Hardware context switching on FPGAs

Sindre Georgsen

Department of Physics
University of Oslo

31st March 2014

Abstract

FPGAs have the ability to replace its hardware modules without disrupting the execution of the other modules of the system. This is called *dynamic partial reconfiguration*. The capability to dynamically replace a hardware module with another, while the rest of the system is still running, could have a major impact on a design in such way that a smaller device can be used to implement a great and complex system. The reconfiguration attribute would become even more advantageous if the modules were allowed to *pause* so that the module didn't have to start at the beginning each time it was reintroduced to the system, it could resume from where it was, when replaced. This feature of partial reconfiguration is called *context switching*. Context switching must be able to preserve a modules internal state and its memory contents in order for the module to continue from where it was, in the execution time-domain, when reintroduced.

A challenge of the partial reconfiguration property is that it is time consuming, especially if frequent context switching is taking place. In this thesis, a bitstream manipulator module is implemented in *hardware* to decrease the time consumption of context switching. The module creates a new partial configuration file based on the initial configuration file, readback file of the captured flip-flop values, and the masking file generated by the Xilinx tool ISE. An external memory is the module repository, which holds all necessary and generated files for each partial module. It is this methodology that decreases the time consumption of context switching.

Simulation shows that the system performance of the process of creating a new partial configuration file is *three orders of magnitude* faster compared to the most relevant research work. The creation of a new partial configuration file is measured to 5.12 µs for a file size of 1 kB (1024 bytes), and 47.7 µs for a file size of 15 kB. The results are promising, and further investigation will probably result in a step closer to realizing a satisfactory partial reconfigurable system that includes context switching.

Preface

When choosing a master project I had two options, either contact companies in the electronic industry, and ask if they had any projects suited for a master student, or I could choose a project given by a research group at the university. I checked for available projects at the university, and I found a few projects related to FPGA and reconfiguration. I read the description of the projects and I was immediately fascinated by the one titled 'High speed context switching on FPGAs'. My supervisor warned me that the project might be comprehensive, but I enjoy a challenge very much so I chose the project, and began enthusiastically.

A Spartan-6 device was chosen because of its relatively small architecture, and it is a low-cost device. I spent couple of weeks searching for articles and documentation on the subject of partial reconfiguration and context switching, and another two weeks trying to figure out the composition of the configuration file; which bit in the configuration file corresponds to which flip-flop in the device. I discovered that the manufacturer of the FPGA device is not so eager to share information about the Spartan-6 configuration. My supervisor and I made a choice; Spartan-6 was unsuited for the tasks at hand due to lack of documentation and the limited time span of the project. Virtex-6 suited better because of its capability to perform a capture of registers.

Most previous work on the subject so far has used an embedded CPU to control and to initiate the reconfiguration and context switching process. I've already had an idea of how the system should work; using external memory and make the bitstream manipulation module interact directly with the memory, *without* the use of a CPU. This approach would make the context switching process go faster.

Although Xilinx provides an IP core for controlling the external DDR3 memory, the documentation for it was not that impressive, some information was directly contradicting, and some information was not related to the current IP version. Due to poor documentation I spent a considerable amount of time going through forum threads searching for answers. It took me about two months to finally achieve a working memory controller that suited my needs. I was now about half way into the project with regards to time, and I realized that I had to moderate my goals, and even abandon a goal in order to produce some results. I focused on bitstream manipulation since that was in fact the main goal of the project. Implementation of a partial reconfigurable region is therefore not included, and it is assumed that is available. By doing these cutbacks I managed to generate some valuable results.

I would like to thank my supervisor, Professor Jim Tørresen, for being critical and for asking questions that challenged me. I would also thank PhD student Alexander Wold for giving me assistance when I desperately needed it.

Contents

List of figures

List of tables

1 Introduction

Field programmable gate array (FPGA) is a highly flexible integrated circuit that is programmable in the field. With an FPGA, complex digital circuits can be realized, and true parallelism is one of its many strengths. Another feature is *partial reconfiguration* (PR); meaning that parts of the FPGA can be reprogrammed, or reconfigured, in the field while the rest of the FPGA is running. The same reconfigurable area can be reconfigured many times over and that enables timesharing of hardware resources, i.e. two or more digital circuits can share the same area, on the FPGA, over time.

PR can be used in many different ways. For instance, a design may operate with several modes where each mode may demand a specialized module, and when that mode is activated the partial reconfigurable area is configured with the required module. Another possibility is when a design has more than one start-up configuration and a selector determines which configuration should be used. A third possibility is that two or more modules continuously switch turns running on the partial reconfigurable region (PRR). The currently running module can then be *paused* before it is swapped with another module. Pausing a module means that the modules internal states, i.e. its register values, are stored. The stored information can then be retrieved when the modules is swapped back in, so that the module does not have to begin from the start again. This method is called *context switching*. A more comprehensive discussion on context switching can be found in chapter 2.5.

Although PR has been available for many years, it has not yet been adopted by the industry on a large scale. A major contributing factor to its unfavorable status is that the process of reconfiguring the targeted area on the FPGA is time consuming and may be cumbersome to implement. The speed of partial reconfiguration is in many cases the essential factor, but there may be situations where the switching frequency is of less importance as exemplified in the 'mode' example above. However, there are benefits associated with PR and context switching. By sharing hardware resources, a smaller device can be used, and hence reduce the cost, mutual exclusive modules may be implemented on the same device, among others.

There has been extensive research on the subject in academia for several years and it has progressed a lot since it was first introduced, see chapter 3.2 for the most relevant research related to partial reconfiguration in general and context switching in particular. There are two main methods of partial reconfiguration that includes context switching. One is to use a soft core embedded microprocessor like the MicroBlaze and write software that will perform all operations needed. This is the most widely used approach. The other is to design a hardware version *without the use of a microprocessor*. The latter is the approach chosen for this thesis on context switching. This choice was made for the following two reasons. First, the hardware approach has not, to my knowledge, been investigated thoroughly enough and I think there is a potential for reducing the overall time consumption of the reconfiguration process compared to the software approach. Secondly, I wanted to gain experience in

VHDL, and to keep the abstraction level as low as possible in order to gain a greater understanding of an FPGA.

1.1 Scope of the thesis

Three main goals were given in the thesis description, and are as follows:

1) Design a hardware module for filtering out the state out of configuration readback data from an FPGA.
2) Development of techniques for setting back a modules state.
3) Design and implementation of a demo system.

The first two goals must be defined more precisely in order to keep the tasks at hand as simple as possible due to limited duration of the thesis. They are also intertwined in such way that the second is dependent on the first. The first goal is, in here, defined to perform a capture of the internal registers of the partial reconfigurable module (PRM) at that point in time when it is ready for replacement, determined by a context switching main controller (CSMC). The captured data is then read back from the FPGAs internal configuration memory and stored in an external memory. The second goal is defined as constructing a *new* configuration file that contains the register information from the previously stored readback data. Both tasks include accessing an external memory for reading and storing of said data.

The third goal was unfortunately not reached. It is therefore assumed that the PRR has already been implemented, and that two PRMs are available. Further, all measurements given in chapter 5.4 are based on simulations rather than actual results from a demo system. These measurements are then extrapolated in order to give an accurate estimation beyond simulation results.

One issue regarding partial reconfiguration is security. To limit the extent of the thesis, this subject is not part of the thesis. See [1], [2] and [3] for a discussion on the security aspect.

1.2 Chapter overview

Chapter 2 Background and theory

This chapter starts with a short description on the architecture of an FPGA. In section 2.3 a brief explanation of FPGA configuration is given. The chapter continues with the theory behind partial reconfiguration in section 2.4, and context switching in section 2.5. The final section goes deeper into configuration of a Virtex-6 FPGA with a description on configuration frames, packets and registers. This section also gives an example on how to decode a bitstream.

Chapter 3 Relevant research

As mentioned in the introduction there has been extensive research on the subject of partial reconfiguration, but not as much on the issue of context switching. The most relevant work, to my

knowledge, compared to this thesis is presented here. They are listed in order of increasing relevance.

Chapter 4 Implementation

This chapter first explains the concept of the system that has been designed for this master thesis. Then follows an overview of the system, in section 4.3, where a functional explanation is given. The chapter continues to give a short description of the hardware platform used in the project. In section 4.5 the implemented system is described, the section gives an overview on the data flow, and an operational description is given. In the final section a detailed description on the implemented modules is given. Simulation captures and excerpt of the code is given when it is necessary to highlight a problem or to illustrate how they were solved.

Chapter 5 Simulation and measurements

This chapter gives a description of the simulation environment and how the simulation was performed. Section 5.2 gives a detailed description on the set-up of the simulation. Section 5.3 takes a closer look at how the DMC and DDR3 memory responds to read requests from the BMM. Measurements and results are given in section 5.4, this section also includes some equation needed to calculate parameters related to configuration frames and timing. The results are plotted in a graph and data extrapolation is used to give a precise timing estimate beyond simulation. The final section compares the simulated results width previously conducted experiments from other relevant work mentioned in chapter 3.2.

Chapter 6 Conclusion and future work

This chapter provides a conclusion of the work presented in the thesis. It also gives a few suggestions on how to improve the system in order to increase its functionality and effectiveness.

2 Background and theory

2.1 Chapter overview

This chapter starts with a short description on the architecture of an FPGA. In section 2.3 a brief explanation of FPGA configuration is given. The chapter continues with the theory behind partial reconfiguration in section 2.4, and context switching in section 2.5. The final section goes deeper into configuration of a Virtex-6 FPGA with a description on configuration frames, packets and registers. This section also gives an example on how to decode a bitstream.

2.2 Field programmable gate array

An FPGA consists of a matrix of programmable logic blocks which can be connected together through programmable interconnects. A network of wires, both horizontally and vertically, are situated between the logic blocks, see Figure 2.1. In addition, there are programmable input and outputs used for communication with external components. This makes the FPGA highly flexible and complex digital circuits can be realized. Bitcoin miner, software defined radio, advanced digital filters, video and image processing, and high performance parallel computing are some few examples.

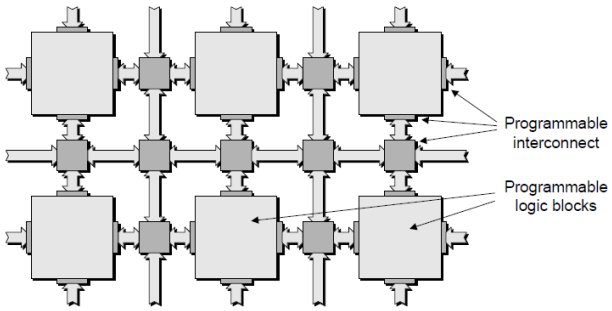


Figure 2.1 - Simplified architecture of an FPGA [4]

In addition to the logic blocks and the interconnects, an FPGA may contain dedicated embedded circuits such as digital signal processors (DSP), block RAM (BRAM) and even a hard core microprocessor, as illustrated in figure 2.2 This makes the FPGA highly versatile and powerful.

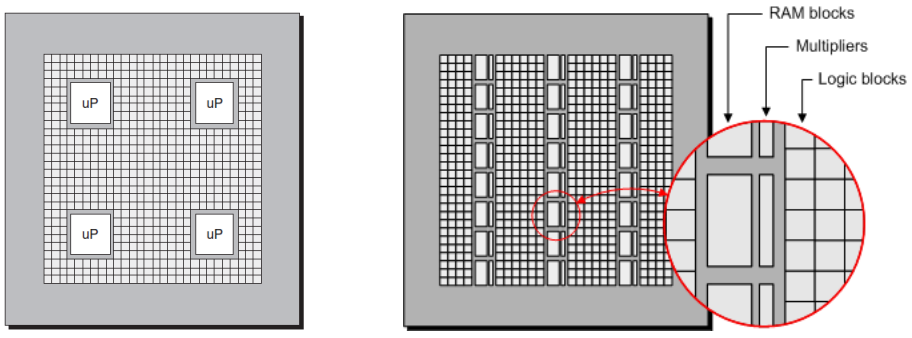


Figure 2.2 - Embedded hard core CPU (a), and Embedded RAM and DSP (b) **[4]**

Each logic block, or configurable logic block (CLB) as Xilinx calls them, consists of slices and each slice contains logic cells, which comprises flip-flops and look-up tables (LUT), usually twice as many flip-flops as LUTs. Most modern FPGA devices are static RAM (SRAM) based; the LUTs are SRAM cells that hold the desired logical values corresponding to the implemented function, see figure 2.3.

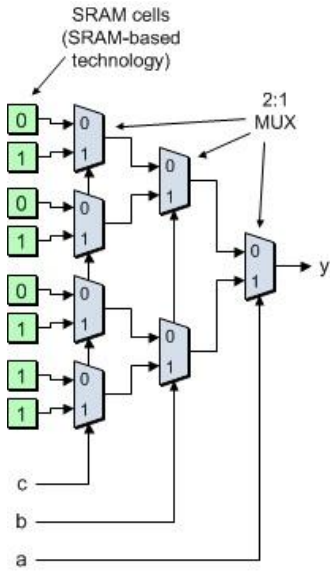


Figure 2.3 - SRAM based LUT.

There are different types of slices, each with different complexity and functionality. For the Xilinx Virtex-6 device used in this thesis, there are only two slice types, called SLICEL and SLICEM. The first one, SLICEL, is the simplest one with four LUTs and eight flip-flops. Additionally, it has the ability for a carry signal to and from adjacent slices. SLICEM have the same functionality as a SLICEL but in addition the LUTs may be used as distributed RAM (256-bit) or as shift registers (128-bit). Each LUT in the Virtex-6 device is a six-input LUT which can be split into two five-input LUTs [5].

A numerous, sometimes thousands, CLBs are necessary to implement complex digital circuits and all these CLBs must be connected to each other. To accomplish this, there

are programmable interconnecting switch boxes adjacent to the CLBs. The configuration of these switch boxes determines which CLBs are connected to each other. To accommodate the clocking of the flip-flops there are dedicated clock paths organized in such way that each end point of the path has the same delay as all the others. This can be accomplished by routing the clock paths in an H-tree or similar, which gives a uniform clock distribution**.** Figure 2.4 shows an example of a simple clock distribution network where each endpoint, represented by the dots, has equal path length.

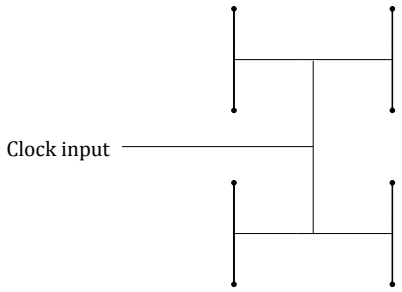


Figure 2.4 - Uniform clock distribution.

An FPGA is divided in clock regions, each region with its own dedicated clock distribution network. In the Virtex-6 device used in this thesis there are in total of 12 clock regions. There are 32 global clock lines that can reach the entire FPGA, in all of the clock regions. In addition, each clock region has up to six regional clock trees [6].

While modern computers have a central processing unit (CPU) that operates at frequencies up to approximately 3.5 GHz, the operation frequency of an FPGA design varies widely depending on the implemented design and the FPGA device. But it is far lower than the CPU frequency. In a computer, all internal components of the CPU are placed at a fixed location and the placement is optimized to minimize the internal routing in order to achieve the lowest possible path delay, i.e. highest possible frequency. This differs from an FPGA; although all CLBs, flip-flops, I/Os, embedded components and the internal routing network are at fixed positions on the die, the routing path between modules may become long due to the interconnecting switch boxes that connect CLBs and other components to each other. The longest path in a design can therefore vary and hence the frequency varies accordingly.

In order to load an FPGA device with a design, the device need to be configured, i.e. all SRAM cells need to obtain their logic state, all flip-flops must be configured with their initial state, all BRAM used must be initialized with their intended data. This is done by uploading a *configuration file*, also known as the *bitstream*, to the FPGA. The next section gives a detailed description of FPGA configuration.

2.3 Configuration

The configuration data is stored in CMOS configuration latches (CCLs) [7]. Since SRAMs and CCLs are volatile, i.e. its value are lost at power down, an FPGA must be configured each time the power is applied. It is therefore necessary to have a memory

device connected to the FPGA that holds the configuration file(s) so the FPGA can be configured at power up in the field. The configuration file, or bitstream, consists of several commands in addition to the configuration data. The commands control the configuration process. A detailed description of the relevant commands can be seen in section 2.6.4. The configuration file is produced by the integrated development environment (IDE) tool and has the file extension of .BIT or .BIN.

File size of the ML605 board Virtex-6 device configuration file is ~70 Mbit (~8.8 MB) according to [7]. When a configuration is complete the configuration interface on the FPGA side asserts a DONE signal, see section 2.6.1, indicating the configuration has completed with success. The configuration process of a Xilinx Virtex-6 device undergoes a specific configuration sequence with two phases; setup and bitstream loading. Figure 2.5 shows the sequence and its steps. The setup phase consists of three steps. The first step is to check availability of voltage power required by the device. The second step is to clear the configuration memory. Final step of the setup phase is to sample the mode pins that determines the configuration interface, the configuration clock (CCLK) also starts at this step. See table 4.1 for mode pin settings.

Next phase is the bitstream loading which consists of four steps. First step is synchronization. A 32-bit synchronization word in the bitstream is detected and tells the configuration logic that a configuration is about to commence. Before the synchronization word there is a bus width detection pattern that tells the configuration logic the bus width of the configuration data. Any data in the configuration file before the bus width pattern is ignored. The synchronization word is `0xAA995566`. Next step is to do a check of device ID to prevent configuration with a bitstream that is formatted for another device. After the device ID check configuration commands and data are uploaded to the device. Final step of the bitstream loading phase is a cyclic redundancy check (CRC). As the configuration data is uploaded the device calculates a CRC value. At the end of the bitstream a CRC command is issued followed by a CRC expected value. If the calculated value does not match the expected value the configuration is aborted. After the bitstream loading phase the bitstream instructs the device to perform the startup sequence, which is described in [8].

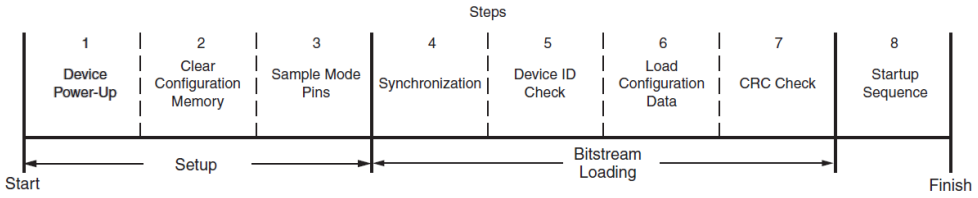


Figure 2.5 - Configuration sequence

2.4 Partial Reconfiguration

FPGAs are flexible in the sense they can be *reconfigured*, i.e. uploading another configuration file, on-site. To perform the reconfiguration the FPGA has to shut down in order to load a new configuration file. The whole FPGA is then reconfigured. *Partial reconfiguration* (PR) is the ability to reconfigure only parts of the. *Passive* PR means

all clocks, and hence the whole system, are suspended during reconfiguration. *Active* PR means the rest of the system is running while reconfiguration takes place. The latter is also known as 'run-time reconfiguration', 'dynamic partial reconfiguration' or as 'hardware multitasking'. In order to perform a partial reconfiguration it is necessary to partition the FPGA fabric into a static region and a partial region. The partial region can be reconfigured while the design in the static region is running. The partial region may take any size of the fabric. Partial reconfiguration enables three dimensional hardware space where different designs share the same area over time. Some benefits of partial reconfiguration are listed below.

- Implementation of a bigger system onto a smaller FPGA device if the system can be divided into smaller independent designs.
- Improved fault tolerance with the ability to reconfigure the damaged module
- Mutual exclusive designs may use the same device
- Faster startup time as only the static region needs to start first. The PR region with its modules may start at a later time
- Custom CPU instruction for a soft core microprocessor may be implemented when necessary [9]

A challenge with partial reconfiguration is that it takes a considerable amount of time to perform a reconfiguration. The bigger the partial region that is to be reconfigured the longer time it takes to reconfigure.

2.4.1 Static region

The static region may contain modules that are needed throughout the entire cycle of operation of the system such as a soft core microprocessor, a reconfiguration controller or others. These modules will never be interrupted or stopped due to partial reconfigurations.

2.4.2 Partial region

There are three different *styles* of PRR. These styles have various complexities and the simplest one is a *single island style*. This style is just one island on the fabric surrounded by the static region as seen in figure 2.6. The size of the island may vary but cannot be lower than the height of one clock region, or a frame. With island style, internal fragmentation may become an issue. Internal fragmentation is the PRR's resources not used if the module has a smaller footprint than the size of the island itself. Only one PRM can be situated in an island at any given time. Multiple islands may be implemented on the same FPGA.

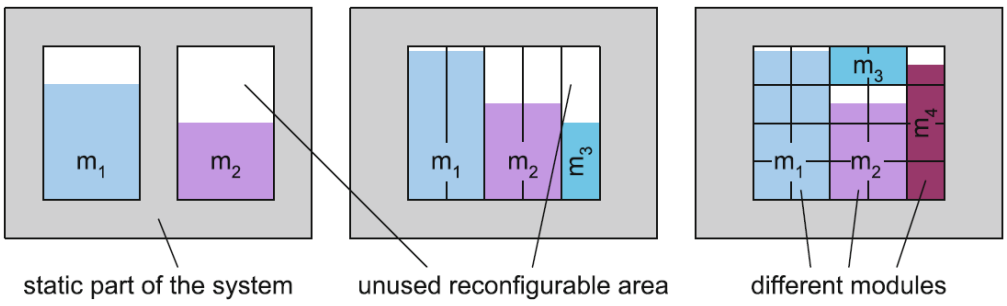


Figure 2.6 - Different styles of partial reconfigurable regions [10]

Slot style partial region may resemble the island style, but it is narrower as a slot may refer to a column or a frame. These slots may have greater height than the height of a frame, just as the island style. They may even span the whole height of the FPGA. The issue of fragmentation becomes less with a slot style region as a multiple slots may host one PRM and the utilization of slots then becomes higher compared to an island style region. On the other hand, slot style region may suffer from external fragmentation due to the fact that a narrow slot may not include all logic resources, i.e. BRAM, DSP or others. Depending on a modules requirements a slot may therefore become completely unused.

Grid style is the most complex style but gives the most flexibility. Each grid block is in its own a partial reconfiguration area. A partial module may have a foot print of several grid blocks but two modules cannot share the same block. So internal fragmentation still exists but is even less than in the slot style.

2.5 Context switching

One problem with partial reconfiguration is that whenever a module is reintroduced into the partial region it starts off from the beginning, i.e. the module's progress is not recorded so that it can be resumed from where it was when it was replaced. Context switching addresses this problem. As mentioned in the introduction, context switching is the capability to preserve a modules internal state for later resumption at that point in time where the module was replaced.

2.5.1 Context switching in software

A computer, with one CPU core, can apparently perform several tasks simultaneously, i.e. applications run in parallel. This is however not the case. The computer switches between processes at a timely fashion. To accomplish this, the CPU, in cooperation with the operating system, has to perform a context switch each time it switches process. Context switch means to save a process' state into memory, from here on out called context save (CS), and then restore those states when the process resumes, from here on out called context restore (CR). This enables multiple processes to share a single CPU. A context is defined to be the contents of a CPUs registers and program counter at any point of time [11]. For the CPU to know when to save the running process' state and which process to switch to there are different triggers. The running process can make itself trigger a switch, e.g. by timeout or by waiting for another process' flag, an interrupt may also trigger a switch,

2.5.2 Hardware context switching

Context switch in a hardware design is not that different from a software context switch. But what constitutes a modules context? In essence the only context available must be that information that is due to sequential logic, i.e. registers, and local RAM inside the module. Combinatorial logic cannot be preserved. Combinatorial circuits are realized with LUTs, multiplexers and latches. Information from these elements is not available unless they are registered through a flip-flop. It is therefore imperative that the design of a partial reconfiguration module includes registers wherever information needs to be passed on to the next run. The designer must also avoid inferring latches as output of a latch is not clocked. Before any CS can begin all clocks of that particular partial reconfiguration module must be halted to avoid saving of

invalid data. If the module uses more than one clock the module must be halted in between the cycles of the slowest clock. [12] lists other requirements. There are two ways to extract information from a module, by *hardware checkpoints* and by *configuration readback with register capture*. The latter is discussed in section 2.6.5 and hardware checkpointing is discussed in next section.

2.5.2.1 Hardware checkpointing

Hardware checkpointing is a way to store a modules state during normal operation, in other words an image capture of the module at run-time. There are different methods to accomplish checkpointing, three which are described in [13], were a definition is given as: *"A checkpoint is a set of data items representing the image of the last error-free state of a module of computation from which in case of the occurrence of a fault may be restored."* These data items are output (O), input (I), state (S), state transition function (S_f), output function (O_f) and initial state (S_0), half of which is necessary for checkpointing; O, I and S. A short summary of the different checkpointing techniques follows.

1) **Memory mapped state access**

   Memory mapped state access utilizes a memory space in a CPU to store a modules states. The present value is fed back to the module while executing a checkpoint. This is achieved by integrating a checkpoint flip-flop into a read/write memory of the CPU.

2) **Scan chain based state access**

   This technique is based on a long chain of flip-flops connected together as a shift register. A multiplexer in front of each flip-flop switches between normal operation and shift register. Chain ends are connected together to from a ring shift register.

3) **Shadow scan chain based state access**

   Duplicating all the flip-flops is another way to take a snapshot of the module. That is what shadow scan chain does. These duplicates are used for the checkpointing only and, as in scan chain, they are arranged in a ring. The overhead for this technique is tremendous as one flip-flop in the module demands another flip-flop for checkpointing. The advantage is that it only takes one clock cycle to perform the checkpointing.

Hardware checkpointing demands added hardware to the PRMs and hence adds logic overhead. Figure 2.7 illustrates the difference in both logic overhead and time overhead between the three hardware checkpointing techniques and compares them to configuration readback. L_{LUT} denotes extra LUTs in the PRM required for implementing the respective technique, and t_c denotes time overhead in terms of clock cycles. It is seen that for the extremes the shadow scan chain provides close to none overhead in time and close to 100% logic overhead. Readback is completely opposite, almost none logic overhead and a huge overhead in time. In between there are scan chain and memory map.

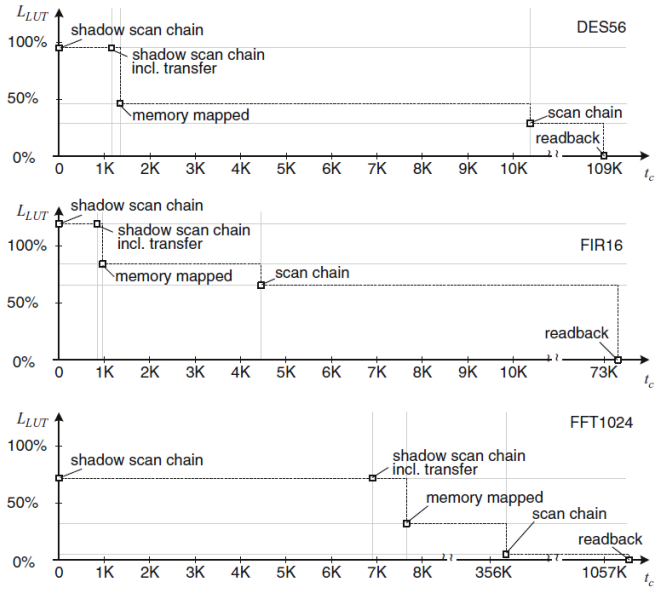


Figure 2.7 - Time and logic overhead for the different hardware checkpointing techniques [10]

2.5.2.2 Bitstream manipulation

Bitstream manipulation is the key point when state extraction, or context save, is performed by configuration readback and register capture. In order to restore a PRM with its context from a previous run, the partial reconfigurable region, where the module is placed, must be uploaded with an updated partial bitstream. Generation of this new and updated partial configuration file is accomplished by masking the necessary bits in both the readback file and the initial partial configuration file. That means three files are used to create a new partial configuration file; the initial bit file, the mask file and the readback capture file. The initial bit file contains the module with its initial states, the readback capture file contains register data at point of capture and the mask file contains information of which bits in these two files are configuration data. According to [14] the creation of the new partial configuration file is done by the modification algorithm in equation 2.1. Next section gives details of the mask file.

$$New\ partial\ config.file = (readback\ file * mask\ file) + \big(initial\ bit\ file * (NOT\ mask\ file)\big) \qquad 2.1$$

2.6 Xilinx Virtex-6 FPGA configuration

2.6.1 Configuration interfaces

The configuration file can be uploaded to the FPGA through four different interfaces. Either by a serial peripheral interface (SPI), bus-width parallel interface (BPI), SelectMAP interface or by JTAG interface. For *reconfiguration* purposes the Internal Configuration Access Port (ICAP) is used, or if the reconfiguration is controlled by an external device the SelectMAP interface is used. A brief description of these interfaces will be given below. In newer FPGAs with integrated embedded microprocessors,

such as the Zynq all programmable system on chip (SoC) from Xilinx, there is also a fifth configuration port, the processor configuration access port (PCAP). This port will not be covered here but see [15] for more information. All interfaces share the same dedicated configuration pins, some of them are shown in table 2.1. All details of the pins are described in [7].

Table 2.1 - Configuration pins

Pin name	I/O	Description
MODE [2:0]	Input	Determines configuration mode. Sampled on the rising edge of INIT_B
CCLK	Output	Configuration clock source for all configuration modes except JTAG
DONE	Bi-directional	Active high signal indicating configuration is complete. 0 = FPGA not configured 1 = FPGA configured
PROGRAM_B	Input	Active-low full-chip reset
D [15:0]	Bi-directional	BPI input / SelectMap input / output.

2.6.1.1 Serial Peripheral Interface (SPI)

This interface is used when the configuration file resides in a serial flash memory. The FPGA device provides clock for the flash, and default memory address always starts from zero. The flash can be programmed through the Xilinx tool iMPACT.

2.6.1.2 Bus-width Peripheral Interface (BPI)

This interface is used when the configuration file resides in a parallel flash memory. Bus widths of 8 or 16 are supported. The bus width is auto detected via the bitstream. BPI-Up mode sets the starting address to zero and increments it by 1 until DONE signal is asserted. BPI-Down mode set the starting address to memory end address and decrements it by 1 until DONE signal is asserted. The flash can be programmed through the Xilinx tool iMPACT.

2.6.1.3 SelectMAP

This interface provides data bus widths of 8, 16 or 32 bits. The data bus is also bi-directional and is therefore used for bitstream readback. As with the BPI interface the bus width is auto detected. The interface is synchronous.

2.6.1.4 JTAG

Xilinx Virtex-6 devices supports IEEE 1149.1 standard of Test Access Port and Boundary-Scan architecture which are commonly referred to as JTAG. Configuring an FPGA by this method is quite slow as the frequency of the JTAG interface is often low and the fact that it is a serial interface. Its upside is that it can be used by debugging software like ChipScope Pro to get access to internal test signals, defined by the user, within the FPGA to provide a debugging platform.

2.6.1.5 ICAP

ICAP is the internal configuration port used for readback and reconfiguration. It is similar to the SelectMAP port but has no physical ports connected to the outside world. It acts solely internal and is therefore the configuration port of choice regarding partial reconfiguration. The ICAP interface is a Xilinx primitive, meaning that it can be instantiated in the VHDL or Verilog code. The ICAP supports 16 and 32-

bits data bus. According to Xilinx documentation, the ICAP clock can only run at frequency of 100 MHz which corresponds to a data rate of 3200Mb/s (400 MB/s). The low clock frequency is the main reason for slow reconfiguration.

2.6.1.5.1 Enhanced ICAP

There have been some attempts at overclocking the ICAP clock with great success. In [16] the authors managed to overclock the ICAP clock up to 500-550 MHz giving a data rate between 2000-2200 MB/s. This is five times faster than original speed and it significantly reduces the configuration speed. However, the measured data rate is accomplished writing the configuration only. When performing readback the authors didn't overclock the ICAP clock in order to prevent corruption of the readback data. The authors used Virtex-4 and Virtex-5 in their experiments and not a Virtex-6 device. So the overclocking has not, to my knowledge, been confirmed to be working on Virtex-6, but it is reasonable to assume that it will since there is no difference between the ICAP primitive from Virtex-4 and Virtex-6.

2.6.2 Configuration frames and frame addressing

The configuration data is organized in frames, the smallest addressable configuration segments. The FPGA logic space is arranged in rows and columns, and in addition it is divided into top half and bottom half [8]. A row has the height of one column, which corresponds to the height of one clock region. Top and bottom half each contains several rows, vertically aligned, each row contains several columns, horizontally aligned, and each column contains several frames, also horizontally aligned, as seen in figure 2.8 below.

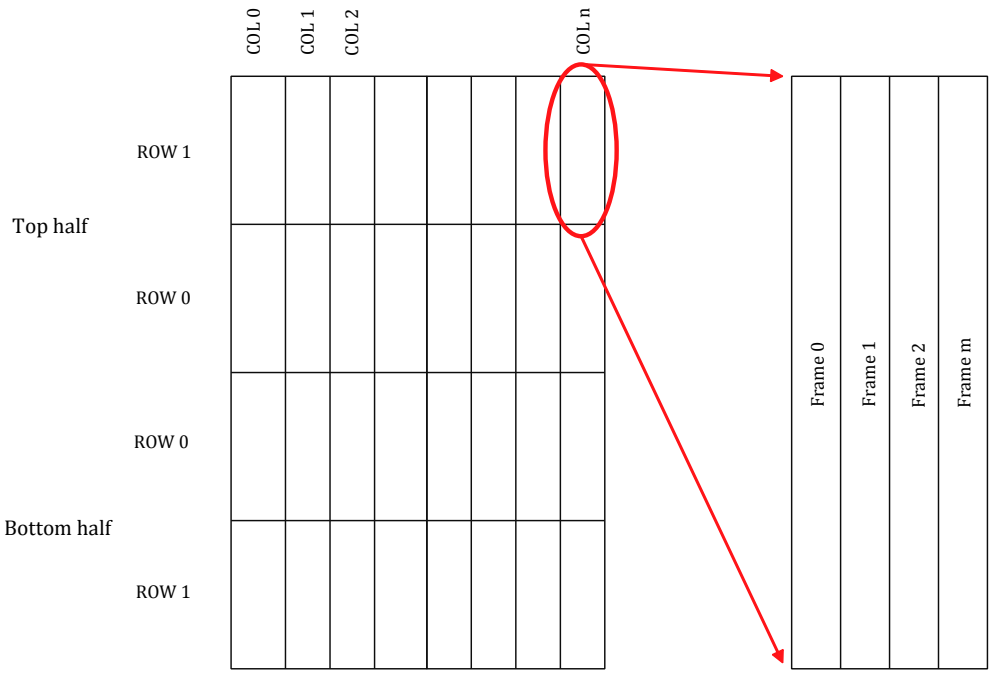


Figure 2.8 - Virtex-6 configuration architecture.

In the center of a row lies a horizontal clock row (HROW). A configuration frame spans the height of 40 CLBs, 20 above HROW and 20 below. Each frame has a unique 24-bit address that can be divided into five sections as seen in table 2.2 below. The 24-bit address is sent as a 32-bit word and hence the eight MSBs are unused. All frames have identical length of eighty one 32-bits words, or 2592 bits. The XC6VLX240T-1 device has a total of 28488 configuration frames and additional 583 32-bit words used for configuration commands [7]. A write to the frame address register (FAR), a section is dedicated to configuration registers, determines which frames to read from or write data to.

Table 2.2 - Frame address format

Address type	Bit index	Description
Block	[23:21]	Valid block types are CLB, I/O, CLK (000), block RAM content (001), and CFG_CLB (010). A normal bitstream does not include type 010.
Top / Bottom	[20]	Select between top-half rows (0) and bottom-half rows (1).
Row Address	[19:15]	Selects the current row. The row addresses increments from center to top, and then resets and increments from center to bottom.
Column Address	[14:7]	Selects a major column, such as a column of CLBs. Column addresses start at 0 on the left and increase to the right.
Minor Address	[6:0]	Selects a frame within a major column.

2.6.3 Configuration packets

All data, either configuration data or configuration commands, are sent to the configuration control logic by means of *data packets*. There are two types of packets, type 1 packets are used to read or write the configuration registers, and type 2 packets are used to write long blocks of data [7]. Type 2 packets must always be preceded by a type 1 packet. Table 2.3 below shows the format of type 1 packets and table 2.4 shows the format of type 2 packets, where x denotes a bit, and R denotes a reserved bit, this applies for both tables on configuration packets. Table 2.5 shows which operation codes are available.

Table 2.3 - Configuration packet, type 1.

Packet type	op. code	conf. reg. address	reserved	word count
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
001	xx	RRRRRRRRRxxxxx	RR	xxxxxxxxxxx

Table 2.4 - Configuration packet, type 2.

Packet type	op. code	word count
[31:29]	[28:27]	[26:0]
010	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxx

Table 2.5 - Configuration register operations.

op. code	function
00	no operation (NO-OP)
01	read
10	write
11	reserved

2.6.4 Configuration registers

A configuration command is either a read from the configuration register or a write to the registers. Table 2.6 below shows which registers are essential for reconfiguration purposes of a Virtex-6 device. All configuration registers are described in detail in [7].

Table 2.6 - Configuration registers essential for reconfiguration.

Name	R/W	address	Description
FAR	r/w	00001	Frame Address Register
FDRI	write	00010	Frame Data Reg. Input, write configuration data
FDRO	read	00011	Frame Data Reg. Output, read configuration data
CMD	r/w	00100	Command Register

The FAR register is explained in the section on configuration frames above. A write to the FDRI register tells the configuration logic that configuration data are about to be sent to the frame address specified in the FAR register The FDRO register provides readback data when readback has been set up. The read starts from the frame address specified in the FAR register and then auto increments until number of words specified in the word count is reached.

The CMD register instructs the configuration logic to perform configuration functions. The instruction present at the CMD register is executed each time the FAR register is loaded with a new value. Table 2.7 shows which instructions are essential for reconfiguration and readback.

Table 2.7 - Essential instructions in the CMD register related to reconfiguration.

Command	Code	Description
WCFG	00001	Writes Configuration Data: used prior to writing configuration data to the FDRI.
RCFG	00100	Reads Configuration Data: used prior to reading configuration data from the FDRO.
START	00101	Begins the Startup Sequence: initiates the startup sequence. The startup sequence begins after a successful CRC check and a DESYNC command are performed.
GCAPTURE	01100	Pulses GCAPTURE: Loads the capture cells with the current register states.

An example of commands is given in table 2.8. This example is an excerpt taken from an actual configuration file for the ML605 Virtex-6 device. The example shows how to decode the configuration data present in the configuration file. The data is broken down do match the format of the specific packet type and then decoded. Marked data in the figure 2.9 shows the configuration commands and data used in the example.

Offset(d)	00	04	08	12	16	20	24	28
00000000	FFFFFFFF	FFFFFFFF	000000BB	11220044	FFFFFFFF	FFFFFFFF	AA995566	20000000
00000032	30020001	00000000	30008001	00000000	20000000	30008001	00000007	20000000
00000064	20000000	30022001	00000000	30026001	00000000	30012001	02003FE5	3001C001
00000096	00000000	30018001	04250093	30008001	00000009	20000000	3000C001	00000001
00000128	3000A001	00000101	3000C001	00000000	30030001	00000000	20000000	20000000
00000160	20000000	20000000	20000000	20000000	20000000	20000000	30002001	00000000
00000192	30008001	00000001	20000000	30004000	502335C8	00000000	00000000	00000000
00000224	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000256	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000288	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000320	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Figure 2.9 - Data from an actual configuration file for a Virtex-6 device.

Note that the format of the command is decided by the packet type. The word count gives how many configuration packets are connected to the configuration command. When the word count in a type 1 packet is set to zero a packet of type 2 is following as seen in the example above. A packet of type 2 must always be preceded by a packet of type 1. The 32-bit word `0x20000000` in the example corresponds to a type 1 packet with operation code `00` which means that is an NO-OP command and the configuration control logic does not execute a command for one clock cycle. The last command tells the configuration control logic that the next 2370528 words are configuration data. This corresponds to 28488 configuration frames and hence configuration for the whole device.

Table 2.8 – Example of decoding the configuration file.

Data in hex:	**30008001**				
Data in binary:	00110000000000001000000000000001				
Format:	*Packet type*	*op.code*	*address*	*res.*	*word count*
Data broken up accordingly to fit the format given by three first bit (packet type):	**001**	**10**	**00000000000100**	**00**	**00000000001**
Decoded command:	Type 1	write	CMD register	-	1 word

Then follows 32-bits of data connected to the command above

Data in hex:	**00000001**	
Data in binary:	00000000000000000000000000000001	
Instruction in CMD reg:	WCFG	Write configuration data

Next command in the configuration file follows

Data in hex:	20000000				
Data in binary:	00100000000000000000000000000000				
Format:	*Packet type*	*op.code*	*address*	*res.*	*word count*
Data broken up accordingly to fit the format given by three first bit (packet type):	**001**	**00**	**00000000000000**	**00**	**00000000000**
Decoded command:	Type 1	no-op	(CRC register)	-	-

The above command is a NO-OP operation, and next command follows

Data in hex:	**30004000**				
Data in binary:	00110000000000000100000000000000				
Format:	*Packet type*	*op.code*	*address*	*res.*	*word count*
Data broken up accordingly to fit the format given by three first bit (packet type):	**001**	**10**	**00000000000010**	**00**	**00000000000**
Decoded command:	Type 1	write	FDRI register	-	-

Then follows 32-bits of data connected to the command above

Data in hex:	**502335C8**		
Data in binary:	01010000001000110011010111001000		
Format:	*Packet type*	*op. code*	*word count*
Data broken up accordingly to fit the format given by three first bit (packet type):	**010**	**10**	**00000100011001101011100 1000**
Decoded command:	Type 2	write	2307528 words (28488 frames)

2.6.5 Configuration readback and register capture

As mentioned earlier, it is possible to perform a so-called readback of configuration data. This means that the configuration memory can be read and outputted. Readback may be used to perform a verification of configuration, to correct a single event upset (SEU), for debugging purposes, or, as in this case, to perform partial reconfiguration with context switching. The readback function is performed at users' request. The initiator must send a sequence of configuration commands to the configuration control logic in order to perform a readback. The command sequence can be found in [7]. The user may use external devices as a microprocessor, another FPGA system or

the FPGA itself to send the command sequence. The commands have the same encoding as seen in the example in the previous section. When a readback is initiated the first frame of data is a *pad-frame*, which contains no data of interest, and must be discarded, see section on mask file below.

The value present at each flip-flop in a LUT can be captured and read back from the device. An instruction, GCAPTURE, in the CMD register initiates the capture and the flip-flop value is transferred to the configuration memory. The value resides in the memory location normally used for programming that specific flip-flop. The command sequence for readback capture requires an additional step compared to a normal readback, and that is to set the GCAPTURE instruction. There is another method to transferring the flip-flop value to the configuration memory; instantiate the CAPTURE_VIRTEX6 primitive and assert the CAP input; the flip-flop value is then sampled and stored in the configuration memory at the next rising edge of the primitives clock [7]. The primitive has two operations: one-shot or continuous. One-shot captures the flip-flop values once.

2.6.6 Xilinx ISE configuration options

There are options in the Xilinx tool ISE that need to be set in order to enable the reconfiguration capability. In process properties for the 'Generate Programming File' process there is a category named 'Readback Options'. Here is the security option of enabling or disabling readback and reconfiguration. It must be set to 'Enable Readback and Reconfiguration'. In addition, there are two other options available that is necessary to accomplish reconfiguration by means of the proposed system in this thesis: 'Create ReadBack Data Files' and 'Create Mask File'.

2.6.6.1 Mask file

In order to extract register information from the readback data, we need to know where in that data the information is located. The mask file generated by the ISE tool masks which bit in the readback data that are actual configuration data and not configuration commands or other data. Mask file has the file extension .MSK. This file is used to create a new partial configuration file, in compliance with equation 2.1, for the PRM that have been replaced. The content of the mask file does not match the readback data as seen in figure 2.10. It is therefore necessary to discard unwanted data from the readback data.

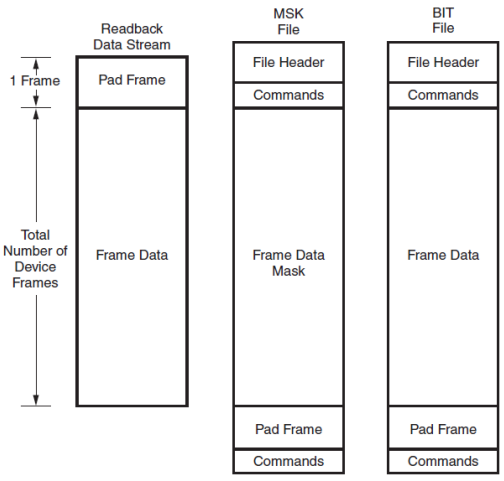


Figure 2.10 – Misalignment of file contents [7].

3 Related research

3.1 Chapter overview

As mentioned in the introduction there has been extensive research on the subject of partial reconfiguration, but not as much on the issue of context switching. The most relevant work, to my knowledge, compared to this thesis is presented here. They are listed in order of increasing relevance.

3.2 Research papers

"*Multitasking on FPGA coprocessors*" [12]. The paper discusses requirements for context switching, or multitasking as the authors describe it. The paper focuses on context switching by means of bit stream read back and is therefore highly relevant. The requirements are listed in table 3.1 below.

Table 3.1 - Context switching requirements.

1	The FPGA must provide a readback bit stream that contains the *current* state of all registers and internal memory.
2	Each register and memory bits in the FPGA must be able to preset or reset when a module is restored.
3	Configuration and readback must be sufficiently fast to keep switching time as low as possible.
4	Freezing the modules clock is necessary in order to take a capture of the registers and memory elements.
5	If the module uses more than one clock the module must be interrupted only between cycles of the slowest clock.
6	To avoid invalid data the clock must be stopped when it is safe, i.e. not when there is a memory addressing phase.
7	The modules that are switched cannot contain latches or registers by means of combinatorial logic loops as states of these types cannot be accessed.
8	If an external data source, or destination, is connected to the switching module the communication between the module and the external source/destination must be halted until the module is restored.

They also propose a client-server model of a hardware management unit (HMU) where the HMU is the server and the partial reconfiguration regions are the clients. The HMU is necessary to control the context switching. The setup included in this paper requires an external CPU that handles the reconfiguration process (the CPU act as HMU). No results were presented.

"*Efficient hardware checkpointing: concepts, overhead analysis and implementation*", [13]. This paper presents a tool, STATEACCESS, which utilizes hardware checkpointing to recover a module's state in case of an operational fault. Hardware checkpointing adds some hardware to an existing module in form of a *checkpointing finite state machine* (CFSM). The main purpose of hardware checkpointing is to regularly take 'snapshots' of the internal states of a module in a fault-free operation in order to restore the module when a fault occurs. The paper also considers three different methods to perform hardware checkpointing which are; i) Memory-Mapped state access (MM), ii) Scan Chain based state access (SC) and iii) Shadow Scan Chain based state access (SHC). These methods provide four types of overheads defined by the authors: hardware overhead H (in terms of extra flip-flops and LUTs), performance reduction R (reduction of maximal clock frequency), time-overhead C (increase in execution time) and latency L (time for a complete checkpoint arrives at the rollback device). MM integrated the checkpointing flip-flops into a memory space of a CPU. With SC the flip-flops are chained together through a long shift-register chain where each flip-flop has a multiplexer associated that switches between normal operation and shift-register operation. The last method, SHC is actually a state copy. For every state-register in a module there exists another register that copies the state. This creates a massive hardware overhead. The authors tested hardware checkpointing on three different modules, (a) DES56 cryptographic IP core, (b) 16 tab FIR filter and (c) FFT/IFFT coprocessor. The paper reported following hardware overhead. 2%-11% increase in flip-flops and 32%-83% increase in LUTs for MM, 4%-12% increase in flip-flops and 5%-66% increase in LUTs for SC, and finally 103%-122% increase in flip-flops and 72%-121% increase in LUTs for SHC. Hardware checkpointing is implemented in design flow and does not consider context save and restore through bitstream readback.

"*Context saving and restoring for multitasking in reconfigurable systems*", [17]. This paper proposes a combined software and hardware solution to context switching. It utilizes the bitstream readback functionality and extracts register information from the bitstream (State Extraction Filter). This information is then used to produce a new partial configuration file that contains the information of the registers and memory elements (State Inclusion Filter). Another benefit of this system is that it contains a database of which configuration frames that holds the value of a register or a memory bit. That means that it isn't required to perform a read back of all the configuration frames within the partial region but only the frames that contains the register and memory elements. This significantly reduces the overall context switching time in comparison to reading all the frames. The main focus of the paper is to develop a system that relocates the saved module to another part of the device. The implemented system includes a previously made relocation filter, REPLICA, from the same authors. They implemented the system on an XCV2000E (Virtex-E) and their test result shows a total relocation time (including state extraction and state

inclusion, also known as context save and context restore) between 0.4 ms and 14.8 ms with a partial bit stream size ranging from 9.3 kB to 331.7 kB.

"*A novel mechanism for effective hardware task preemption in dynamically reconfigurable system*" [18]. This paper presents a tool, BitFormatter, that analysez the configuration file for each PRM. The tool then generates a DPRS Bitstream Format (DBF) file that contains a State Data Descriptor Table (SDDT). The table contains information about which frames the PRM's state information is situated in, and that frame's FAR value. When readback capture is initiated only those frames listed in the SDDT are captured. These frames are then written into their corresponding location in the initial configuration file. Restoring the same PRM is done by loading the FPGA with the modified bitstream. The authors used a Virtex-4 device in their experiments and an embedded CPU to control the reconfiguration process. Since only a few frames are read, compared to the total number of frames in the PRR, the readback process is fairly fast. Their test results shows a readback time of only 60.11 µs for a readback file of 3.68 kB, and a reconfiguration time of 536.9 µs for a modified configuration file of 158.83 kB. Modification time was not directly measured, but a comment on the issue was given; 'slightly more than 1 ms (…)'

"*Hardware context-switch methodology for dynamically partially reconfigurable systems*" [19]. The paper presents a hardware system that performs context switching. This system is similar to [17] except that instead of a CPU all work is done by means of hardware and without the relocation filter. The database in this system contains information about frame address and bit index. This information is derived from the logic allocation file generated by BitGen. The register information is extracted from the file, and with the frame address information, this constitutes the database contents. Whereas [17] used a CPU to perform the read operations, the system proposed in this paper use a command ROM to store the configuration commands needed. In experiments the authors used XC2V1000 and XC2VP20 devices. The presented results in this paper are not directly comparable.

"On-chip context save and restore of hardware tasks on partially reconfigurable FPGAs" [14]. This paper proposes an autonomous software based on-chip context save and restore (CSR). They used a MicroBlaze soft core embedded processor with a Linux-like operating system based on BusyBox. The process of context switching consist of three main steps. First is to save the context (CS), which means to initiate capture of flip-flops. Next, the captured data is merged with the initial bitstream to make a new partial bitstream; this process involves bitstream manipulation with mask file, captured file and the initial bit file. The last step is context restore (CR) which is to send the new merged bit file to the PRR. In compliance with requirements presented by [12], the PRR clock is halted during the entire CSR. Their experiment was performed on a Virtex-5 device and they tested the system on PRRs that ranged from one to twelve columns. They only measured how long the intermittent processes lasted. There are four different time measurements. T_{CS}, T_{merge} and T_{CR} where 'CS' is the context save process, 'merge' is the process of creating new partial configuration file, and the 'CR' is the context restore process. For a PRR of twelve columns and a PRM flip-flop count of 1920 the reported times where: T_{CS} = 13.06 ms, T_{merge} = 19.39 ms and T_{CR} = 13.23 ms.

3.3 Summary

Table 3.2 below gives a summary of the research papers above. The table shows which device was used, which techniques was used for extracting the states, and if the reconfiguration controller were implemented in software (CPU) or in hardware.

Table 3.2 - Summary of essential parameters.

Author	Device	State Extr. Tech. [1]	HW/CPU[2]
[13]	Virtex-II	Hardware checkpoints	-
[17]	Virtex-E	Bitstream manipulation	Both[3]
[18]	Virtex-4	Bitstream manipulation	CPU
[19]	Virtex-II	Bitstream manipulation	HW
[14]	Virtex-5	Bitstream manipulation	CPU

[1] State extraction technique used.
[2] Reconfiguration controlled by hardware or by CPU?
[3] A combination of embedded CPU and hardware modules are used.

4 Implementation

4.1 Chapter overview

This chapter first explains the concept of the system that has been designed for this master thesis. Then follows an overview of the system, in section 4.3, where a functional explanation is given. The chapter continues to give a short description of the hardware platform used in the project. In section 4.5 the implemented system is described, the section gives an overview on the data flow, and an operational description is given. In the final section a detailed description on the implemented modules is given. Simulation captures and excerpt of the code is given when it is necessary to highlight a problem or to illustrate how they were solved.

4.2 System Concept

As described in the introduction, the foundation of the system in this thesis is based on two PRMs sharing the same PRR over time. Figure 4.1 below shows the system concept.

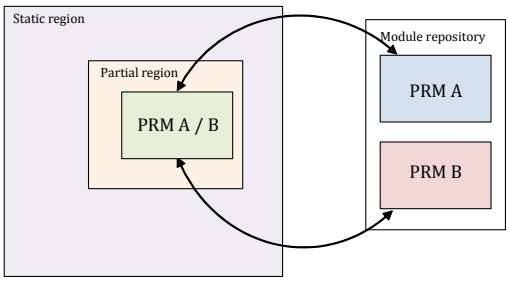


Figure 4.1 - Concept of partial reconfiguration with context switching

PRM A runs for a predetermined amount of time, and it is then switched with the PRM B. After the same amount of time the PRM A is reintroduced in to the PRR. Now it starts from where it was when it was replaced. Switching of PRMs continues as long as the FPGA is powered. As the figure 4.2 below shows, the PRMs are time multiplexed in the PRR. In-between execution time of PRMs, the system performs context save (CS) and context restore (CR).

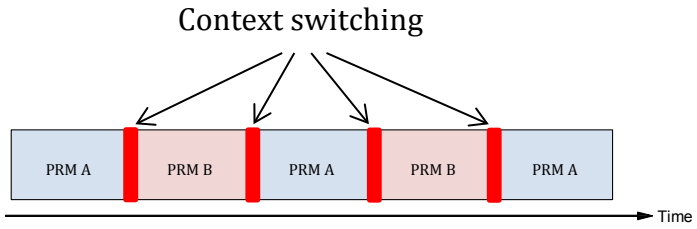


Figure 4.2 - Time multiplexed hardware sharing

The creation of new partial configuration file is performed concurrently as the other PRM is executing, illustrated in figure 4.3. The CS and CR process occupies both the static region and the partial region. In the static region it writes and reads the readback and partial bitstream data to and from the memory. The partial region is halted when both CS and CR is in progress.

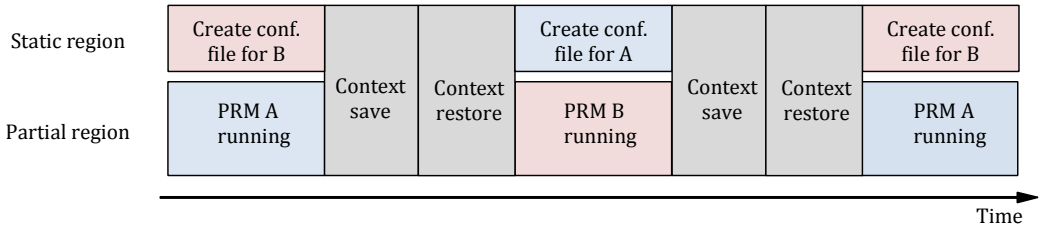


Figure 4.3 - Details of the context switching in Figure 4.2

4.3 System overview

The system for switching between PRM A and PRM B can be described as four phases of operation, visualized in figure 4.4 on the next page.

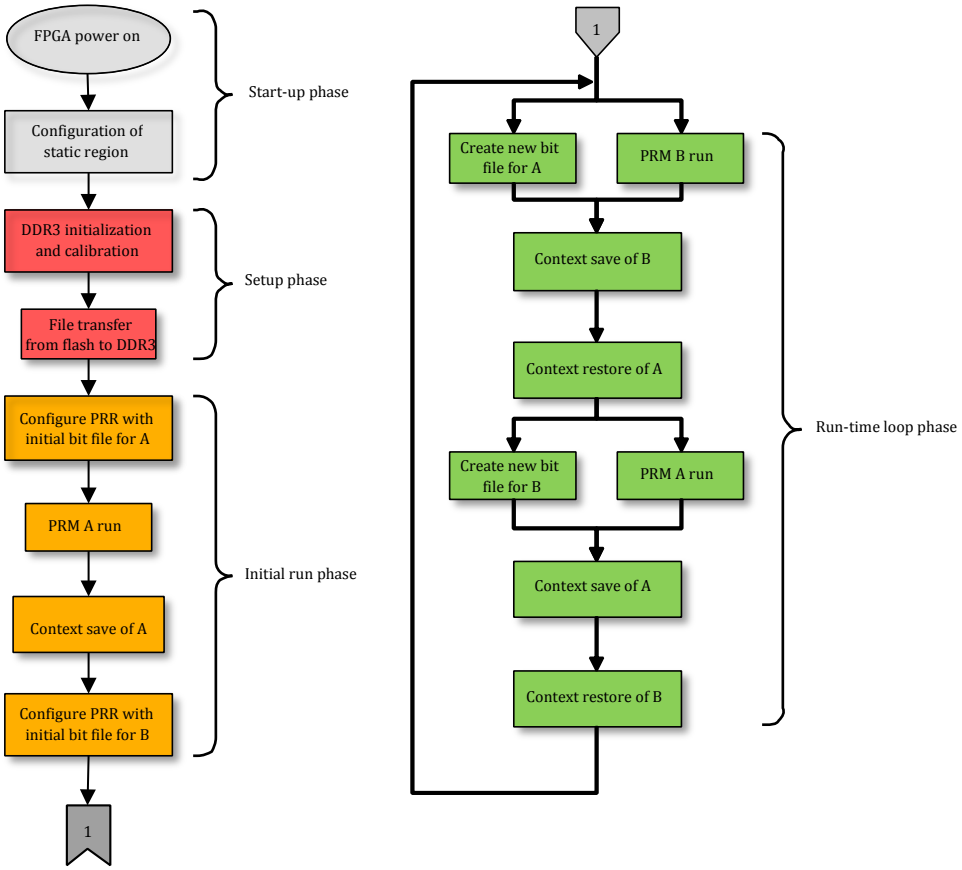


Figure 4.4 – The four phases of operations of the system.

The start-up phase, marked as grey in the figure, consists of two stages:

1) Powering up the FPGA
2) Configuration of the static region

This phase is normal for any partitioned design. The partial region is not configured.

Next phase is the set-up phase, marked as red in the figure, and consists of two stages:

1) Initialization and calibration of the DDR3 memory
2) Transferring of files from the flash to the DDR3

Stage 1 is normal for the onboard DDR3 memory at every start-up, see [20] for more details. Duration of the initialization and calibration sequence is about 220 µs. Since the generation of a new partial configuration file is depended on the mask file and the initial partial configuration file, these files must be uploaded from a computer to the onboard flash memory, via the Xilinx tool iMPACT. The flash memory is not suitable for context switching for reasons given in the next section. The files must therefore be transferred to the DDR3 memory. Immediately after completion of stage 1 the transferring of files starts. This stage continues until all the files are transferred.

The initial run phase, marked as orange, consists of four stages:

1) Configuration of the PRR with the initial bit file for PRM A
2) PRM A run-time
3) Context save of PRM A

4) Configuration of the PRR with the initial bit file for PRM B

Since there is no saved context, the PRR is configured with the initial bit file for PRM A. After configuration, the PRM A runs for a predetermined amount of time. When the time limit is reached, the context of PRM A is saved through register capture and readback. The saved data is stored at a predefined memory location. Final stage of the initial run phase is the configuration of PRR with PRM B. This is the first run of the PRM B, and there is, as for the first stage, no saved context for the PRM. The initial bit file of the PRM B is therefore used to configure the PRR.

Next and final phase is the run-time loop phase, marked as green. The creation of new partial configuration file for PRM A is done simultaneously as the PRM B runs, as per figure 4.3. When the run-time limit is again reached the context of the PRM B is saved. After the first run of both PRMs there is now a readback file for each PRM so a context restore is performed. This operation fetches the modified bit file for PRM A from the DDR3 and reconfigures the PRR. While the PRM A runs the PRM B is modified just as previously for the PRM A. This phase loops until reset is asserted or the power is removed. The operation of the system is described later in this chapter.

4.4 Hardware platform

The ML605 evaluation board is used during the project. The board consists of a Virtex-6 XC6VLX240T-1 FPGA, 32 MB linear BPI flash memory, 16 MB platform flash memory, 512 MB DDR3 memory and some other peripherals. Figure 4.5 shows the block schematic of the board. Only peripherals used in this thesis are described here, the other peripherals are more thoroughly described in [21]. There are two main clocks connected to the FPGA device. A 66 MHz single ended clock and a 200 MHz differential clock. It is the 200 MHz clock that is the system clock for the system in this thesis.

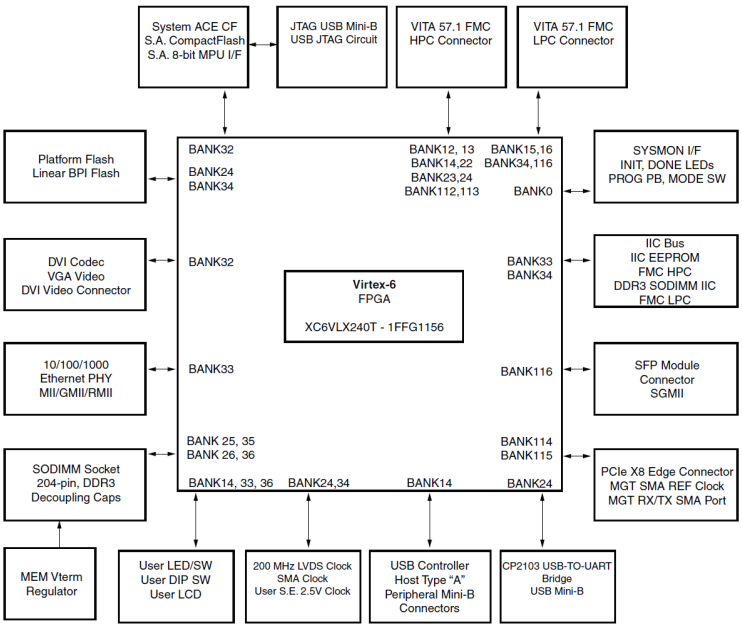


Figure 4.5 – ML605 block schematic [21].

4.4.1 Virtex-6 XC6VLX240T-1 device

The Virtex-6 XC6VLX240T-1 device mounted on the ML605 board has 37680 CLBs, each with four LUTs and eight flip-flops which means 150720 LUTs and 301440 flip-flops. Further, it has 768 DSP slices, and with the package FFG1156 it has 600 user I/Os. For more details of the Virtex-6 device see [22].

4.4.2 Flash memory

There are two on-board flash memories on the ML605 evaluation board. One that primarily holds the configuration files (128 Mb Xilinx XCF128X-FTG64C Platform flash XL) and the other that holds data files as well as configuration files (256 Mb Numonyx / Micron JS28F256P30 Linear BPI Flash). The Platform flash is there to provide fast configuration in compliance with PCIe startup requirements and will configure the FPGA in less than 100 ms, the bit rate can be as high as 800 Mb/s [23]. These two flash memories share the same address and data lines but since the BPI flash is twice as large as the Platform flash it has one more address line. The selection of which flash to use is determined by the signal `P30_CS_SEL` and at startup the S2 switch-2. The S2 switch-2 can be overridden after configuration by the `P30_CS_SEL` signal [21].

The Linear BPI flash memory is used to hold the initial bit files and the mask files. The flash defaults to an asynchronous page-mode read upon power up or after a reset. For every page, of 4 words, there is an initial access delay of max 85 ns, and a delay of 25 ns from a valid address to valid output within a page [24]. Since the clock signal of the memory is tied to V_{CC2V5} synchronous burst read is not possible [21].

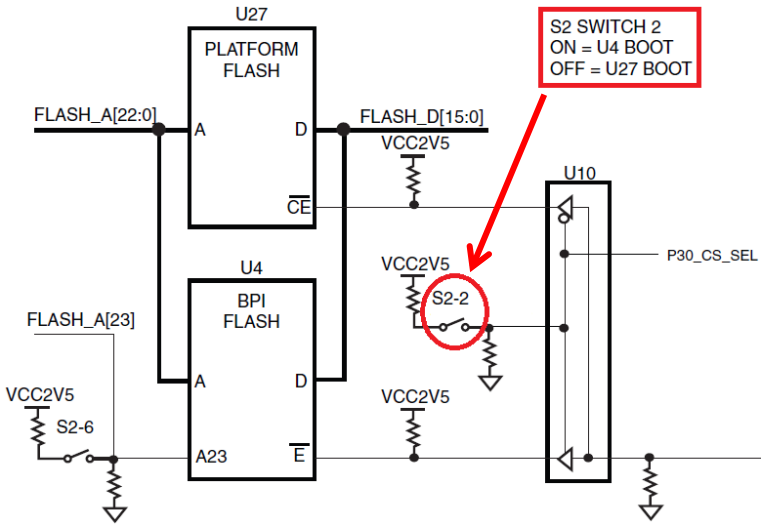


Figure 4.6 – Flash memory selector [21]

4.4.3 SDRAM memory

A 512 MB DDR3 SODIMM SDRAM is mounted on the ML605 evaluation board. The memory is of vendor Micron with part number MT4JSF6464H-1G1. The memory is capable of a transfer rate of 1066 MT/s (mega transactions per second) [25], and with a data width of 64 bits it corresponds to a data rate of ~67 Gb/s (8.5 GB/s). The memory device operates from a differential clock. A temperature sensor IC is

mounted on the memory device which uses the I²C communication protocol. This sensor is however not used in the memory interface generator (MIG) core provided by Xilinx. This is explained more detailed in section 4.6.3.1. The memory supports x8 burst which means memory accepts 8 x 64-bits data to be read or written at the same address, and that the next address is incremented by 8.

4.4.4 ML605 configuration mode switch

There are three configuration mode switches (S2 switch 3-5) that sets the configuration mode as mentioned in chapter 2.6.1. Table 4.1 lists different modes available for the ML605 evaluation board.

Table 4.1 - Configuration mode switch settings.

Configuration Mode	S2 switch 3-5	Bus width
Master Serial	000	1
Master SPI	001	1
Master BPI-Up	010	8,16
Master BPI-Down	011	8,16
Master SelectMAP	100	8,16
JTAG	101	1
Slave SelectMAP	110	8,16,32
Slave Serial	111	1

Since the configuration file is stored in the BPI flash, the switch setting is `010` which set the mode to master BPI-up.

4.5 Implemented system

According to [24] the highest frequency of the flash memory is 52 MHz. Flash memory data bus is 16-bit wide, and the memory has a capacity of only 32 MB. The low capacity may become a concern when the PRR becomes large. The memory should be able to hold 4 files for each PRM; the initial bit file, the mask file, the readback file and the created partial configuration file.

The onboard DDR3 memory takes 512 bits of data per address and operates at 400 MHz. Its capacity is 512 MB and has no problem of holding all the necessary files. Consequently, the DDR3 memory is faster at creating the new partial configuration file than the flash memory.

The system consists, as seen in figure 4.7, of two external components; flash memory and DDR3 memory, and six hardware modules; flash memory controller (FMC), DDR3 memory controller (DMC), file transfer module (FTM), bit manipulator module (BMM), data width converter (DWC) and ICAP instruction memory module (IIMM). The latter three modules reside in the CSMC. The DMC is based on an IP core from Xilinx, memory interface generator (MIG), and modified to suit the design. The system also includes one ICAP primitive in order to perform reconfiguration and readback from within the FPGA.

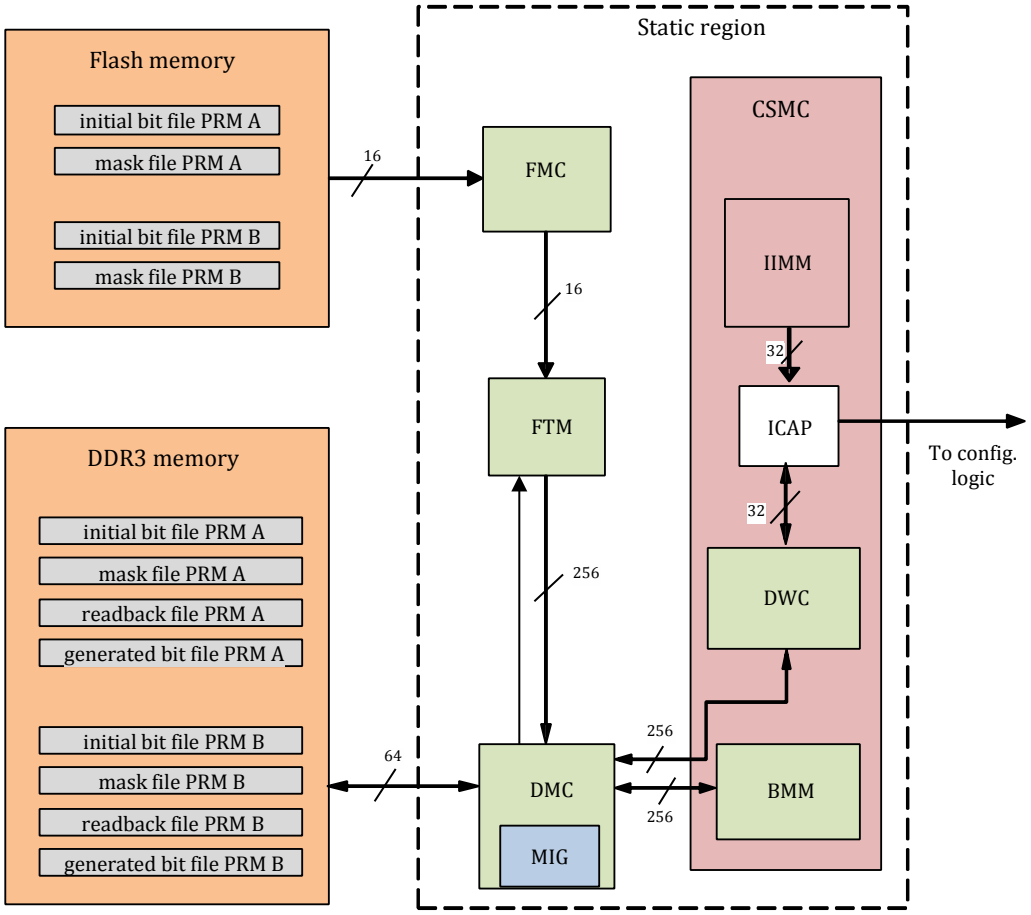


Figure 4.7 – Implemented system and data flow.

Figure 4.7 also shows which modules that have been implemented and tested. The orange blocks represent the external components, the green blocks represents the modules implemented in this work, the white block represents the ICAP primitive, and the red block are the modules that have not yet been implemented due to restricted duration of the thesis, as mentioned in the introduction. The DMC has an internal blue block that represents the IP core MIG.

The flash memory holds the two of the three files, initial bit file and mask file, essential for the generation of new bit file. To read these files, a flash controller is therefore necessary. These files are transferred to the DDR3 memory at start-up. This is the only interaction with the flash memory. All files are stored, at run-time, in an external DDR3 memory. The CSMC, FTM and the BMM all interact with the external DDR3 memory, FTM only at the start-up phase.

The CSMC is controlling the whole process of context saving, creation of a new partial configuration file and context restore. The IIMM contains the necessary reconfiguration and readback command sequences. To generate a new partial configuration file the algorithm in equation 2.1 is realized in the BMM. ICAP data bus is 32-bits wide and the DMC has a 256-bits wide data bus. The DWC converts 32-bits data into 256-bits data and vice versa.

4.5.1 Data flow

Figure 4.7 also shows the data flow of the system. There is no writing of data to the flash memory and since the data width is 16-bit it must be converted into 256-bit which is the internal bus width of the DMC. This conversion is performed inside the FTM. The ICAP primitive has a 32-bit data bus and the DWC converts it to 256-bit before the readback data is sent to the DMC. This is a standalone converter since the one used in FTM is used only once in the set-up phase and it can only convert from 16-bit to 256-bit, not from 32-bit to 256-bit and the other way around. The BMM does not interact with the ICAP and is therefore not connected to the converter since the data width of BMM is 256-bit. The IIMM has a data bus width of 32-bit which matches the ICAP data bus, and is therefore not connected to the converter.

4.5.2 System operation

Due to the fact that the system is not complete, it is necessary to make some assumption in order to explain the theory of operation of the system as it is intended. The assumptions are listed below.

1) The partial reconfiguration region is already implemented.
2) The region is a single island style.
3) Two partial reconfigurable modules, A and B, are available.
4) The modules have a footprint that fits the partial region.
5) Start-up phase has completed.

At set-up phase, the DMC performs initialization and calibration. When these tasks are completed the controller asserts a `phy_init_done` signal, which indicates that the initialization and calibration of the DDR3 memory have been completed, and the transferring of mask and initial files for each partial module begins, as seen in figure 4.8 below.

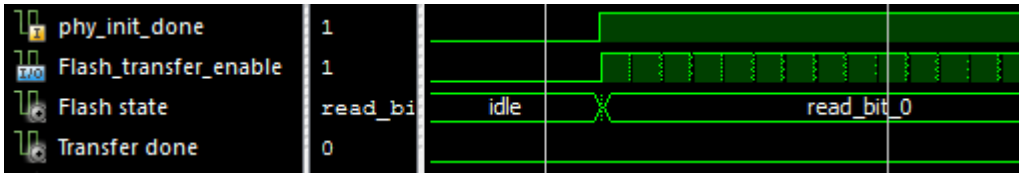


Figure 4.8 - File transfer begins when the DDR3 initialization and calibration is done.

The file transfer is automatic and is not controlled by another module other than the assertion of the `phy_init_done` signal. When the transfer is complete the transfer module asserts a `transfer_done` signal, which indicates that the transferring of files are complete, as depicted in figure 4.9, and the system enters the initial-run phase

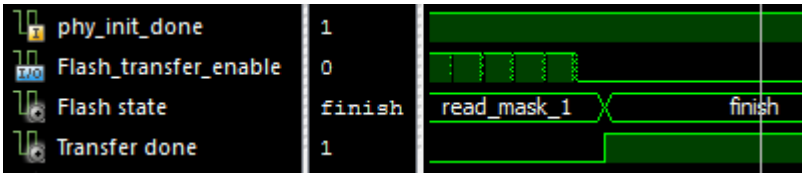


Figure 4.9 – The transfer_done signal indicates the completion of file transfer.

PRR is configured with PRM A. Since it is the first run of PRM A, and there is no saved context, the PRR is configured with the initial bit file. After a predefined time interval the context switching main controller initiates a CS of partial module A; a readback with capture command sequence is sent to the ICAP.

When the ICAP starts sending the readback data it is converted from 32-bit to 256-bit by the DWC. Immediately after the conversion, the CSMC asserts a write signal and sends the data to the DDR3 memory controller with a destination address. The memory controller sets the appropriate control signals and stores the data in the designated address.

For each DDR3 address the memory controller stores 512-bits of data so for the next piece of 256-bit data from the context switching main controller, the DDR3 address stays the same. The address is incremented by eight for each 512 bits of data. This process continues until the readback has completed. When the CS is finished, i.e. the last configuration frame of data is stored; the process of CR of PRM B begins. Configuration command sequence is sent to the ICAP. Since it is the very first run of the partial module B the context switching main controller fetches the initial bit file from the DDR3 memory and sends it to the ICAP via the data width converter. When the reconfiguration is done, the system enters the run-time loop phase.

Next process is important; while the partial module B is running, the bit manipulator receives a signal from the context switching main controller telling it to go ahead and generate new partial configuration file for the partial module A. The bit manipulator then fetches the first 512 bits from the location of the initial bit file and stores it in a local register; it then fetches the first 512 bits from where the mask file is stored and stores that data in another local register. The last data fetch is from the previously stored readback data. It now contains three 512-bit registers with data; it then runs the algorithm from equation 2.1 and as a final step it stores the newly created data in another memory location. The process described above repeats until power down of the FPGA. The time it takes for the bit manipulator to generate a whole new partial configuration file sets the minimum run-time width of the running module.

4.6 Module descriptions

4.6.1 Context switching main controller

Unfortunately this module was not implemented due to limited duration of the thesis, but as stated above, the CSMC controls the entire process of reconfiguration and context switching including the creation of new partial configuration file. ICAP, BMM,

DWC and the IICM should all reside in the CSMC, as per figure 4.7. It should be implemented as a state machine, in order to achieve correct sequencing of the different processes. The state machine should be designed in accordance with the flow chart in figure 4.4. There must be a multiplexer in the CSMC that selects the ICAP input. In case of CR, both DWC and IIMM would send data to the ICAP. In case of CS, the IIMM would send data to ICAP and DWC would receive data from ICAP.

4.6.2 Flash memory controller

The flash memory has four control signals; CE (chip enable), ADV (address valid), OE (output enable), and WE (write enable). Other I/Os include CLK, RST, DQ, ADDR, and WAIT [24]. All control signals and the reset signal is active low inputs. The FMC has one additional custom control signal, TRANSFER_EN, which is controlled by the FTM. It acts like an enable signal; reading is not available unless the TRANSFER_EN has been asserted. On the ML605 evaluation board, the clock and WP is tied to V_{cc2V5} and the ADV is tied to ground [26]. Only two signals, WE and OE, are therefore necessary to control. **Table 4.2** shows the settings of the control signals of the flash memory when a read operation is to be performed.

Table 4.2 - Control signals for read operation of the flash memory [24].

Bus operation		RST	CLK	ADV	CE	OE	WE
Read	Asynchronous	1	n/a	0	0	0	1
	Synchronous	1	running	0	0	0	1

The controller sets the signal WE and OE to logic high at reset. When the system is out of reset the controller waits for the TRANSFER_EN enable signal coming from the FTM. When the signal is asserted the controller sets the OE signal logic low, and reading of data from the flash begins. Since the address in to the controller is updated once every rising edge of the clock of the transfer module, the update of output is synchronously. Reading of flash data continues until TRANSFER_EN is de-asserted by the transfer module. When that happens the OE is set to logic high again, preventing further reading from the flash. Since the WE is tied to logic high through a pull-up resistor [26] it is actually not necessary to control this signal and hence the OE is the only signal the module need to control.

4.6.3 DDR3 memory controller

The DMC is based on the MIG IP core from Xilinx. The 'user design' block in figure 4.10 represents the DMC designed specifically for this thesis. A description of the 'user design' is given later in this section. The available documentation for the MIG IP core is, in my opinion, inadequate and lacks sufficient information. It also contains several errors, which makes the implementation difficult and time consuming.

4.6.3.1 Memory Interface Generator

It is quite difficult to control a DDR3 memory due to strict timing requirements and advanced data handling, and that is why Xilinx has made a memory controller IP core. The core makes controlling of the memory easy, compared to designing a DDR3

memory controller from scratch. However, there are some signals that need to be controlled in the right manner. A deeper understanding of the Xilinx MIG IP core is given here. MIG version 3.91 is used in this thesis. The core comprises three main blocks; physical layer block, memory controller block and a user interface block (UI) [27]. Figure 4.10 shows these blocks.

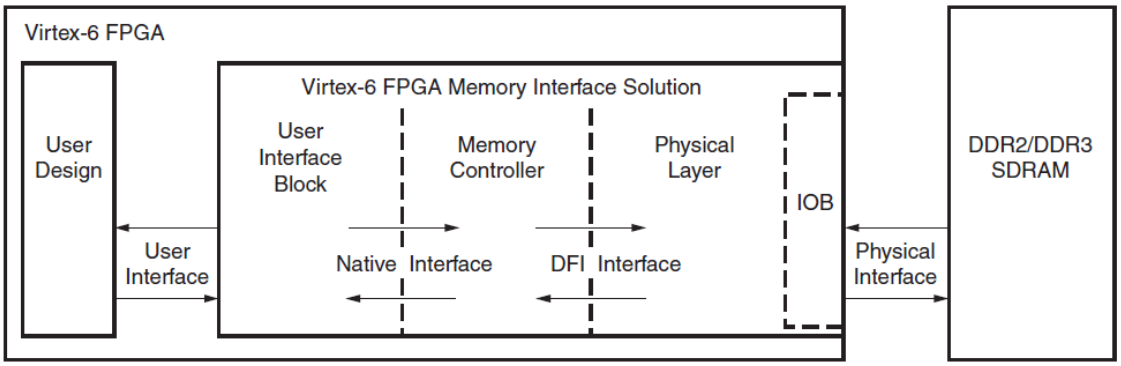
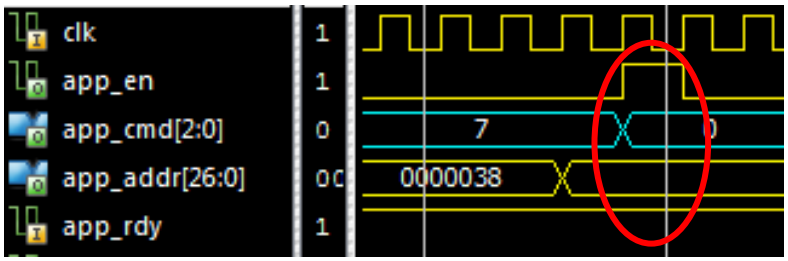


Figure 4.10 - MIG interfaces [27].

The physical layer block generates the correct timing and sequencing of signals to the memory module. The memory controller block receives requests from the UI and stores them in a logical queue. The UI creates an interface and presents a flat address space to the user. In addition it does reordering of retrieved data to match the requests. The interface between the memory controller and the UI block is called 'native interface'. It is possible to skip the UI block and interact with the memory controller through the native interface. This is somewhat complicated as retrieved data may be unordered compared to requests which mean that the user must reorder the retrieved data to match the requests. The UI block is therefore included in the system used in the thesis in order to keep it simple.

The MIG provides an 'infrastructure' module which instantiates the advanced mixed mode clock manager (MMCM_ADV) primitive. This primitive generates different clocks. The physical memory module runs at 400 MHz, while the IP core and rest of the modules runs at 200 MHz.

The UI has several inputs that need to be controlled by the user, and a few status outputs. Table 4.3 shows a list of these I/Os, signals marked red are control signals, and the signals marked blue are status signals. A command, that determines whether a read or a write operation is about to happen, is sent to the UI. The command is accepted by the UI when the `app_rdy` is asserted by the UI and the `app_en` is asserted by the user. This is called the *Command Path*. The `app_cmd`, `app_addr` and the `app_en` signal must be upheld until the `app_rdy` has been asserted. Figure 4.11 shows a command being accepted by the MIG. The address is set first; the command is set simultaneously as the enable signal is asserted. Since the `app_rdy` is already asserted by the MIG, the command is accepted immediately.



Command accepted.

Figure 4.11 - Command path.

When a write command is sent and accepted, the UI is waiting for the `app_wdf_wren` from the user. This signal is the enable signal for writing data to the DDR3. If the `app_wdf_rdy` is asserted by the MIG, two 256-bit words are sent to the memory in two consecutive clock cycles. Along with the last word `app_wdf_end` must be asserted by the user, indicating the last word of the data burst. The sequence is illustrated in figure 4.12. This means that it takes two clock cycles to write a full x8 burst of data. This is called the *Write Path*.

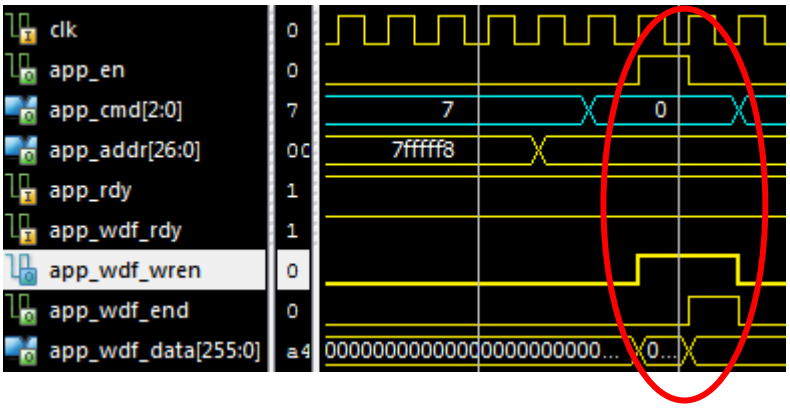


Figure 4.12 - Write path.

When a read command is accepted by the UI, the data at the current address are available at the `app_rd_data` output when the `app_rd_data_valid` is asserted by the UI. Just as in the write operation, there is an `app_rd_data_end` signal asserted by the UI, indicating the last 256-bit word of the read. The returned data from the memory are presented in the same order as requested by the user. This is called the *Read Path*. Figure 4.13 shows the read path. The assertion of `app_en` results in assertions of both `app_rd_data_valid` and `app_rd_data_end` after some clock cycles. It takes 23 clock cycles for the status signals to be asserted, so the figure has been modified in order to visualize the behavior.

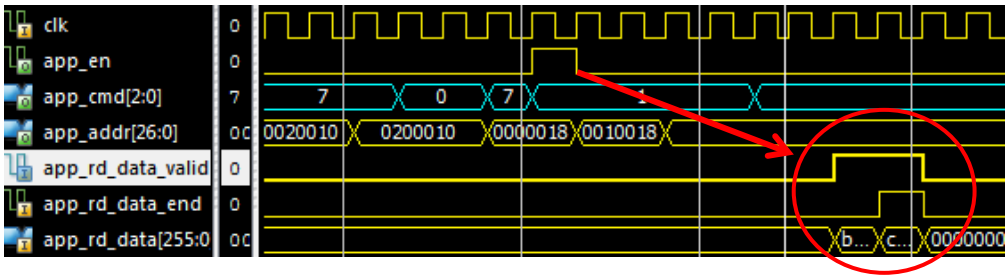


Figure 4.13 - Read path.

Table 4.3 - Control and status signals from the MIG user interface **[27]**. Signals marked red are control signals, and signals marked blue are status signals. The other signals are: address, data and write mask.

Signal	UI - I/O	Description
app_rdy	out	Indicates that the UI is ready to accept commands. If the signal is de-asserted when `app_en` is enabled, the current `app_cmd` and `app_addr` must be retried until `app_rdy` is asserted.
app_wdf_rdy	out	Indicates that the memory controller is ready to receive data. Write data is accepted when `app_wdf_rdy` and `app_wdf_wren` are both logic high.
app_cmd	in	Command for the current request.
app_addr	in	Address of the current request.
app_en	in	Active high strobe signal for the `app_addr` and `app_cmd`.
app_wdf_wren	in	Active high strobe signal for `app_wdf_data`.
app_wdf_end	in	Indicates that the current clock cycle is the last cycle of input data on `app_wdf_data`.
app_wdf_data	in	Data for the write command.
app_rd_data	out	Output data for the read command.
app_rd_data_valid	out	Indicates that the `app_rd_data` is valid.
app_rd_data_end	out	Indicates that the current clock cycle is the last cycle of input data on `app_rd_data`.

4.6.3.1.1 User design

The user design is the module interacting with the MIG user interface, in this thesis that corresponds to the DMC in figure 4.7. As mentioned earlier, the UI has three control signals that the user need to assert relative to a read or a write operation, see figure 4.14 for I/O ports of the controller. There are two ways this can be accomplished, the designer can let all the modules that interact with the MIG, control the signals themselves, or a single module can control them all, and have just a few enable signals as input, in addition to the address and data bus. This module then becomes a memory controller itself with a higher abstraction level than the MIG UI. The system proposed in the thesis uses the latter alternative.

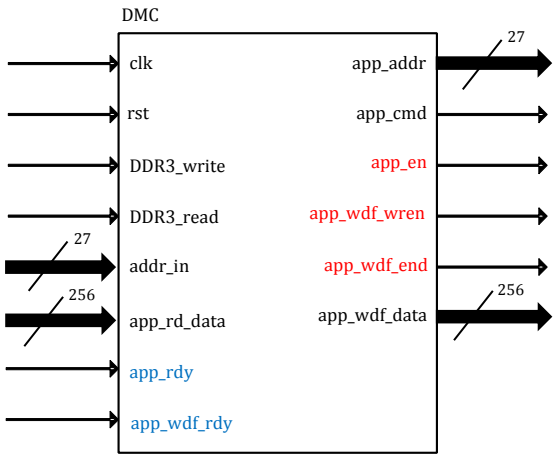


Figure 4.14 - I/O ports of the DMC. Color coding corresponds with the color coding in table 4.3.

The DMC has to enable signals; `DDR3_write` and `DDR3_read`. Based on these inputs, the controller asserts the corresponding control signals going to the MIG UI. Because controlling the memory is only dependent on two of the status signals coming from the UI, `app_rdy` and `app_wdf_rdy`, the other two are not connected to the DMC, as seen in figure 4.14. Each interacting module asserts the enable signal, corresponding to the request, for as long as it takes to perform the request. A multiplexer at the top level prevents more than one module accessing the memory at the same time.

In the case of a memory read there is a delay of twenty three clock cycles between the rising edge of the `DDR3_read` signal and the rising edge of the `app_rd_data_valid` signal from the MIG UI, see chapter 5.3. It is not necessary to hold the `DDR3_read` for the duration of the delay but it must be held high for two clock cycles per read operation.

The operation of the controller is based on an FSM, which consist of five states; idle, cmd, first_write, second_write and read. ASM of the FSM can be seen in figure 4.15. The idle state just waits for the assertion of either `DDR3_write` or the `DDR3_read` signal. When that happens the next state is set to cmd. The cmd state waits for the assertion of `app_rdy` from the MIG UI, if the `DDR3_write` is still high. If the `DDR3_read` is high the next state is set to read. The read state asserts `app_en`.

The first_write state sets `app_en` and `app_wdf_wren` high. The `app_wdf_end` is not asserted at this state and that indicates that the first 256-bit word of data is written at the current address. If `DDR3_write` is still asserted, the state waits for `app_wdf_rdy` signal. The next state is set to second_write when the `app_wdf_rdy` signal goes high. If the `DDR3_write` signal is de-asserted, the state machine goes back to the cmd state.

In the second_write state both `app_wdf_wren` and `app_wdf_end` signal is asserted, indicating the last 256 bit word that is to be written at the current address.

If the `DDR3_write` is still asserted then the state waits for confirmation by the MIG UI via the `app_wdf_rdy` signal before the FMS goes back to the cmd state. The DDR3 address is controlled by the interacting module.

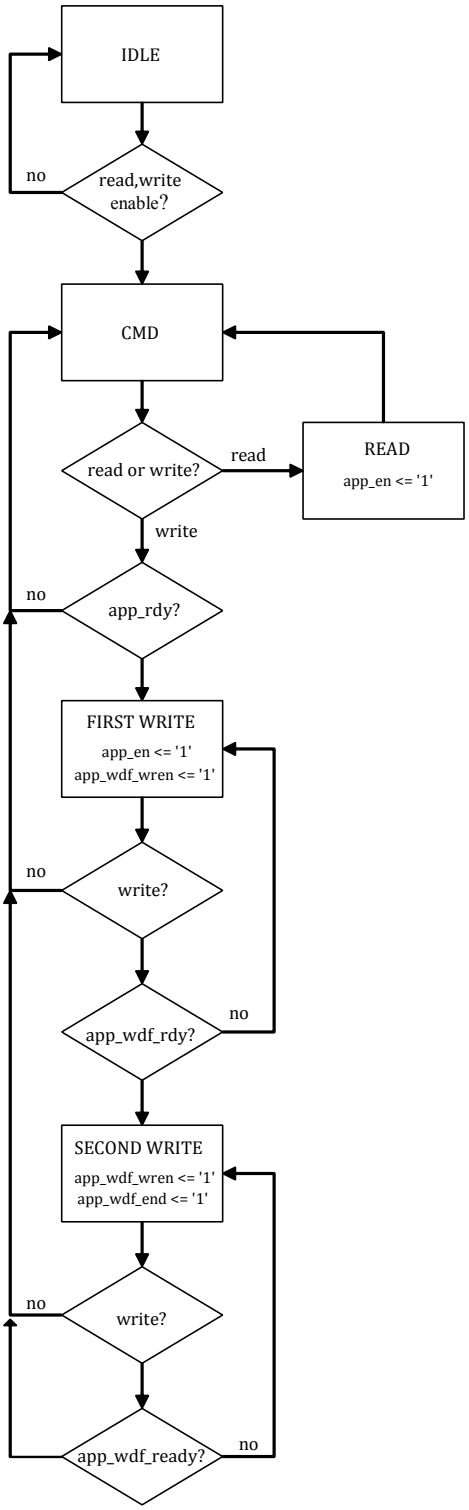


Figure 4.15 – State machine for DMC.

4.6.3.1.2 Constraints

The Xilinx MIG IP core demands several constraints in order to work properly. When generating the core, a constraints file is made and the file needs to be modified to match the ML605 evaluation board; all location constraints must match the pin locations given in [21], all lines related to SDA and SCL must be commented out since the onboard temperature sensor mounted on the DDR3 module is not in use by the core. The core takes two different clocks, one as the system clock for the DDR3 module itself and another for the controller. On the ML605 however, only one clock is used so the location constraints of the clock that is not used, and its associated timing specification, must be deleted.

4.6.4 File transfer module

As mentioned in system concept, necessary files must be transferred from the flash to the DDR3 memory. The FTM reads 512 bits of data from the flash, and then writes that data at a predefined location in the DDR3 memory. This continues until all files are read. The start address of each file, in DDR3, can be arbitrary chosen, but the address difference between two neighboring files must be greater than the address space each file consumes.

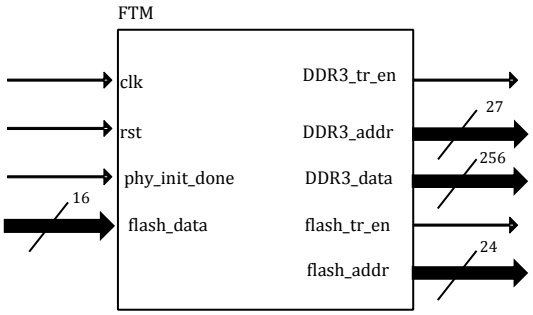


Figure 4.16 - The file transfer module and its ports.

As mentioned in section 4.4.2 there is a significant read access delay. The transfer module must therefore run at a much lower frequency than rest of the system. This does not reduce the performance of context switching as the system accesses the flash memory at the start-up phase only.

Transferring data from flash to DDR3 begins when the `phy_init_done` has been asserted by the MIG, and is based on four counters. The counters are listed below.

1) Transfer_counter_18
2) Transfer_counter_36
3) Start_up_delay_counter
4) No_flash_data_counter

Transfer_counter_18 counts continuously from 0 to 17, when it becomes 17 it starts over from 0 again at the next clock cycle. When the counter is between 1 and 16 the `flash_tr_en` is asserted, and de-asserted when the counter is either 0 or 17. This translate to that the FTM reads 16 words from the flash, waits for two clock cycles, and then reads another 16 words.

The FTM must read 32 x 16-bit before the data can be written to the DDR3. Another counter is therefore necessary; transfer_counter_36 counts from 0 to 35. When the counter is between 0 and 2 the `DDR3_tr_en` is asserted and data is written to the designated address.

Start_up_delay_counter is different. It is active only at start-up. The counter inhibits writing to the DDR3 memory at the beginning of file transfer since there is not any data to be written. Figure 4.17 illustrates the problem. The upper value of the counter is arbitrary, but it must be above 36 to allow for reading of 512 bits from the flash memory.

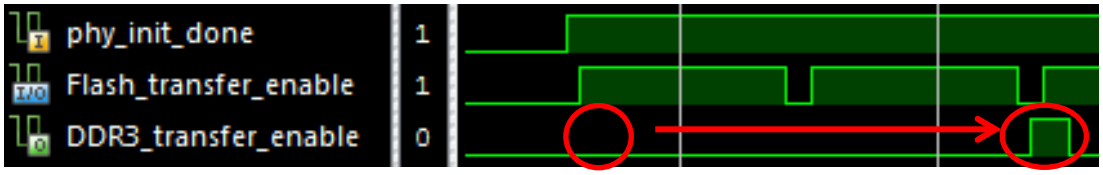


Figure 4.17 - Start-up counter inhibits writing invalid data to DDR3 at the beginning of file transfer.

No_flash_data_counter counts clock cycles when there is no reading from the flash, i.e. the counter is only active when the `flash_tr_en` is de-asserted. This is necessary in-between files. If the size of a file is a multiple of 16, a gap of invalid data appears when changing from one file to another. Writing to DDR3 must be prevented in this case. Figure 4.18 illustrates the problem. If the file size is not a multiple of 16 then the last read data fills that gap with valid data, and writing to the DDR3 is allowed, as seen in figure 4.19.

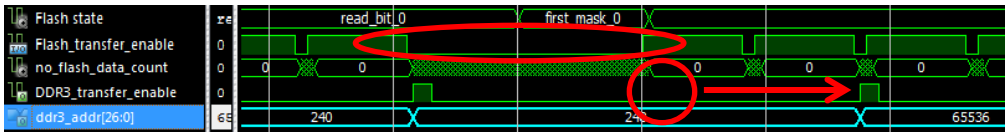


Figure 4.18 - Invalid data between files.

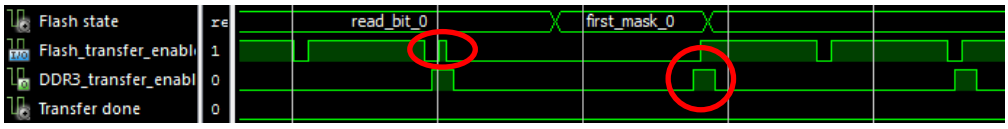


Figure 4.19 - Valid data between files.

The flash address it updated once every clock cycle as long as the `flash_tr_en` is asserted. Address is halted when the enable is de-asserted. The DDR3 address is updated when the transfer_counter_36 is 0. The issue with changing files applies also for the DDR3 address update. The address is therefore additionally dependent on the no_flash_data_counter.

4.6.5 Bitstream manipulator module

In order to create a new partial configuration file that includes the register value from the previous run, the initial bit file must be modified before it is sent to the partial region again. This modification is done by the BMM. The concept of the bit manipulator is that it reads 512 bits of data from each of the files, stores that data in local registers, then applies the modification equation from equation 2.1. The final step is to write the modified data at a new predefined address location in the memory. Figure 4.20 illustrates the concept, the skew of input register relative to each other represents that the reading of data from memory is performed in sequence. This process repeats until the final address of the files has been reached. The modification process starts when the context switch main controller asserts either PRM_0_run signal or the PRM_1_run signal. When the partial module A is running, indicated by the PRM_0_run signal, the creation of the new partial configuration file for partial module B is done and vice versa. The duration of the modification process determines minimum run time of the partial modules.

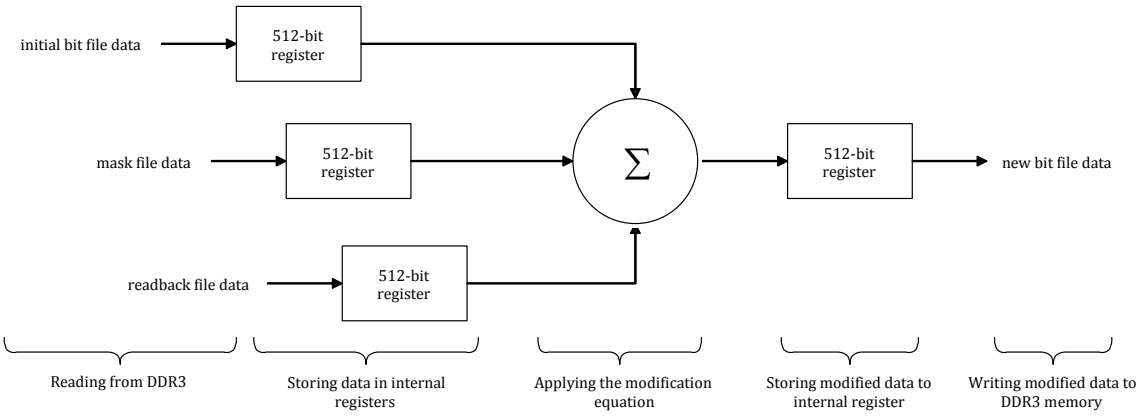


Figure 4.20 - Process of creating new partial configuration file.

The module must know when to stop reading from the memory and the end address for each file must therefore be determined. The static bit file is of known size, and determination of where the other files can reside in the flash memory is therefore possible. Since the memory space of the DDR3 memory differs from the flash memory, the file end address for each file must be changed compared to the file end address in the flash memory. The start address of these files may be chosen arbitrarily. The end address can be calculated, either on the basis of the file size, or, as it is implemented, on the basis of the start and end address of that particular file in the flash memory. Equation 4.1 shows how the address is calculated.

$$DDR3\ end\ addr = \left(floor\left(\frac{flash\ end\ addr - flash\ start\ addr}{32}\right) * 8\right) + DDR3\ start\ addr \qquad 4.1$$

The function CALC_DDR3_END_ADDR performs the calculation as seen by the code excerpt in figure 4.21. The flash memory takes 16 bits of data at one address location and the DDR3 memory takes 512 bits of data at one address location. The flash address is incremented by one each time and the DDR3 address is incremented by eight each time. That means that for each 32 increments of the flash address the DDR3 address is incremented once. Based on this the file end address in the DDR3 memory space can be calculated by the equation 4.1. As mentioned in chapter 2.6.6.1, the mask file and the configuration file contains data that are not of importance for the creation of new configuration file; header data and commands. So when reading both initial configuration file and the mask file from the DDR3, the start address for these files must be set accordingly.

The **mod** operator checks whether the address is a multiple of 32, if it is then the address is then calculated in accordance with the equation. Otherwise, the address is decremented by one until it is a multiple of 32.

```
result  := flash_end_addr - flash_start_addr;

   for i in 0 to 31 loop
     if result mod 32 /= 0 then
         result := result - 1;
     else
        DDR3_end := ( (result / 32) * 8) + DDR3_start_addr;
        exit;
     end if;
   end loop;
```

Figure 4.21 - Calculating the DDR3 file end address.

4.6.5.1 Creating new configuration file

The modification equation 2.1 is realized in hardware as a single signal assignment shown in figure 4.22. The modification process is controlled by a FSM which controls the address and read enable signal for the DDR3, and the actual modification. Figure 4.23 shows the algorithmic state machine (ASM) chart of the FSM.

```
new_partial_bit  <= ( (readback and mask) or (initial and (not mask) );
```

Figure 4.22 – Equation 2.1 realized in hardware.

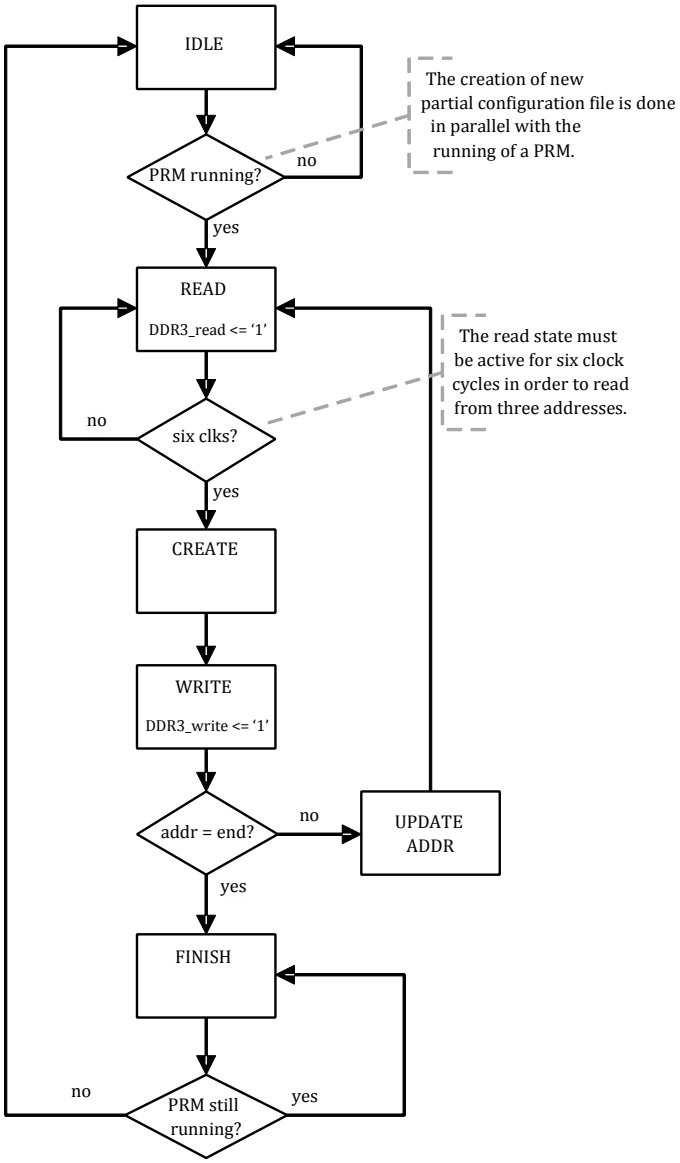


Figure 4.23 - Simplified ASM of the bitstream manipulator FSM.

The `DDR3_read_enable` signal must be held high for six clock cycles in order to read from three address locations. Since the module performs three reads in a row it is necessary to have a counter the counts the number of `app_rd_data_end` assertions so the data is transferred to the correct register. The counting is done by the `read_counter`. If the `read_counter` is 1 it means that the initial bit file has been read, if the counter is 2 then the mask file has been read, and finally, if the counter is 3 the readback file has been read.

The saving of the first 256 bits of the read data to the internal registers happens when the MIG user interface asserts the `app_rd_data_valid` and when the `app_rd_data_end` is asserted the last 256 bits are saved. There is a delay of one clock cycle from assertion of the last `app_rd_data_end` to the actual saving of data to the internal register. It is therefore necessary to wait one clock cycle after the de-assertion of `app_rd_data_end` to apply the modification equation to the registers. There is a counter that is activated when the `read_counter` has counted to three,

`commence_merge_counter`. This counter also counts to three before the equation is applied. This is done to ensure that all three internal registers contains the correct data before applying the equation. The writing of the modified data happens at the next clock cycle and the writing takes two clock cycles. After the writing of modified data there must be a delay of one clock cycle before the reading of the three files can begin again, this assures that the `app_rdy` in the MIG user interface is given enough time to be asserted and a new command is accepted. During this delay the internal addresses are updated.

4.6.6 Data width converter

As mentioned earlier the ICAP data bus is 32 bits wide and the DDR3 memory data bus is 256 bits wide. It is therefore necessary to have a data bus width converter. This module has two enable inputs and two status outputs in addition to two data inputs, one for each width, and two data outputs, also one for each width. The converter works on the same principle as the converter in the transfer module; a counter controls the conversion. Note that when sending data to the ICAP, i.e. context restore, the reading of data from DDR3 memory is 8 times slower than the frequency of the ICAP. The same happens for writing to the memory when retrieving data from the ICAP, i.e. context save.

4.6.7 ICAP instruction memory module

This module was not implemented, but it should be easy to do so. It is just a small memory module that contains the command sequences necessary to perform a readback capture and PRR reconfiguration. The memory can be split into two smaller modules in order to have better control of the reconfiguration process, one for the CS process, and one for CR process, but that is up to the user to decide. The IIMM should be controlled by the CSMC. If two or more PRRs are implemented the FAR and FDRI/FDRO word count must be altered according to which PRR is about to be reconfigured. The memory modules should, in that case, be writeable.

5 Simulation and measurements

5.1 Chapter overview

This chapter gives a description of the simulation environment and how the simulation was performed. Section 5.2 gives a detailed description on the set-up of the simulation. Section 5.3 takes a closer look at how the DMC and DDR3 memory responds to read requests from the BMM. Measurements and results are given in section 5.4, this section also includes some equation needed to calculate parameters related to configuration frames and timing. The results are plotted in a graph and data extrapolation is used to give a precise timing estimate beyond simulation. The final section compares the simulated results width previously conducted experiments from other relevant work mentioned in chapter 3.2.

5.2 Simulation set-up

The Xilinx MIG IP core provides a batch file for generating simulation environment for both Xilinx' own simulation tool iSim, and for the standalone simulation software ModelSim from Mentor Graphics. The simulation environment includes a testbench, and a simulation model of the DDR3 memory that corresponds to the MT4JSF6464H-1G1 from Micron mounted on the ML605. In addition the core provides an example design, shown in figure 5.1**.** This design consists of the controller itself (Memc_ui_top), a clock and reset manager (infrastructure) and a data traffic generator. The testbench instantiates the example design, and makes simulation without a user design possible.

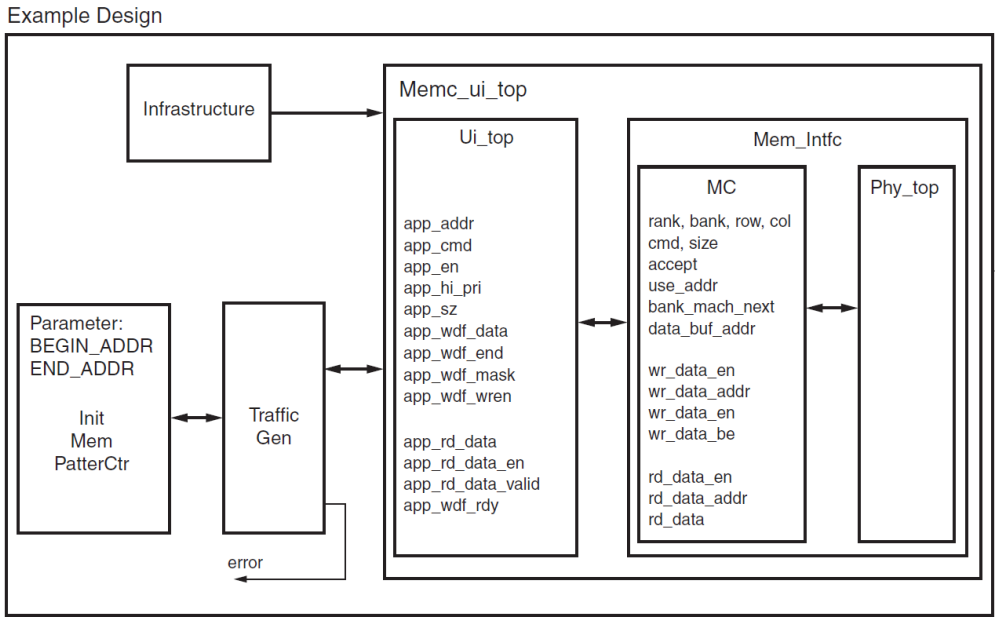


Figure 5.1 - Example design provided by the MIG IP core [27].

The example top file was modified to exclude the traffic generator, and to include the modules described in chapter 4.6**,** except for the data width converter, flash controller and the context switching main controller. The converter was not included since ICAP was not instantiated. The simulation testbench was also modified to simulate the `PRM_A_run` signal since the context switching main controller was not included in the simulation. In addition, data from the flash was generated by a random number generator function. The pseudorandom number generator takes two seed numbers as arguments and generates a random 16-bit number based on the seeds. In order to generate different numbers each time the function is called, i.e. each clock cycle, the seeds are changed every time. The seeds are initialized to 5618 (0x15F2) and 12300 (0x300C), and they are accumulative added by 347 and 1097 respectively. These numbers are arbitrary chosen.

There is no simulation model of the flash memory. However, the memory controller was tested in real using the ML605 evaluation board. The eight MSBs of the flash data bus were outputted onto user LEDs on the board, a test file was uploaded to the flash memory through the iMPACT tool and address was controlled by onboard DIP switches. It was confirmed that the LEDs showed the eight MSBs of the 16-bit words in the test file. The read access time, and hence the reading speed, mentioned in chapter 4.4.2, was not tested.

Since readback of captured data was not available, data from the initial bit file for the other partial module was used instead. This was done by changing the address location of the readback file to the location of the initial bit file of the other PRM. The testbench needed only to simulate the assertion of the reset and the `PRM_A_run` signals.

The provided testbench has a long list of parameters that is passed on to the MIG IP core, but there are only a few parameters related to simulation [27]. The DDR3 model starts initialization and calibration immediately after de-assertion of reset and this process lasts for about 220 µs [27]. This is impractical for testing purposes as simulating that amount of time usually takes up to an hour and a half. The most relevant parameter and its available options are listed in table 5.1 below.

Table 5.1 - Available options for reducing the simulation time [27].

Simulation parameter	Options	Description
SIM_BYPASS_INIT_CAL	OFF	Regular simulation with normal initialization and calibration.
	FAST	Bypasses initialization and performs an abbreviated calibration sequence.
	SKIP	Bypasses initialization and skips the calibration sequence. Various timing relations between the FPGA and the memory are fixed.

The ‘SIM_BYPASS_INIT_CAL’ parameter was set to ‘skip’ which reduces the simulation duration significantly. With this option the `phy_init_done` is asserted after just 15 µs, which corresponds to approximately 8 minutes in real time.

5.3 DMC and BMM simulation

When performing data read from the DDR3 memory there is a delay of 23 clock cycles from the initial assertion of the enable signal to the output of data, as seen in figure 5.2. When reading from several address consecutively, as the figure shows, the distance *between* outputs is only 9 clock cycles. This means that reading data from the three files takes in total 47 clock cycles (23 + 2 + 9 + 2 + 9 + 2), which corresponds to 235 ns.

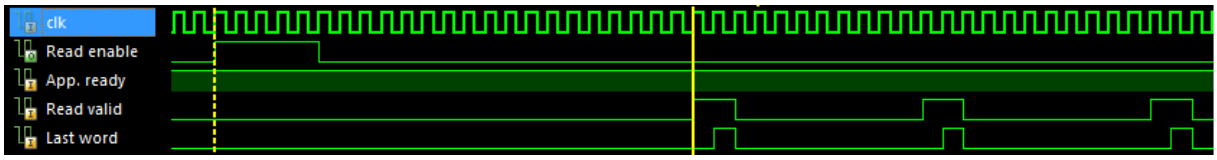


Figure 5.2 - Delay from read command is given till the data is available.

This applies however, only for the very first read, because the MIG de-asserts the `app_rdy` signal for 8 clock cycles after the read, as can be seen in Figure 5.3. The initial delay increases to 36 clock cycles, and the total delay increases to 59 clock cycles, which corresponds to 295 ns.

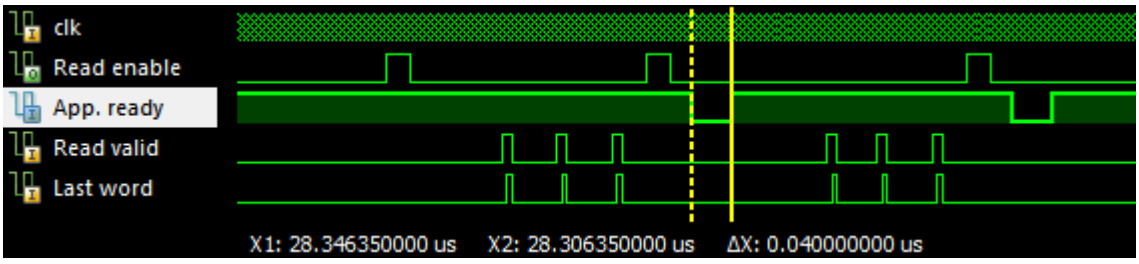


Figure 5.3 – De-assertion of app_rdy after reading from three addresses.

As mentioned in chapter 4.6.5, storing of data into local registers is done in a sequence, and that a counter keeps track of which data belongs to which register. Figure 5.4 shows that the data is stored in local registers accordingly. Note that the equation 2.1 is applied immediately after the last read.

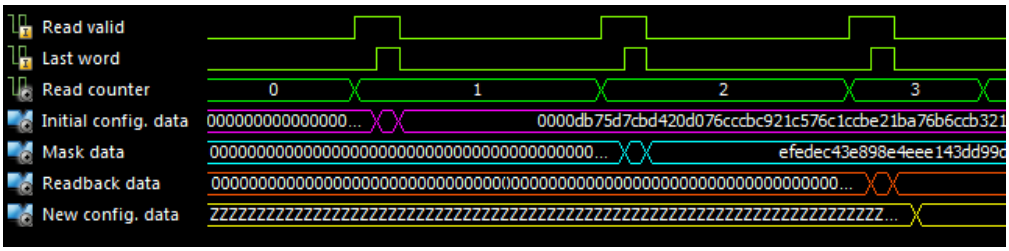


Figure 5.4 – Storing data in local registers.

5.4 Measurements

Four different file sizes, ranging from 1kB to 15kB, were simulated and the time to completion of the generation of new partial configuration file was recorded. The measurements were performed by measuring the time between the idle state and the finish state of the bit manipulator FSM, as shown in figure 5.5 below.

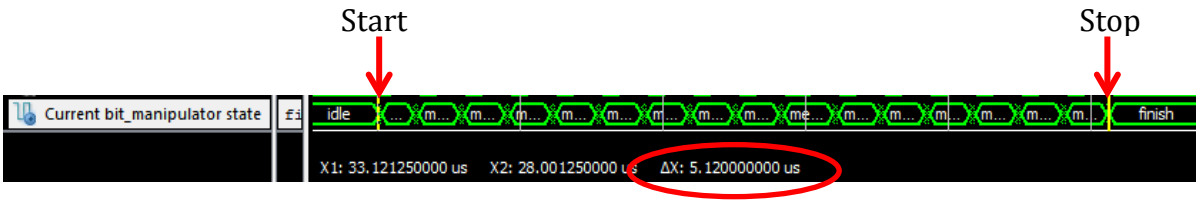


Figure 5.5 - Measurement method.

Table 5.2 shows the results for the four files tested. The file size has also been converted into configuration frames by the equation 5.1, where F_{kB} denotes how many frames there are pr. kB.

$$F_{kB} = \frac{kB}{frame\ length} \qquad 5.1$$

As mentioned in chapter 2.6.2, the length of a configuration frame for a Virtex-6 device is 2592 bits. This gives 3.16 frames pr. kB.

Table 5.2 - Meausrement results.

File size [kB]	File size [frames]	T_{create} [µs]
1	3.16	**5.12**
2	6.32	**10.29**
8	25.28	**41.65**
15	47.40	**77.38**

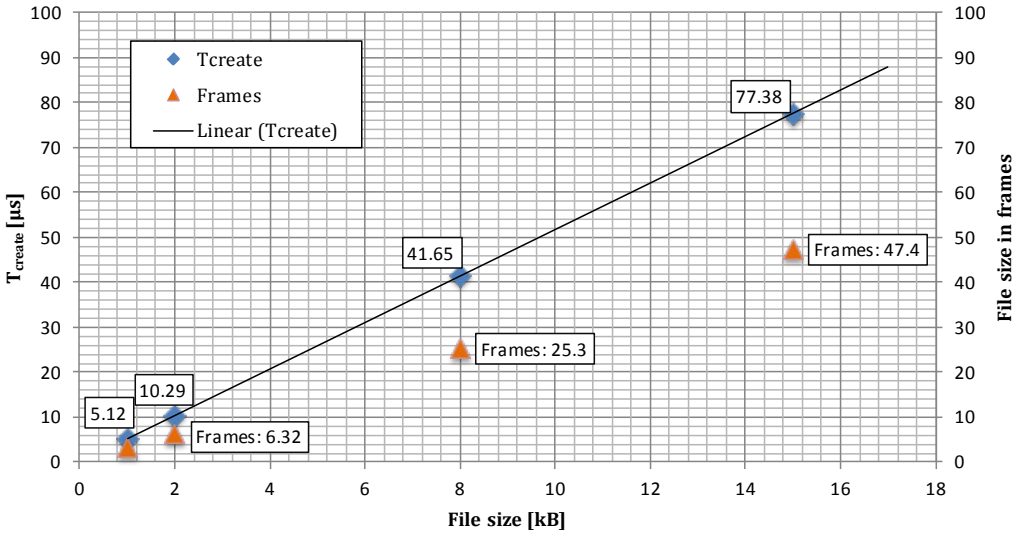


Figure 5.6 – Plot of the result from the simulation data.

As seen by the trend line in figure 5.6, the data is approximately linear, and that property is used to extrapolate the data beyond simulation. The graph also shows the calculated frames. File sizes of 64 kB and 128 kB is used as data points in the extrapolation. Table 5.3 shows the result of the extrapolation, and figure 5.7 shows a plot of the extrapolated data.

Table 5.3 - Extrapolated data.

File size [kB]	File size [frames]	T_{create} [µs]
64	202.27	334.04
128	404.54	668.21

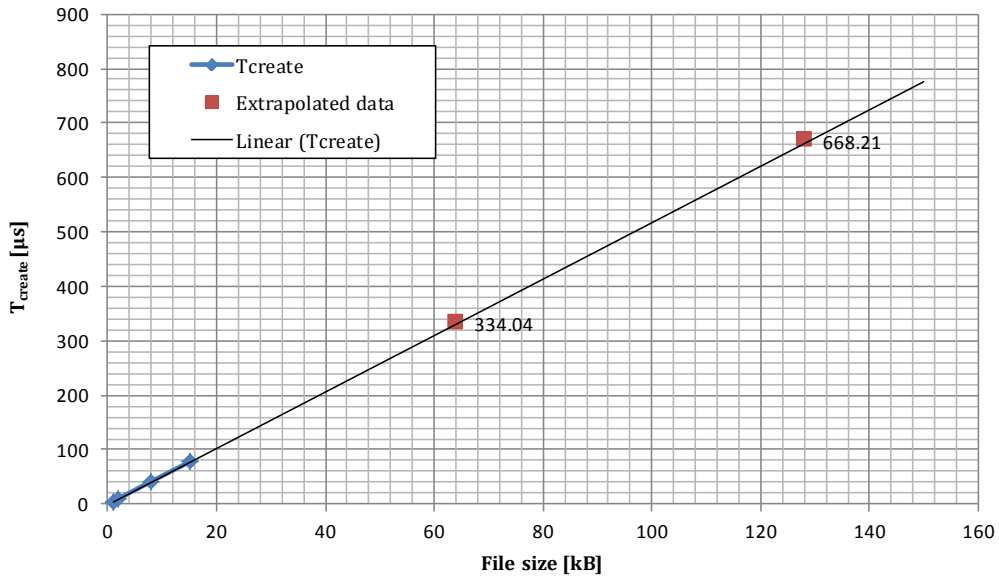


Figure 5.7 - Plot of the extrapolated data.

Equation 5.1 can be used, in conjunction with T_{create}, to derive the time it takes to generate a new frame, as shown in equation 5.2, where T_{frame} denotes the time to create a single frame.

$$T_{frame} = \frac{T_{create}}{F_{kB}} \quad 5.2$$

And we can also find how many frames can be created per µs, as per equation 5.3 where $F_{\mu s}$ denotes frames per µs:

$$F_{\mu s} = \frac{F_{kB}}{T_{create}} \quad 5.3$$

By applying a file size of 1 kB into the equations we get that T_{frame} is 1.62 µs, and $F_{\mu s}$ is 0.62 frames per µs. To put this in perspective, XAPP883 [20] provides a partial reconfiguration example, and the PRM has a length of 1673 frames. Generating new configuration file for that PRM would take only 2.71 ms. Another example; to generate a full configuration file for the Virtex-6 device used in this thesis (28488 frames) requires only 46.15 ms.

To calculate the data rate of the bit file creation process we take the data from table 5.2 and divide the file size with the corresponding T_{create}, as shown in table 5.4. Average data rate of the four file sizes are 1514 Mb/s.

Table 5.4 - Data rate of the creating new bit file process.

File size [kB]	Data rate [Mb/s]
1	**1600**
2	**1592**
8	**1573**
15	**1588**

5.5 Comparison

Table 5.5 shows measurement results from the relevant research from chapter 3.2. Only two of the papers provided comparable data.

Table 5.5 - Measurements from relevant research

Author	HW/ CPU	Device	PRR file size [kB]	Reconfiguration time [µs]		
				CS	Bitstream manipulation	CR
[18]	CPU	Virtex-4	3.68 / 158.83[4]	60.11	~1000[5]	536.90
[14]	CPU	Virtex-5	0.96	6330	7220	5610

[4] Smaller readback file compared to the configuration file since only 'state information' frames are read
[5] Not recorded, but the authors states: 'slightly more than 1 ms (…)'

As the measurements performed in this thesis focuses on the bitstream manipulation process, [18] is excluded in the comparison as the data is not recorded. Therefore there is only one paper that the result from the simulation is compared to. It was reported in [14] that the process of creating a new partial configuration file, or merging as the authors named it, lasted for a few milliseconds, see table 5.6. The authors used columns, frames and flip-flops as unit of measurement. This is not directly comparable to the results shown above since the authors used a Virtex-5 device, where frames are smaller, in terms of bits, compared to a Virtex-6 device. The unit of measure from [14] must therefore be converted into file size as used in this thesis. A Virtex-5 device has a frame length of 1312 bits [8], which leads to 0.16 kB per frame, or ~6.25 frames per kB. The authors also reported near linear results.

Table 5.6 - Results from [14].

V5-frames	bits	kB	T_{merge} [ms]
2	2624	0.32	4.5
4	5248	0.64	5.85
6	7872	0.96	7.22
~6.25	8192	1.00	~7.38
8	10496	1.28	8.57
10	13120	1.60	9.91

The data point marked red, has been calculated based on the other data points As seen in table above, the calculated data point, corresponding to a file size of 1 kB, shows that the proposed system in [14] use approximately 7.4 ms to complete the process of creating new partial configuration file. The system proposed in this thesis is over *three orders of magnitude faster*. This confirms the notion stated in the introduction that a hardware based partial reconfiguration is faster than a system based on a soft core microprocessor.

6 Conclusion and future work

6.1 Chapter overview

This chapter provides a conclusion of the work presented in the thesis. It also gives a few suggestions on how to improve the system in order to increase its functionality and effectiveness.

6.2 Conclusion

Partial reconfiguration with context switching demands that the internal state of a PRM is preserved for later resumption. The preservation of states can be done either by hardware checkpointing, as mentioned in chapter 2.5.2, or by capturing the register values which is then read back from the configuration memory. The latter option was chosen. Creating a new partial configuration file that contains state information is done by manipulating the initial configuration file for the PRM. To modify the file, two other files are necessary, the masking file and the readback file. The mask file is generated by the implementation tool ISE. Modification equation given in 2.1 determines how to manipulate the initial configuration file. Final step is to reconfigure the PRR with the newly created configuration in order to resume the PRM from where it was replaced.

A bitstream manipulator module that generates a new configuration file has been implemented. The mask file and initial configuration file are uploaded from a computer to the onboard flash memory, but since flash is much slower compared to SDRAM these files are transferred to the onboard DDR3 memory. A flash memory controller, file transfer module and DDR3 memory controller have been implemented in order to transfer the files. All data files used by the BMM are stored in the DDR3 memory. The manipulator has direct access to the memory, and hence, acts as a DMA. The DDR3 memory controller is based on the MIG IP core from Xilinx and modified to suit the system.

Previous chapters have given an insight into the theoretical aspect of the subject of FPGA configuration, partial reconfiguration and context switching. A detailed description on how the system was implemented was also given. Since the project was comprehensive for a master project some modules, that is necessary to complete the partial reconfigurable system, was unfortunately not implemented. But the simulation of the implemented modules showed that the bit manipulator, and hence the generation of new partial configuration file, was working as intended.

The system performance can be described, or measured, in two different ways, one is the time to completion of the creation of new partial configuration file, and the other is to determine the data rate at which the bit file creation is performed. The system can create a new partial configuration file, dependent on the file size, within a few milliseconds. Four file sizes were tested and the simulation showed that a file size of 1

kB, the process of creating new partial configuration file was completed within 5.12 µs, and with a file size of 15 kB, the process clocked in at 77.38 µs, see chapter 5.4. This is more than *three orders of magnitude faster* compared to the system presented in [14], which used 7.38 ms for a file size of 1 kB. To put this in perspective, XAPP883 [20] provides a partial reconfiguration example, and the PRM has a length of 1673 frames. Generating new configuration file for that PRM would take only 2.71 ms. The data rate has been calculated to be between 1573 and 1600 Mb/s for the measured data set.

6.3 Future work

The system presented is based on a single island style partitioning, and the two PRMs run for an equal amount of time. The design is kept as simple as possible in order to minimize complexity. However, the system should be modified, optimized and improved in order to increase its functionality and effectiveness. A few examples of improvements are given below.

6.3.1 Complete the design

The system is not complete, as explained in the introduction. It is therefore necessary to finish the design in order to perform complete tests, including CS and CR, and to compare those results with other work. To perform these tests, PRR must be implemented and two or more PRMs must be available. What need to be implemented, to complete this system, is the CSMC and the IIMM, and to connect them to the other modules of the system.

6.3.2 Advanced CSMC with a scheduler

A PR system may have demands that go beyond the limitations mentioned above. These demands may include, among others:

- Three or more PRMs
- Prioritized PRMs. A PRM can have a higher priority compared to other PRMs.
- Different run-time. Not all PRMs are required to run for the same amount of time.
- Relocation. A PRM must be able to relocate to another PRR.
- Location requirements due to signal latency or other dependent components.

These requirements demand a smarter CSMC, and an advanced CSMC must therefore be able to handle the required demands efficiently. It must act more like a scheduler that keeps track of which PRMs are currently running, how long their respective run-time is, which has the higher priority, the PRMs constraints if any, availability of PRRs, and other demands or constraints.

6.3.3 Increase DDR3 memory controller efficiency

A limiting factor of the performance of the file creation process is that the ML605 have a 200 MHz system clock. The mixed mode clock manager (MMCM) in the 'infrastructure' module then produces a 400 MHz clock for the memory module, which gives a data rate of 6.4 GB/s. The maximum frequency of the DDR3 module that the MIG IP core is designed for is 533 MHz [28], which gives a data rate of 8.5 GB/s. By using a higher clock frequency, i.e. an FPGA device with better speed grade, a higher data rate for the creation process may be achieved.

6.3.4 Use Enhanced ICAP hard macro

As mentioned in chapter 2.6.1.5 a partial reconfigurable system uses ICAP to reconfigure the PRR. This interface has restrictions in data rate, 100 MHz and 32-bit data bus giving a data rate of 3200 Mb/s, which limits the reconfiguration process severely. Chapter 2.6.1.5.1 discussed attempts at overclocking the ICAP to increase its throughput. By implementing the enhanced ICAP proposed by [16], which has a data rate of up to 17600 Mb/s at readback, the overall reconfiguration time will be reduced significantly.

7 Bibliography

[1] K. Krzysztof, F. Morgan and K. Kosciuszkiewicz, "SeReCon: A Secure Dynamic Partial Reconfiguration Controller," in *Symposium on VLSI*, 2008.

[2] Y. Hori, A. Satoh og T. Katashita, «Tackling the security issues in FPGA partial reconfiguration with physical unclonable functions,» i *International Conference on Reconfigurable Systems and Algorithms*, 2012.

[3] A. S. Zeineddini and K. Gaj, "Secure Partial Reconfiguration on FPGAs," in *Proceedings. 2005 IEEE International Conference on Field Programmable Technology*, 2005.

[4] C. Maxfield, The Design Warrior's Guide to FPGAs - Devices, Tools and Flows, Elsevier, 2004.

[5] Xilinx Inc., *Virtex-6 Configurable Logic Block User Guide, UG364,* 2012.

[6] Xilinx Inc., *Virtex-6 Clocking Resources, UG362,* 2014.

[7] Xilinx Inc., *Virtex-6 Configuration User Guide, UG360,* 2013.

[8] Xilinx Inc., *Virtex-5 FPGA Configuration Guide, UG191,* 2012.

[9] A. H. Fritzell, "A System for Fast Dynamic Partial Reconfiguration using GoAhead: Design and Implementation," 2013.

[10] D. Koch, Partial Reconfiguration on FPGAs - Architecture, Tools and Applications, Springer, 2013.

[11] The Linux Information Project, "Context Switch Definitions," 28 May 2006. [Online]. Available: http://www.linfo.org/context_switch.html. [Accessed 6 March 2014].

[12] H. Simmler, L. Levinson and R. Männer, "Multitasking on FPGA Coprocessors," in *Proceedings of the 10th International Conference on Field Programmable Logic and Application*, 2000.

[13] D. Koch, C. Haubelt and J. Teich, "Efficient hardware checkpointing: concepts, overhead analysis and implementation," in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Filed prgrammable gate arrays*, 2007.

[14] A. Morales-Villanueva and A. Gordon-Ross, "On-chip context save and restore of hardware tasks on partially reconfigurable FPGAs," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2013.

[15] Xilinx Inc., *Zynq 7000 All Programmable SoC Technical Reference Manual, UG585,* 2014.

[16] S. G. Hansen, D. Koch og J. Tørresen, «High speed partial run-time reconfiguration using enhanced ICAP hard macro,» i *2011 IEEE International Symposium on Parallel and Distributed Processing Workshop and Phd Forum (IPDPSW)*, 2011.

[17] H. Kalte and M. Pormann, "Context saving and restoring for multitasking in reconfigurable systems," in *International Conference on Field Programmable Logic and Apllications*, 2005.

[18] K. Jozwik, H. Tomiyama, S. Honda and H. Takada, "A novel mechanism for effective hardware task preemtion in dynamically reconfigurable system," in *2010*

International Conference on Field programmable Logic and Applications, 2010.

[19] T. Lee, C. Hu, L. Lai and C. Tsai, "Hardware context-switch methodology for dynamically partially reconfigurable systems," *Journal of Information Science & Engineering,* vol. 26, no. 4, pp. 1285-1309, 2010.

[20] Xilinx Inc, "AR# 34740: MIG Virtex-6 DDR2/DDR3 - PHY Initialization and Calibration," 15 12 2012. [Online]. Available: http://www.xilinx.com/support/answers/34740.html. [Accessed 30 March 2014].

[21] Xilinx Inc., *ML605 Hardware User Guide, UG534,* 1.8 ed., 2012.

[22] Xilinx Inc., *Virtex-6 Family Overview, Product Specification, DS150,* 2012.

[23] Xilinx Inc., *Platform Flash XL High-Desity Configuration and Storage Device, DS617,* 2010.

[24] Micron / Numonyx, *Numonyx StrataFlash Embedded Memory (P30) datasheet,* 2008.

[25] Micron, *DDR3 SDRAM SODIMM MT4JSF6464H-512MB datasheet,* 2007.

[26] Xilinx Inc., *ML605 schematic,* 2009.

[27] Xilinx Inc., *Virtex-6 FPGA Memory Interface Solution, UG406,* 2012.

[28] Xilinx Inc, *Virtex-6 FPGA Memory Interface Solution, DS 186,* 2011.