Remote Reconfiguration for hardware-accelerated integrated Circuits

M Nagler\textsuperscript{1}, G G Kotre\textsuperscript{2}, W Hardt\textsuperscript{1}
\textsuperscript{1}Computer Engineering Department, Chemnitz University of Technology, Chemnitz, Germany
\textsuperscript{2}Electrical Engineering Department, Chemnitz University of Technology, Chemnitz, Germany
E-mail: nagm@hrz.tu-chemnitz.de

Abstract. When it comes to hardware-accelerated solutions, Field Programmable Gate Arrays (FPGA) are often used to have a good trade-off between performance and the conservative cost restrictions the systems FPGAs are used in commonly have. With reconfiguration or partial reconfiguration the amount of needed resources can be reduced. In this work we show a solution to change the functionality of the system remotely. This means no physical connection is required. The different configurations can either be stored on the board in advance or can be sent to it just in time.

1. Introduction
Embedded systems in current world scenarios have to meet very high standards of performance and other requirements. The challenge is to get the needed computational resources while satisfying strict cost restrictions, which gets more complicated over time due to increasing consumer demands. This applies especially for automotive engineering, aircraft construction, and space industry. Systems in these domains are very complex, so they have to be separated into different subsystems [1–3].

In the field of autonomous driving or unmanned aerial vehicles, the cost requirements, especially regarding size, weight, energy consumption, and price are very conservative. On the other hand these systems need a high reliability as well as real-time capabilities. To be able to meet all these requirements Field Programmable Gate Arrays (FPGA) are used more often in the last years as current chips available on the market provide more functionalities, peripherals and essentially faster processing speeds as compared to their older counterparts. In contrast to software-based solutions they also provide a higher performance while consuming less energy. On the other side they have a higher flexibility than application specific integrated circuits.

For autonomous vehicles the situation the system is currently in, has a high impact on the needed functions [2, 4]. E.g. a car only needs the parking assistant while driving within a city whereas the lane assistant is only needed while driving on rural roads or highways. The same applies for aerial vehicles where the flight control has to be more precise during take-off and landing whereas the system for evaluating the sensor data needs more resources while flying over the target area (e.g. to find the important points in the target area). With this, the available resources can be shared during the runtime of the system.

The flexibility to share or re-allocate resources during runtime is also beneficial in the scenario described in [5]. Here a distributed system, which consists of several computational nodes, is to be developed. On one hand it tries to satisfy the conservative cost restrictions, that autonomous embedded
In contrast to regular solutions there are no redundant resources available that can be used in case of a failure. This full utilization of the system resources puts special requirements to the system when a computational node fails. In this case the Quality of Service of the application running on the system will be reduced. By e.g. reducing the resolution or the frame rate of an image processing application, resources can be freed up so the application will fit again into the reduced amount of available resources. This is achieved by having each module of the application available in different versions with a different Quality of Service which will be used for re-allocation. To be able share or to re-allocate resources, the system needs to change its functionality based on the current situation. In software-based solutions the functionality of a system can be changed by unloading not needed functions and replacing them with different ones. In FPGA-based systems the functionality of a system cannot be changed that easily. There is only the possibility to reprogram the complete chip with a different configuration or, if it is supported by the chip, reprogram a part of the chip while the remaining chip stays untouched. The latter one will not require to restart the chip. In general both cases need physical access to the chip to load the configuration via a programming interface. Nevertheless, there is also the possibility to write the configuration into a nonvolatile memory where it can be loaded from on system startup. In this paper we show a solution to change the functionality of an FPGA remotely, without the need of physical access to the system. Different configurations will be stored inside the non-volatile memory and can be loaded dynamically during runtime. For this an internal configuration port is used to reboot the FPGA with one of the pre-stored configurations. In case a configuration is needed, that was not placed in the memory in advance, the configuration can be sent to the FPGA and a configuration slot in the memory is overwritten so that the new configuration can be loaded. The structure of this paper is as follows. In section 2 solutions for reconfiguration in a software- and hardware-based systems are discussed. Section 3 describes the conceptual system setup as well as the different functionalities the proposed solution offers. Section 4 covers the technical realization of the system, its distribution into different modules and their division into subroutines, that are needed for the different functionalities. Section 5 evaluates the proposed solution. Next to the resource consumption of the different modules, measurements of the required time for the different operations are discussed. Section 6 concludes this paper with a short summary followed by the next steps which will be done to further improve the proposed solution.

2. State of the Art

System reconfiguration has to be done, based on the used components, either in software or hardware. This section provides a short overview about possible solutions for software and hardware-based systems.

2.1. Software-based System Reconfiguration

To change the functionality of a software-based system, not needed functions have to be unloaded to free up resources. These resources can be used to load new functions that suit to the current situation of the system. In general this is handled by the operating system. The challenge is to find an appropriate mechanism to select the necessary functions based on the inputs of the system.

Hanti proposes in his work [2] a solution that is based on a hierarchical scheduling algorithm for multi-core processors (HAMS). The HAMS scheduler has two abstraction levels. The first level assigns the functions to the different cores of the processor and the second level schedules the functions for every core independently. To find a suitable scheduling, all use cases for every situation are calculated apriori and stored in a so called Knowledge Base. With this, always the best scheduling can be selected, based on the current situation of the system.

Another approach are Expert Systems. One of such systems is discussed in [6]. It uses pre-defined rules to switch between different system states and draws decisions in real-time. The behavior is
similar to a finite state machine. In contrast to [2] not all cases have to be specified in advance, so this rulebased decision making algorithm can react dynamically on the current situation.

2.2. Hardware-based System Reconfiguration

In hardware-based systems the change of their functionality has to be done by changing the configuration of the chip. Depending on the used device family, this can only be done completely (with rebooting the device) or it can also be done partially (without reboot). When targeting Xilinx FPGAs, not all devices support this so called partial reconfiguration. Looking at FPGAs from the 6th generation, there is only support for the Virtex family, not for FPGAs from the Spartan family. The same applies for FPGAs from other companies like Intel (formerly Altera).

Although officially not supported, there is research going on that provides partial reconfiguration for unsupported devices. References [7] and [8] propose a design flow that can generate partial bitstreams for Spartan 6 FPGAs from Xilinx. Here a new tool, called GoAhead, is introduced. It replaces the PlanAhead tool from the Xilinx toolchain. Based on userdefined areas, this tool removes everything from the complete bitstream, that should not be changed. With this a partial bitstream, that can be used to flash the FPGA, is generated.

To get access to the configuration memory, there are primitives available on the FPGAs. For Intel devices they are called Partial Reconfiguration IP (PR-IP) [9] and Xilinx has the so called Internal Configuration Access Port (ICAP) [9, 10]. Based on the used device, the functionality of these primitives is different. Devices that support partial reconfiguration can use it to update the configuration directly. If partial reconfiguration is not supported, it can only be used to reboot the FPGA with one of the configurations that is stored in a non-volatile memory. The selection of the according configuration is done by providing the start address of it.

The non-volatile memories can either be connected to a serial or a parallel port. A Nexys 3 board from Digilent [11] with a Spartan 6 FPGA on it, shall be shown as an example. This board contains three external memories, while two of them are non-volatile phase-change memories (PCM) [12]. Both have a size of 16 Mbyte. The serial memory is connected to a quadmode SPI bus whereas the parallel memory is connected to a 16-bit bus that only supports 16-bit operations. The latter is divided into 128 individually erasable 128 Kbyte blocks where one of these blocks is sub-divided into four individually erasable 32 Kbyte sub-blocks.

3. System Setup

The solution proposed in this work shall be realized on the previously mentioned Nexys 3 board from Digilent. As the Spartan 6 FPGA on it does not support partial reconfiguration, the method to reboot the board with a configuration that was stored in the non-volatile memory is used. This configuration can either be placed in the memory in advance or it can be sent to the board during runtime. The communication channel that is used to send a new configuration to the board or to trigger an reconfiguration shall be arbitrary, so an easy to use interface has to be designed, that can be adapted easily to different communication channels.

Furthermore, the proposed solution shall be as generic as possible, so there is only a low overhead to port it to a different FPGA with a different memory. To achieve this, all functionalities will be placed in separate modules; the System Control will handle the communication channel, extract the received data, and forward it to the according sub-components to call the different commands these components implement. The subcomponents will be the Configuration Select module, that can be used to reboot the chip with a different configuration, and the Memory Controller module, that can erase and write the non-volatile memory.

3.1. Memory Controller

This module can handle the data that is stored in the memory and will access the parallel memory. Based on the size of a configuration bitstream, the amount of possible configurations in the memory can be determined. A configuration for the Spartan 6 FPGA used on the Nexys 3 board has a size of
approximately 453 Kbyte. To be able to exchange a configuration inside the memory, the according part has to be erased first. As this is done blockwise, it has to be ensured that two different configurations are not stored in the same block. This requires that every configuration starts at a new block. We can calculate the amount of blocks per configuration by dividing the size of a configuration by the size of a block:

$$blocks = \frac{453\text{Kbyte}}{128\text{Kbyte}} = 3.54$$

This means every configuration will need four blocks in the memory, which is equivalent to 512 Kbyte. By dividing the size of the memory with this value, we get the amount of configurations, that can be stored in the memory:

$$configs = \frac{16\text{Mbyte}}{512\text{Kbyte}}$$
$$= \frac{16384\text{Kbyte}}{512\text{Kbyte}}$$
$$= 32$$

Based on this the memory can store up to 32 configurations. One of these configurations is reserved for the design that is developed in this work and must not be overwritten. This design is placed in Slot 0 and will be called the Golden Image. It is the default configuration of the FPGA that is loaded on power-on. As Figure 1 shows, Slot 0 covers Block 0 to Block 3. After this, the user configurable slots follow. As the last slot would use Block 127 which is sub-divided into four 32 Kbyte sub-blocks, it cannot be used here and leaves Block 124 to Block 127 unused. This means in total there are 31 slots in the memory for different configurations where 30 of them can be used for user designs.

The following commands will be implemented by this component:

- Write Data – write one 16-bit word
- Write Block – write up to 32 16-bit words
- Read Data – read one 16-bit word
- Read Status Register – read status register of memory
- Clear Status Register – reset error flags in status reg.
- Block Lock Status – check if block is locked for writing
- Unlock Block – unlock block for writing
- Erase Block – erase data in block

3.2. Configuration Select

This module can trigger the reboot of the chip with a different configuration. To do this, the Internal Configuration Access Port (ICAP) [10] primitive is used to issue a so called IPROG command. The instruction sequence is described in [10]. As input the module gets the number of the configuration that shall be loaded. Based on the size of a configuration the address in the memory, that has to be provided to the ICAP interface, is calculated.

The following command will be implemented by this component:

- Reboot – load a different configuration

3.3. System Control

This module is the main control unit in this work. It integrates the communication channel and handles the two sub-components Memory Controller and Configuration Select. The received operations will be converted into commands, the sub-components implement.
The complete system structure as well as the memory division is illustrated in Figure 1. The Golden Image is placed in Slot 0. The other slots can be used for configurations, that contain user logic. All configurations need to implement the Configuration Select module as well as a connection to a communication channel to be able to change the functionality of the system remotely. As the Configuration Select module is realized as independent component, it can be easily integrated into the user logic.

4. Implementation
The different components described in section 3 are realized as separate modules using the hardware description language VHDL. To show their functionality an example design with a serial interface as communication channel is implemented.

4.1. Memory Controller
This component is implemented as a finite state machine (FSM). The challenge with this FSM was the fact, that there is no clock signal, which can be used to synchronize the FSM with the memory. The memory latches the data on a rising edge of the according control signals. To implement this behavior, all states to write instructions or data were doubled so a rising edge could be modeled during the transition of these states. Nevertheless, the timing restrictions do not allow clock rates higher than 20 MHz.

The commands that were discussed in section 3 were implemented accordingly. Table 1 lists the input and output for every command. The start for every command will be triggered by the System Control module. After the completion of the command, a finish signal will be emitted.

| Command          | Input          | Output       |
|------------------|----------------|--------------|
| Write Data       | Address, Data  | –            |
| Write Block      | Address, Data(N) | –           |
| Read Data        | Address        | Data         |
| Read Status Register | –           | Status Register |
| Clear Status Register | –           | –            |
| Block Lock Status | Block Base Address | Lock Status |
| Unlock Block     | Block Base Address | –          |
| Erase Block      | Block Base Address | –          |

4.2. Configuration Select
This module was implemented as FSM as well by realizing the instruction sequence mentioned in [10]. It uses the same clock frequency of 20 MHz as the Memory Controller. The input for the reboot command is the slot number in the memory, that shall be loaded. Generics for the memory size, the size of a block, and the size of the configuration bitstream are used to calculate the start address of the slot. This makes the module reusable for different FPGAs.
4.3. System Control
This module triggers the different commands of its submodules. For this it also implements a FSM that starts a command and waits until its finish signal is emitted. Furthermore, it handles the data writing process. Internal count registers are used to keep track on how much data was already written or how much blocks were already erased. This module uses the same generic values as the Configuration Select module to calculate the addresses accordingly.

4.4. Example Design
An example design was used to show the functionality of all implemented modules. It integrates the System Control together with its sub-modules Memory Controller and Configuration Control. The communication channel was realized using a serial interface. Furthermore a script that converts a configuration bitstream into a binary file, which can be written to the non-volatile memory, was realized.

5. Results
The example design could show the proper functionality of the implemented modules. The generic address calculation of the Configuration Select and the Memory Controller module were able to select
the different slots in the memory correctly. With this, any configuration could be changed by erasing it and writing a new bitstream to its memory location. After selecting a new configuration and rebooting the FPGA, it was loaded successfully. Additionally, the memory content written with the developed solution was compared to the memory content written with the Digilent Adept tool, that is provided by the manufacturer, to verify the correctness of the writing process.

Furthermore, the performance of the Memory Controller module was evaluated by measuring the time to complete the different operations. This was done by counting the clock cycles it took to perform an operation. The operation was applied to the complete slot. With the clock frequency of 20 MHz the elapsed time was calculated. The results show, that the time depends on the data that is stored in the memory or the data that is to be written. Two methods were tested for writing: wordwise and blockwise writing. The measurement was done with four different values:

- Zero – writing/erasing the value 0x0000
- One – writing/erasing the value 0xFFFF
- Count – writing/erasing an up-counting value
- Alternating – writing/erasing a value where consecutive bits have a different value: 0x5555

The results are shown in table 2. Every test was repeated 10 times. The table shows averaged values. As expected, the blockwise writing is much faster than wordwise writing. It outperforms wordwise writing by a factor of 11 to 19 (50 for 0xFFFF). Furthermore, it can be seen that writing and erasing 0xFFFF is much faster than the other operations. The reason here is the way the memory works. All entries have the value 0xFFFF after erasing. With this the controller of the memory chip does not have to change the value when writing or erasing, which results in a faster execution of the according operation.

Table 2. Performance of Controller Commands (Time to erase 4 Blocks or Time to write a Configuration of 453 Kbyte)

| Value      | Erase (ms) | Write Word (ms) | Write Block (ms) |
|------------|------------|-----------------|------------------|
| 0x0000     | 1634       | 5025            | 442              |
| 0xFFFF     | 113        | 2240            | 44               |
| "Count"    | 1726       | 5029            | 313              |
| 0x5555     | 1729       | 5032            | 268              |

Additionally, the resource consumption of the design was evaluated. The Golden Image uses less than <5% of the FPGA resources. Considering that the Spartan 6 on the Nexys 3 board is a comparatively small chip from its device family, the resource consumption can be considered as very low. This makes it possible to integrate the complete design into the user configurations, not only the Configuration Select. If there are not enough resources available to integrate the complete design, at least the Configuration Select module has to be integrated to support the remote reconfiguration. Without System Control and Memory Controller, this module consumes approximately 1% of the FPGA resources and will fit in nearly every user design.

6. Conclusion and Future Work
In this work, a method to remotely reconfigure hardware-accelerated integrated circuits was shown. The design was realized for a Spartan 6 FPGA mounted on a Nexys 3 board from Digilent. With the independent modules, the implementation can be ported easily to different boards and memory chips. The low resource consumption makes it possible to integrate this solution into any design. The time to
erase a configuration and write a new one into the memory is comparatively high with approximately 2000 ms. But with the low resource consumption of the proposed solution, the complete design can be integrated into the user designs, too. With this, it is possible to replace a configuration during runtime of a user design.

To further improve the proposed solution, the memory usage shall be optimized. Currently every slot uses with 3.54 out of 4 blocks only 88.5% of its size. Additionally, Block 124 to Block 127 at the end of the memory is unused, too. With this, approximately 2337 Kbyte (14%) of the complete memory is unused. By optimizing the memory usage up to 5 more configurations can fit into it.

Furthermore the here proposed solution shall be tested with the partial bitstreams generated by the GoAhead tool proposed in [7] and [8].

Also the time it takes to reboot into a different configuration has to be evaluated in the future. As this needs additional external logic, it was not considered in this work. Finally, the design proposed in this work will be integrated into the system discussed in [5] to be able to react on unforeseen situations as they can occur in autonomous distributed embedded systems.

References

[1] Watkins C B, Walter R 2007 Proceefing of the 26th Digital Avionics Systems Conference 2.A.1–1–2.A.1–10
[2] Hanti T, Frey A, Hardt W 2015 Proceefing of the 34th Digital Avionics Systems Conference (DASC) 7A3–1–7A3–11
[3] Moens H, Famaey J, Latre S, Dhoedt B, De Turck F 2011 Proceefing of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops 137–144
[4] Schneider S, Meisel A, Hardt W Proceefing of the 2009 IEEE/IFIP International Symposium on Rapid System Prototyping 61–67
[5] Nagler M, Blokzyl S, Hardt W 2017 Proceefing of the IEEE Sensors Applications Symposium 85–87
[6] Halbig J, Windisch A, Kingsbury P, Oswald N, Hardt W 2012 Autonomous Unmanned Systems 119–128 doi: 10.1007/978-3-642-32217-4_13
[7] Koch D, Beckhoff C, Tørrison J 2010 Proceefing of the International Conference on Field-Programmable Technology 361–364
[8] Beckhoff C, Koch D, Torresen J 2012 Proceefing of the 20th International Symposium on FieldProgrammable Custom Computing Machines 37–44
[9] Vipin K, Fahmy SA. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications. doi:10.1145/3193827g.
[10] Spartan-6 FPGA Configuration User Guide. 2100 Logic Drive, San Jose, CA 95124; 2019.
[11] Nexys 3 FPGA Board Reference Manual. 1300 Henley Court Pullman, WA 99163; 2016.
[12] 128Mb: P8P Parallel Phase Change Memory (PCM); 2012