Still the HWT_ICAP variant that has no local RAM is superior to XPS_HWICAP and needs significantly less resources.

Table 5.4.: Resource Usage of HWT_ICAP and XPS_HWICAP.

ICAP Core	**#Flip-flops**	**#LUTs**	**#BRAMs**
HWT_ICAP, Dbl. Buff., RAM 32 KB	331	749	8
HWT_ICAP, Dbl. Buff., RAM 8 KB	323	741	2
HWT_ICAP, Dbl. Buff., RAM 2 KB	314	733	1
HWT_ICAP, Dbl. Buff., RAM 0.5 KB	304	709	1
HWT_ICAP, Sngl. Buff., RAM 8 KB	252	465	2
HWT_ICAP, Without RAM	213	300	0
XPS_HWICAP	750	804	1
Fifo32	14	152	0
FSL	14	152	0
PLB Bus	170	569	0

Chapter 6

Conclusion and Future Work

6.1. Contributions

In this semester thesis I have developed a new ReconOS hardware thread (HWT_ICAP) that is able to perform FPGA reconfigurations using the ICAP interface. After the CPU has started a reconfiguration using HWT_ICAP, it is no longer involved in the reconfiguration. I have created several versions of this hardware thread with different resource needs, so that a user is able to select the version that fits his purpose best. Reconfiguration with this new HWT_ICAP core works reliably, I have not found any issues during testing and performance evaluation.

I have integrated configuration readback capabilities into HWT_ICAP which allowed me to do state capturing and state restoring. For state capturing I wrote a software that parses partial configuration bitstreams and extracts the information needed for state capturing. The software then performs state capturing using the HWT_ICAP core and creates a new bitstream that can be used for state restoring.

State restoration is also done using the HWT_ICAP core. The state saving and restoring process still has some issues left which lead to crashes of the CPU, however those crashes happen in only 1 % of all cases.

Since clock gating is needed for successful capturing and restoring of FPGA state, I have implemented a core that takes a clock signal as an input and outputs a gated clock. If this core is used to perform clock gating on the hardware threads, then no changes in the source code of these hardware threads are needed.

6.2. Conclusion

The goal of this semester thesis was reached as the developed HWT_ICAP core does reconfigurations 16 to 19 times faster than the currently used XPS_HWICAP core. Also the new core works independently of the CPU, so that it is freed from reconfigurations and can run other tasks while a reconfiguration is performed. The HWT_ICAP core seems to have reached the peak memory bandwidth that the ReconOS system provides. In order to get higher reconfiguration speeds, this limit would have to be circumvented.

A proof-of-concept implementation of state capturing and restoration on a Virtex-6 FPGA was developed during this semester thesis. This implementation allows preemptive hardware scheduling on the ReconOS system and requires no changes to the existing hardware threads. State capturing and restoring takes only two to three times as long as reconfiguration and therefor allows very fast task switching. For a 361 KB bitstream state capturing and restoring takes 15.8 ms which allows for over 60 task swaps per second.

6.3. Future Work

6.3.1. Circumventing Memory Bandwidth Limit

To get faster with the given memory bandwidth, one could cache the partial bitstream in a block RAM on the ICAP core [8]. This has some drawbacks as the bitstream has to be transferred to the core before it is needed for reconfiguration and it wastes many block RAMs. Another possibility is to use bitstream compression. Compression ratios of up to 75% have been reported [6, 7], so with a memory bandwidth of about 80 MB/s in the ReconOS system a reconfiguration performance of over 300 MB/s seems possible.

6.3.2. Stability Issues

State capturing and restoration has some stability issues in the current implementation, in about 1% of all restoration attempts a segmentation fault occurs or the restored hardware thread does not answer to a message sent to it over the OS interface. I did not manage to find the reason why segmentation faults occur during state restoration, it is even possible that their origin could be somewhere else than in the code that I have written during this project. In future work this stability issues should be investigated.

6.3.3. ReconOS Interface

The interfaces that ReconOS uses for hardware threads is not well suited for preemptive multitasking. ReconOS uses FIFO queues for both the operating system interface and for the memory interface. When a hardware thread is captured and then replaced by a different thread, those FIFO queues are cleared and its content is lost. If there were messages waiting in those queues, then those messages are lost and cannot be recovered.

Similarly if a delegate thread is in a specific state that is different from the state that the reconfigured thread assumes, then they will be out of sync and no messages can be exchanged between them anymore. For an explanation of delegate threads in the ReconOS environment see [4]. A solution to this problem would be to save the state of a delegate thread together with the hardware thread itself. ReconOS does not provide support for this right now, so this would have be developed.

6.3.4. Faster State Restoration

The speed of state capturing and state restoration could be improved by developing a more sophisticated ICAP core that handles the capturing entirely in hardware.

6.3.5. Zynq Platform (Xilinx 7-Series Devices)

ReconOS version 3.1 now supports the 7-Series FPGAs from Xilinx, namely the Zynq platform. HWT_ICAP could be ported to this new platform and tested if everything I have done for the Virtex-6 platform also works for the Zynq. It would also be interesting to see if the stability issues mentioned above also exist on this platform.

Appendix A

Lessons Learned

A.1. Two ICAP Interfaces

Virtex-6 FPGAs contain two ICAP interfaces, a fact that is not well documented. We discovered this during the development of the HWT_ICAP core and hoped that we could use one ICAP interface for the new core and the other one for the XPS_HWICAP core, making it possible to compare the performance of the two cores directly.

It turned out that out of the two interfaces, only one is active at a time. Per default this is the ICAP interface on the top half of the FPGA. When an ICAP interface is not active, it outputs zeros and does not react to any of its inputs. As my new core was always mapped to the bottom ICAP interface by the Xilinx tools, while the old core was mapped to the top one, my core never did anything. It was necessary to either remove the other core or force the mapping of the new core to the top ICAP interface.

It later turned out that it is possible to switch the active interface by sending some configuration commands over the active ICAP interface. It was thus possible for me to change the active interface from the top interface to the bottom one using my new core. But for some reason I was not able to switch back to the top one using the XPS_HWICAP core.

A.2. ICAP Interface Issues

The ICAP interface has slow timings. For example the propagation delay of the BUSY signal is 6.909 ns, while the setup time of the Chip Select (CS) and Read/Write select (RDWR) signals is 4.017 ns. This means that it is not possible to use a combinational function like in Figure A.1 to react to the BUSY signal in the same cycle if a 100 MHz clock is used. This problem can be solved by inserting a register for the BUSY signal, but it is important to know about this behavior in order to be able to design an ICAP core.

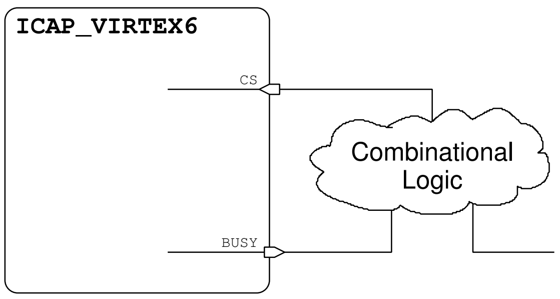


Figure A.1.: ICAP Interface Timing Problems.

Although writing to the ICAP interface is rather easy, reading from it is not straightforward. After initiating a readback and enabling the interface (CS is set low), it takes three cycles until it outputs valid data [3] and the BUSY signal goes low. The BUSY signal shows the user when readback data is valid, it goes high as soon as CS is disabled, but readback data continues to be output for 3 additional cycles after CS is disabled. Figure A.2 show this behaviour. Note that CS is an active low signal.

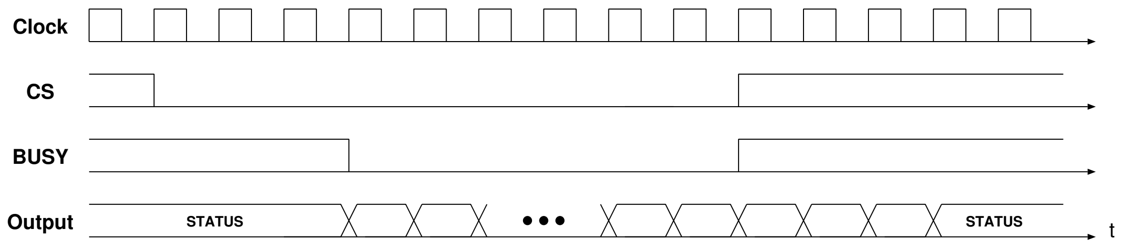


Figure A.2.: ICAP Interface Readback Timing Diagram.

It is also important to note that when more data is tried to be read from ICAP than was requested by the last command, it stops after the specified amount of data and does not output anything. It is thus important to specify exactly the number of words one intends to read as otherwise the ICAP core may hang as it waits for more output data which will never come.

Also note that if less data is read than was requested, the ICAP interface is in an unspecified state and does not react to commands anymore.

All this issues are especially bad because Xilinx does not provide a simulation model for their ICAP interface and one thus has to look at the hardware to understand how it works.

A.3. GRESTORE does not work

The GRESTORE command is intended to reset all flip-flops of a specific region to their initial state. This command performs a GSR event which can also be triggered by manually setting GSR on the STARTUP_VIRTEX6 primitive [3]. It turned out that this command does not work on the Virtex-6 that I was using for this project.

To check if it really does not work I used a partial bitstream that has the RESET_AFTER_RECONFIG property set. If this property is set the FPGA should reset the flip-flops of this partial region to their initial state. The initial state of some flip-flops was specified in VHDL and could be read by the CPU. Analysis of the partial bitstream showed that it contains the GRESTORE command at its end, so if this bitstream is programmed into the device, the flip-flops should be in their initial state.

Configuration via ICAP and JTAG showed the same result, the flip-flops did not return to their initial state but were in a random state that was probably a leftover from the previous configuration. As the bitstream was generated using Xilinx tools and not modified by anything, I therefor conclude that the GRESTORE command is in fact broken. If a GSR should be performed, one thus has to use the STARTUP_VIRTEX6 primitive.

A.4. Block RAM Restoration

It turned out that block RAMs could not be as easily restored as flip-flops. After I had found a way to use GSR to restore the state of flip-flops, I wanted to do the same for block RAMs. In my first attempts the content of a block RAM was not restored even when the flip-flops in the same region returned to their initial / captured states. I then compared the bitstream of my captured state with a partial bitstream that was generated using `bitgen`. Both bitstreams contained the same RAM contents, so they could be compared directly. The captured bitstream had some additional bits set in the block RAM configuration frames compared to the genuine bitstream, interestingly those bits were placed in regular intervals. When I compared a different region using the same method, those additional bits were also set and they followed the same pattern.

I then tried to set those bits to zero prior to writing the captured bitstream back to the device and this worked perfectly, the block RAM contents were now also restored to their captured state. Thus I believe that those bits control if the memory contents should be overwritten by the values given in the bitstream.

A.5. Clock Gating

Since clock gating was needed to ensure proper capturing and restoring of FPGA states, I wanted to implement it with as little changes as possible in the reconfigurable hardware threads, so I decided to disable the whole clock net that leads to a reconfigurable area.

Correct clock gating is difficult because glitching and other issues have to be avoided [13]. Since this project runs on an FPGA, it is not possible to create entirely new components for clock gating and one thus has to use what is available on the FPGA. The Virtex-6 platform has three primitives that are intended for clock gating purposes [14]:

- **BUFGCE**. A global clock buffer with enable pin.
- **BUFR**. A regional clock buffer with an optional clock divider.
- **BUFHCE**. A horizontal clock buffer with an enable pin.

It turned out that BUFHCE leads to terrible hold time problems. I think that the reason for this is that using BUFHCE upsets clock tree routing in reconfigurable regions. BUFR were not mappable when a reconfigurable region spanned a large portion of a clock region of the FPGA. Finally BUFGCE worked well for all designs that were tested. Using BUFGCE has only one limitation, there are only 32 global clock buffers available on a Virtex-6 FPGA, so this limits the number of reconfigurable partitions that can be created.

Note that clock buffers cannot be instantiated inside reconfigurable regions [11].

Appendix B

Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Semester Thesis

Resource-efficient Dynamic Partial Reconfiguration on FPGAs

Andreas Traber

Advisor: Dr. Markus Happe, markus.happe@tik.ee.ethz.ch
Co-Advisor: Ariane Trammell, ariane.trammell@tik.ee.ethz.ch
Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

17 March 2014 - 20 June 2014

1 Introduction

System-on-Chip (SoC) architectures can implement an entire computing system on a single chip. This includes a central processor, a memory controller, dedicated hardware accelerators, and several peripherals. Reconfigurable SoCs (RSoCs) support to partially adapt the hardware architecture at run-time. Using partial reconfiguration of the chip, an RSoC can support a large library of hardware accelerators, which are initially stored in (cheap) external memory. At run-time the system identifies, which hardware accelerators promise the most performance speedup and partially reconfigures the chip to map these accelerators to the FPGA fabric. The chip area is strictly constrained, therefore only a limited number of hardware accelerators can be mapped to the FPGA at the same time. When the workload of the system changes, this mapping needs to be updated. Therefore a resource-efficient way is required to reconfigure the chip.

ReconOS is a execution environment for RSoCs, which connects multiple so-called hardware threads to a central processor. ReconOS extends the multithreading approach to reconfigurable hardware. Here, hardware threads can share operating system resources, such as semaphores, mutexes or message boxes with software threads that are executed on the central processor. The hardware threads is connected to the processor using a operating system interface and to the main memory using a memory interface. ReconOS supports dynamic partial reconfiguration, i.e. hardware threads can be reconfigured at run-time. For this reason, multiple reconfigurable hardware slots (rectangular areas on the chip) are reserved on the FPGA fabric. Each slot can contain at most one hardware thread at a time. In the current ReconOS version, a software driver reconfigures these reconfigurable hardware slots. However, this means that the processor is busy for the entire duration of the reconfiguration. As a result, the system freezes for the entire reconfiguration time. It is the goal of this semester thesis to drastically decrease this overhead by introducing a new hardware thread, which reads the configuration logic from the main memory and feeds it to the Internal Configuration Access Port (ICAP) of the system.

2 Assignment

This assignment aims to outline the work to be conducted during this thesis. The assignment may need to be adapted over the course of the project.

2.1 Objectives

The goal of this semester thesis is to design and implement a new ReconOS hardware thread, which reads the configuration logic of a partial bitstream from the main memory and feeds it to the Internal Configuration Access Port (ICAP) of the system. The expected outcome of the semester thesis is a hardware controller that can reconfigure hardware slots with minimal interaction with the central processor and improves the reconfiguration performance compared to the Xilinx XPS_HWICAP hardware module.

2.2 Tasks

This section gives a brief overview of the tasks the student is expected to perform towards achieving the objective outlined above. The binding project plan will be derived over the course of the first three weeks depending on the knowledge and skills the student brings into the project.

2.2.1 Familiarization

- Xilinx Design Tools (XPS, SDK, PlanAhead, Partial Reconfiguration Toolflow, Isim, ChipScope)
- ReconOS v3.0 architecture, execution environment and VHDL libraries
- ICAP module for Virtex-6 FPGAs
- In collaboration with the advisor, derive a project plan for your semester project. Allow time for the design, implementation, evaluation, and documentation.

2.2.2 Architecture and hardware design

- Develop a ReconOS hardware thread that receives the memory address and the size of a partial bitstream (which is stored in main memory) from the processor in order to reconfigure this partial bitstream using ICAP.
- Optional: Improve the hardware design. For instance, double buffering could be used to increase the performance of the hardware thread
- Optional: Extend the hardware thread to allow users to read configuration frames to main memory.

2.2.3 Implementation

- Determine an appropriate version control system and set it up for further use. You might consider using git and branch the official ReconOS git repository into your git repository.
- Implement the hardware ICAP controller on a Xilinx Virtex-6 ML605 board.

2.2.4 Validation

- Validate the correct operation of your implementation.
- Check the resilience of the implementation, including its configuration interface, to uneducated users.

2.2.5 Evaluation

- Do a performance evaluation of your implementation. This includes a stress test, in order to verify that your hardware thread does not introduce any instabilities into the overall system.
- Provide a performance comparison between your ICAP controller and the Xilinx XPS_HWICAP module.
- Optional: Experiment with different sizes of the local memory.

2.2.6 Documentation

- Provide appropriate source code documentation.
- Write a step-by-step how to that describes the compilation of your code, the loading of the code into the hardware and the execution of your code.

3 Milestones

- Provide a project plan, which identifies the milestones.
- One intermediate presentations: Give a presentation of ten minutes to the professor and the advisors. In this presentation, the student presents major aspects of the ongoing work including results, obstacles, and remaining work.
- Final presentation of 15 minutes in the CSG group meeting, or, alternatively, via teleconference. The presentation should carefully introduce the setting and fundamental assumptions of the project. The main part should focus on the major results and conclusions from the work.
- Any software and hardware modules that is produced in the context of this thesis and its documentation needs to be delivered before conclusion of the thesis. This includes all source code and documentation. The source files for the final report and all data, scripts and tools developed to generate the figures of the report must be included. Preferred format for delivery is a CD-R.
- Final report: The final report must contain a summary, the assignment, the time schedule and the Declaration of Originality. Its structure should include the following sections: Introduction, Background/Related Work, Design/Methodology, Validation/Evaluation, Conclusion, and Future work. Related work must be referenced appropriately.

4 Organization

- Student and advisor hold a weekly meeting to discuss progress of work and next steps. The student should not hesitate to contact the advisor at any time. The common goal of the advisor and the student is to maximize the outcome of the project.
- The student is encouraged to write all reports in English; German is accepted as well.
- The core source code will be published under the GNU general public license.

5 References

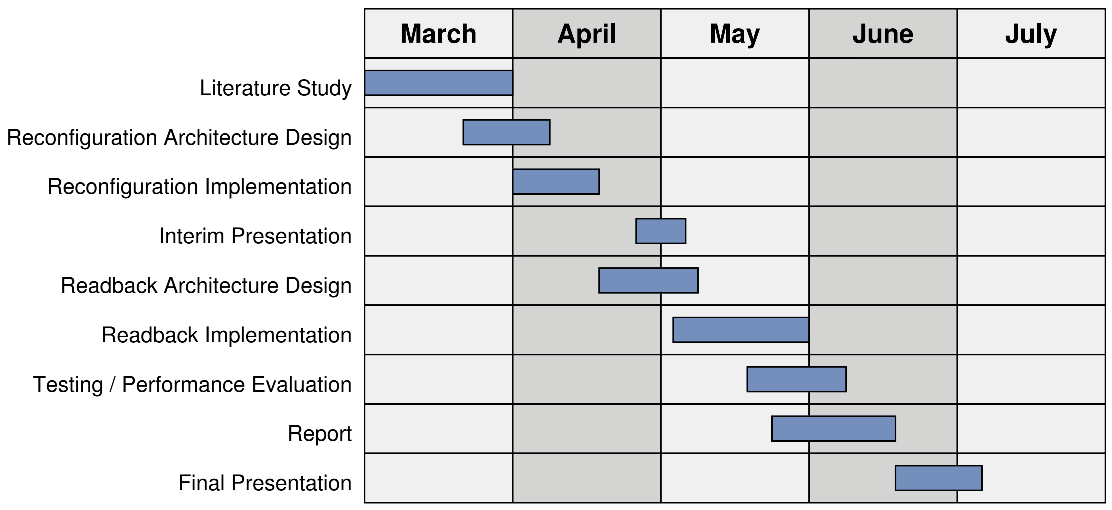
[1] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. ,,ReconOS – An Operating System Approach for Reconfigurable Computing". In IEEE Micro, Nov. 2013. (PrePrint)
[3] Simen Gimle Hansen, Dirk Koch, and Jim Torresen. ,,High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro". In IEEE International Parallel and Distributed Processing Symposium 2011
[4] Ming Liu, Wolfgang Kuehn, Zhonghai Lu and Axel Jantsch.,,Run-time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration". In IEEE Conference on Field Programmable Logic and Applications (FPL) 2009.
[5] Julien Delorme, Amor Nafkha, Pierre Leray, and Christophe Moy.,,New OPBHWICAP interface for Realtime Partial reconfiguration of FPGA". In IEEE Conference on Reconfigurable Computing and FPGAs (ReConFig) 2009
[6] Git Repository: https://github.com/EPiCS/reconos (branch: v3.0_dev)

User Guides:
[6] UG 702: Partial Reconfiguration (v14.5) `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug702.pdf`
[7] UG 360: Virtex-6 FPGA Configuration (v3.6) `http://www.xilinx.com/support/documentation/user_guides/ug360.pdf`
[8] ,,Driving ICAP Resource" (slides): `http://home.mit.bme.hu/~feher/Reconf_Comp/10_Driving_ICAP.pdf`

Webpages:
`http://www.epics-project.eu`
`http://www.reconos.de`

Appendix C

Project Plan



Appendix D

Project Files & How-to

D.1. File Structure

The source code for HWT_ICAP and its associated software can be found on GitHub: `https://github.com/atraber/reconos/tree/v3.0_dev`

The git repository is organized as follows:

```
demos/icap_dev ............. Contains everything that is related to HWT_ICAP.
└─hw
  ├─edk ................................... Contains the project files for XPS.
  ├─pcores ................... Contains all cores developed during this project.
  │ ├─hwt_icap_simple_v1_00_a ....... HWT_ICAP core without local RAM.
  │ ├─hwt_icap_v1_00_a ........... HWT_ICAP core with readback support.
  │ ├─hwt_icap_single_v1_00_a ...... HWT_ICAP that uses single buffering.
  │ ├─hwt_pr_block_v1_00_a ................... ADD/SUB hardware thread.
  │ ├─hwt_pr_lfsr_v1_00_a ......................... LFSR hardware thread.
  │ └─hwt_pr_mul_v1_00_a ........................... MUL hardware thread.
  ├─pr_design ........................ Mapping, P&R and bitgen is done here.
  ├─pr_design_swap Almost identical to pr_design but reconfigurable regions for
  │ slot 1 and 2 are swapped.
  ├─pr_design_large ...... Similar to pr_design but for different reconfigurable
  │ regions.
  └─all.sh .............................. Executes the complete PR toolflow
└─sw ...................................... Contains software for HWT_ICAP.
  └─test_icap ...................................... Source files for software.
    └─test_icap.c ...................................... Contains main().
```

Note that only hwt_icap_v1_00_a has support for readback and GSR. It uses double buffering for writes and single buffering for reads.
hwt_icap_single_v1_00_a is an early implementation of the hardware thread and only uses single buffering for writes.

For applications where readback and GSR is not required hwt_icap_simple_v1_00_a is probably the best choice as it has the lowest resource consumption and also is the fastest implementation.

D.2. How-to

This section gives a short overview over the partial reconfiguration toolflow for ReconOS and explains the software that was developed to test HWT_ICAP.

D.2.1. Partial Reconfiguration Toolflow

You must have a working Xilinx ISE Design Suite v14.7.

1. Clone the Git repository and checkout the branch used for this project.
   ```
   git clone https://github.com/atraber/reconos.git
   cd reconos
   git checkout v3.0_dev
   ```
2. Generate netlists. This step creates two netlists, one for configuration a and one for configuration b.
   ```
   cd demos/icap_dev/hw/edk
   ./generate_netlist.sh
   cd ..
   ```
3. Generate bitstream for both configurations.
   ```
   cd pr_design/imp/config_pr_a
   ./generate_config.sh
   cd ../config_pr_b
   ./generate_config.sh
   ```
4. The full and partial bitstreams are ready and a full bitstream can now be loaded on the FPGA:
   ```
   cd ../../bitfiles
   dow ./system_add.bit
   ```
5. The partial bitstreams must be copied to the compact flash card of your FPGA or to the nfs root directory:
   ```
   cp ./partial*.bin cf_dir/partial_bitstreams/
   ```

The default toolflow uses the HWT_ICAP core with double buffering and 8 KB of RAM. If a different core should be used, then the symbolic link in `demos/icap_dev/hw/edk/pcores/hwt_icap_v1_00_a` can be changed, so that it points to hwt_icap_simple or hwt_icap_single.

All the steps mentioned above can also be done by executing `all.sh` in the `hw` directory, except that the bitstream is not automatically loaded on the FPGA and that the partial bitstreams are not copied to the compact flash.

D.2.2. Software

You must have a working gcc toolchain for the MicroBlaze CPU.

1. Change to the git root directory.
2. Prepare the ReconOS system.
   ```
   cd linux/reconos/libreconos-us
   make
   cd ../../..
   ```
3. Compile the software.
   ```
   cd demos/icap_dev/sw/test_icap/
   make
   ```
4. Copy the compiled program to the compact flash card of your FPGA or to the nfs root directory:
   ```
   cp ./icap_demo cf_dir/
   ```
5. Download the Linux kernel image to the FPGA. The device tree that was used to generate this kernel can be found here:
   `git_root/demos/icap_dev/hw/device_tree/icap_2slots.dts`
6. Run the icap_demo program on the FPGA
   Try `./icap_demo -h` for an overview over the arguments it supports. The most interesting options are explained below:
   - `./icap_demp -w 10`
     Performs 10 reconfigurations in a loop using HWT_ICAP. Every reconfiguration iteration consists of loading and testing the ADD and MUL hardware threads first, then the same is done for the SUB and LFSR hardware threads.
   - `./icap_demo -w 10 --linux`
     The same as above but using XPS_HWICAP.

- `./icap_demo --test_swap=10`
  Performs 10 state capturing and state restoring rounds with the MUL and LFSR hardware threads.
- `./icap_demo --test_add`
  Performs state capturing and state restoring for the ADD hardware thread. First the ADD hardware thread is loaded and its state captured, then it is replaced by a SUB thread and later the captured state of ADD is restored.
- `./icap_demo --test_lfsr`
  Similar to `--test_add` but for the LFSR hardware thread.

Appendix E

Diagrams

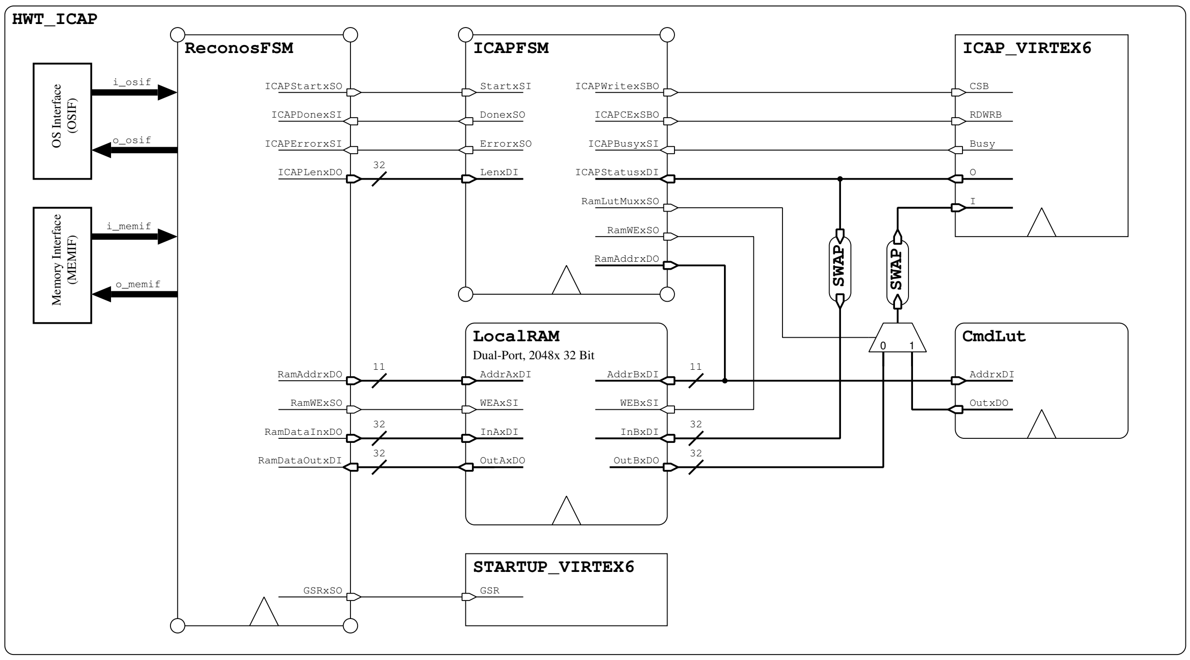


Figure E.1.: Detailed Block Diagram of HWT_ICAP.

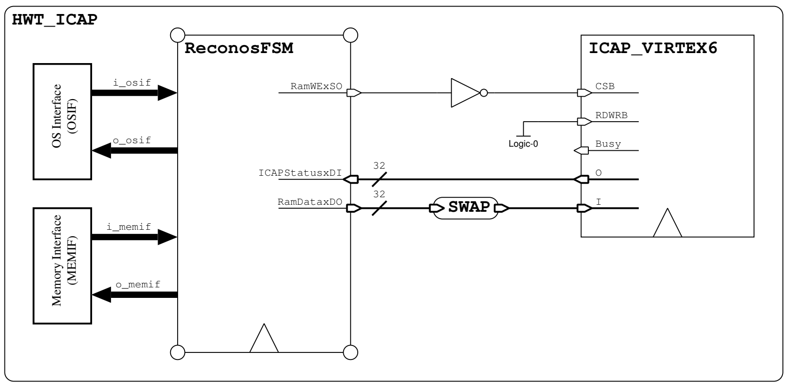


Figure E.2.: Block Diagram of HWT_ICAP without local RAM.

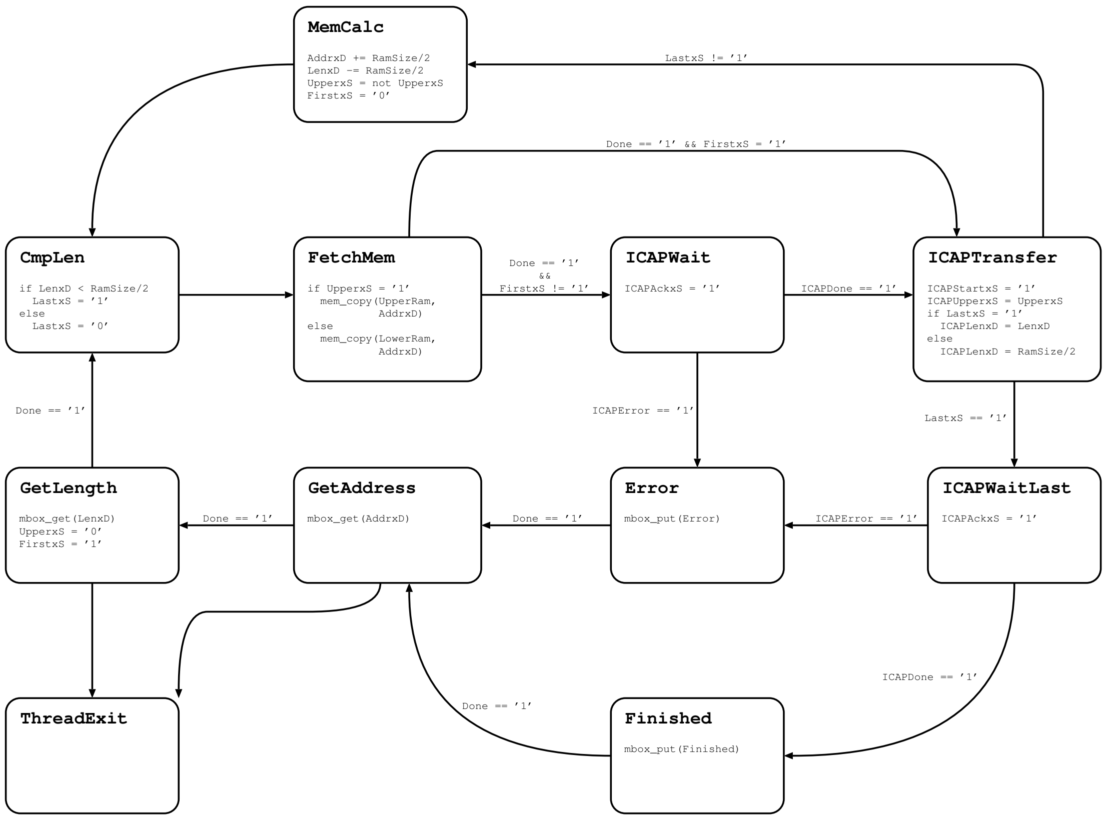


Figure E.3.: State Machine of HWT_ICAP for a Write.

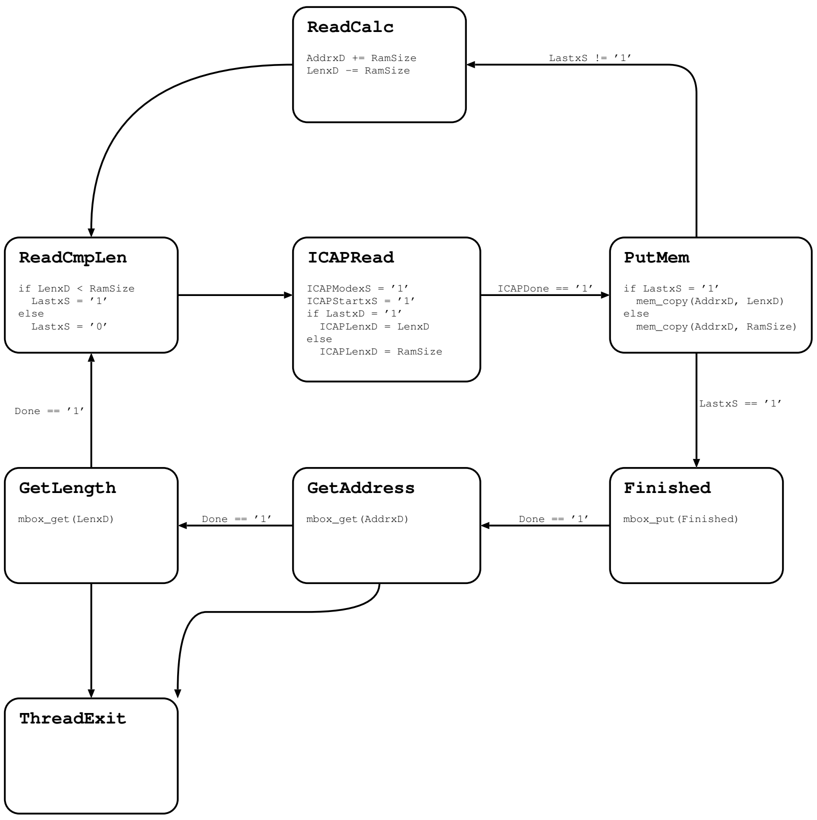


Figure E.4.: State Machine of HWT_ICAP for a Readback.

Bibliography

[1] Xilinx. (2012, Jan.) Virtex-6 Family Overview. DS150 (v2.4) January 19, 2012. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[2] S. Hansen, D. Koch, and J. Torresen, "High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, May 2011, pp. 174–180.

[3] Xilinx. (2013, Nov.) Virtex-6 FPGA Configuration - User Guide. UG360 (v3.7) November 27, 2013. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug360.pdf

[4] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An Operating System Approach for Reconfigurable Computing," *Micro, IEEE*, vol. 34, no. 1, pp. 60–71, Jan 2014.

[5] J. Delorme, A. Nafkha, P. Leray, and C. Moy, "New OPBHWICAP Interface for Realtime Partial Reconfiguration of FPGA," in *Reconfigurable Computing and FPGAs, 2009. ReConFig '09. International Conference on*, Dec 2009, pp. 386–391.

[6] S. Liu, R. N. Pittman, and A. Forin, "Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2010, pp. 292–292.

[7] R. Bonamy, H.-M. Pham, S. Pillement, and D. Chillet, "UPaRC - Ultra-fast power-aware reconfiguration controller," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, March 2012, pp. 1373–1378.

[8] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time Partial Reconfiguration speed investigation and architectural design space exploration," in *Field Programmable*

Logic and Applications, 2009. FPL 2009. International Conference on, Aug 2009, pp. 498–502.

[9] K. Jozwik, H. Tomiyama, S. Honda, and H. Takada, "A novel mechanism for effective hardware task preemption in dynamically reconfigurable systems," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, Aug 2010, pp. 352–355.

[10] A. Morales-Villanueva and A. Gordon-Ross, "HTR: On-Chip Hardware Task Relocation for Partially Reconfigurable FPGAs," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, 2013, pp. 185–196.

[11] Xilinx. (2013, Apr.) Partial Reconfiguration User Guide. UG702 (v14.5) April 26, 2013. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug702.pdf

[12] ——. (2013, Oct.) Command Line Tools User Guide. UG628 (v14.7) October 2, 2013. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/devref.pdf

[13] H. Kaeslin, *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication.* Cambridge University Press, Apr. 2008.

[14] Xilinx. (2014, Jan.) Virtex-6 FPGA Clocking Resources. UG362 (v2.5) January 24, 2014. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug362.pdf