Context Save and Restore of Partial Reconfiguration Regions for Xilinx FPGAs

Marcel Eckert and Dominik Meyer and Bernd Klauer
Dept. of Computer Engineering

Helmut-Schmidt University Hamburg
eckert@hsu-hh.de dmeyer@hsu-hh.de klauer@hsu-hh.de

***Abstract*—Preemption is an important technique used by modern operating systems to achieve multitasking. When FPGAs need to be integrated into the preemptive multitasking mechanisms of an operating system, a context switch of the design configured into the FPGA is required. Several proposals on how to take advantage of such a context switch by saving and restoring the internal state of an FPGA have been presented in the past. Unfortunately, these discussions give neither much of the technical details of the save and restore mechanism itself, nor are they available for modern FPGAs like Ultrascale-Series. This paper summarizes the technical details of these officially unsupported techniques to save and restore the internal state of modern Xilinx FPGAS (namely 7-Series and Ultrascale-Series) which are currently scattered over several other publications and technical manuals.**

***Index Terms*—FPGA, context switch, ICAP, partial reconfiguration**

I. Introduction

Reconfigurable computing, which combines reconfigurable technology in form of Field Programmable Gate Arrays (FPGAs) with conventional computers and servers is gaining more and more popularity (refer to Amazons F1 elastic cloud or Microsofts Project Brainwave). Furthermore several attempts to integrate reconfiguration technology into the hardware management and abstraction tasks of an operating system exist. See [4] for an overview on operating system concepts for reconfigurable computing. One of the key ideas is to include reconfigurable resources into the preemptive mechanisms of the operating system. To enable preemption a context switch (save the current status of a process to restore it later) is necessary. For a reconfigurable resource this means to not only extract the hardware design but to extract the internal state of this hardware design (state of Flip-Flops and RAM contents) additionally.

For reconfigurable resources, there are mainly two distinct approaches to achieve this save and restore capability. The first kind are called ”scan chain techniques”. These embed additional circuitry into a hardware design to allow for extraction and insertion of relevant context bits. Latest publications, using the scan chain approach are [1], [20] or [19]. A survey on early scan chain approaches can be found in [12].

In contrast to scan chain techniques, another approach to achieve context switching capabilities within FPGAs is to use the configuration readback mechanisms of the FPGA itself. In [5] these mechanisms are used targeting Virtex-6 FPGAs, limiting the context to CLB-FlipFlops and block Ram state extraction but lack the readback of distributed RAM. In [17] the configuration readback mechanism is applied to Virtex-5 FPGAs. Several further references exist, e.g. [10], [13], [11], [18] or [14], targeting older FPGA Families. Unfortunately, the aforementioned references focus their discussion on the application, that uses the FPGAs readback and capture capabilities but only give little information on the technical details of the context switch itself. Furthermore, a context save and restore via readback and capture is officially not supported by Xilinx and therefore no corresponding technical manual exists.

Hence, this paper is summarizing the technical details to perform a readback and capture based context switch for modern Xilinx FPGAs (7-Series and Ultrascale Series families) without distracting the reader with the application that is using it. This will disburden feature research from gathering the information related to readback and capture based context switching, which is scattered in several publications and technical manuals. This paper is focusing on partial reconfigurable regions in a single FPGA. However, the presented techniques may easily be adopted and applied to an entire FPGA context switch, initiated by a host system.

The paper is structured as follows: Section II gives background information on bitstreams. Section III presents knowledge to perform configuration readback and state capture. Section IV provides details on how to apply the readback and capture mechanism to achieve the context save and restore for modern Xilinx FPGAs. The feasibility of the presented techniques is proven by testing in Section V. Finally, Section VI gives a conclusion.

II. Partial Reconfiguration Bitstreams

Bitstreams are the final result of a FPGA synthesis and implementation design flow. The bitstream files are suitable to program/ (or to be more precisely) configure the entire FPGA. When a partial reconfiguration design flow is applied, for each partition of the FPGA, bitstreams allowing only to change the corresponding FPGA area (reconfigurable partition) are also generated. These are called partial bitstreams (PR-bitstreams). For details on the differences between a conventional and a partial reconfiguration design flow see [9].

978-1-7281-4770-3/19/$31.00 ©2019 IEEE

interface direction	data	explanation	step summary
write	ffffffff	dummy word	sync sequence
	000000bb	bus width sync word	
	11220044	bus width detect	
	ffffffff	dummy word	
	AA995566	sync word	
	20000000	no-operation (nop) word	
write	28018001	write packet to read IDCODE	IDCODE read sequence
	20000000	nop word	
	20000000	nop word	
read	<IDCODE>	IDCODE answer from FPGA	
write	30008001	write 1 word to CMD	desync sequence
	0000000d	DESYNC command	
	20000000	nop word	
	20000000	nop word	

TABLE I
EXAMPLE BITSTREAM TO READ THE FPGAS ID-CODE.

The remainder of this section presents details on the common internal structure of the bitstream files as a brief summary of the information provided by the family specific configuration user guides [6], [8]. For the reader interested in bitstream details beyond context save and restore, see [16] or [2].

A. General Overview

Xilinx provides two bitstream file formats. A **.bit* and a *.bin file format. The bitstreams can be applied to the FPGA via different configuration interfaces (see [6], [8] for details). Without any loss of generality this paper focuses on the ICAP-interface (Internal Configuration Access Port). Both file formats contain the entire bitstream. The **.bit* format additionally includes header information for the *Vivado Hardware Manager* (the host to FPGA configuration tool). The **.bin* file format only contains the pure bitstream information and is most commonly use to perform partial reconfiguration from within the FPGA itself via the family specific ICAP element.

A bitstream consists of 32 bit wide control and data words. An example bitstream, to read the ID-code of the FPGA is given in Table I. A valid bitstream always starts with a *sync sequence* and is finished with a trailing *desync sequence*. In between theses two sequences, read and write operations, targeting specific registers are performed. These register reads and writes contain control information for the configuration process itself and the configuration data to configure the FPGA. For a detailed list of the available registers and their meaning, see the family specific configuration guides. In this paper, only the registers relevant for partial reconfiguration will be covered.

Table II shows an example for a small partial reconfiguration bitstream targeting an Artix-7 FPGA (XC7A100T-1CSG324C). As a first step, the CRC (Cyclic Redundancy Check) register is cleared and the IDCODE check is performed. This check prevents the application of bitstreams, that were created for another FPGA type. During the *CTL0 setup sequence* several CTL0 bits are set. This setup is undone (values are set back to default) during the *restore CTL0 sequence*. After the *restore CTL0 sequence* the FAR

interface direction	data	explanation	step summary
write	*sync sequence* (see Table I)		
write	30008001	write 1 word to CMD	reset CRC sequence
	00000007	reset CRC command	
	20000000	nop word	
write	20000000	nop word	IDCODE check sequence
	30018001	write 1 word to IDCODE	
	03631093	XC7A100T IDCODE	
write	30008001	write 1 word to CMD	CTL0 setup sequence
	00000000	NULL command	
	3000c001	write 1 word to MASK	
	00000100	MASK data	
	3000a001	write 1 word to CTL0	
	00000100	data for CTL0	
	3000c001	write 1 word to MASK	
	00000400	MASK data	
	3000a001	write 1 word to CTL0	
	00000400	data for CTL0	
write	30008001	write 1 word to CMD	write config data sequence
	00000001	WCFG command	
	20000000	nop word	
	30002001	write 1 word to FAR	
	00420500	address for FAR	
	20000000	nop word	
	30004000	large value write to FDRI	
	5000559d	value for FDRI (21917)	
	????????	config data word 1	
	...	further config data words	
	????????	config data word 21916	
write	3000c001	write 1 word to MASK	restore CTL0 sequence
	00000100	data for MASK	
	3000a001	write 1 word to CTL0	
	00000000	data for CTL0	
write	30002001	write 1 word to FAR	park FAR sequence
	03be0000	park FAR value	
write	30000001	write 1 word to CRC	CRC check
	f5caed84	CRC check value	
write	*desync sequence* (see Table I)		

TABLE II
MINIMAL 7-SERIES PR-BITSTREAM.

(Frame Address Register) is set to the parking position. The *CRC check* finalizes the configuration stream, except for the *DESYNC* sequence. A *CRC check* can either be performed by writing the correct crc-value to the CRC register (as given in the example), or by reseting the CRC (shown by *reset CRC sequence*). For a detailed analysis on when a CRC check is necessary and where it can be replaced by a *reset CRC* can be found in [15].

The configuration of the FPGA itself is done by a *write config data sequence*. It starts with sending the WCFG (Write Configuration) command to the FPGA, followed by setting the Frame Address Register (FAR) to the correct position (more details on the FAR will follow later). The configuration data itself is send via the FDRI (Frame Data Register Input). In the example of Table II, 21917 configuration words are sent to the FPGA. This number results from the size of the partition. A partial bitstream file may contain several *write config data* sequences, depending on the layout, size and contents of the corresponding reconfigurable partition.

Element	7-Series [6]	UltraScale Series [8]
configuration frame	101 words	123 words
CLBs per Frame	50	60
conf. frames per CLB column	28 (interconnect) 8 (content)	58 (interconnect) 12 (content)
conf. frames per BRAM column	128	128

TABLE III
FPGA CONFIGURATION FRAME SIZES

B. Configuration Memory, Configuration Frames, Columns, Rows and Frame Addresses

The configuration information of an FPGA are stored in the configuration memory. This configuration memory is arranged in frames. These configuration frames are the smallest addressable segments of the FPGAs configuration memory. All configuration frames have a fixed size and are addressable by their individual frame address. The size of the frames differ from FPGA family to FPGA family. See Table III for the frame sizes for different FPGA families covered by this paper.

The structure of a 32 Bit wide frame address is as follows:

31:26	25:23	22,21:17	16:7	6:0
unused	block type	row addressing	column address	minor address

The commonly used *block types* are

- **CLB,I/O,CLK** encoded as 000, called CLB type in the following
- **block RAM content** encoded 001, called BRAM type in the following
- **CFG_CLB** encoded 010, called CFG type in the following, only for 7-Series FPGAs

Xilinx FPGAs are structured in rows, as can be seen in Figure 1. The height of a row corresponds with the height of a clock region (label X0Y0 to X1Y3 in Fig. 1). The row addressing scheme differs slightly between the different families (see the corresponding guides for details). Within each row, configuration columns are found, covering the row from left to right; so the height of a column corresponds to the height of a row. In Fig. 1, one such (CLB-)configuration column is highlighted in clockregion X0Y2. Depending on the specific column type (which is different to the frame address type field), a number of configuration frames are addressable with the minor field. The column type can be one of (list incomplete) CLB content, CLB interconnect, DSP, I/O, BRAM content, BRAM interconnect etc. The number of configuration frames per configuration column differs among the column types (see Table III for selected examples). In effect, the FPGAs frame addresses are not continuously increasing.

When the bitstream contains a *write config data* sequence (see Table II), the frame address register (FAR) is set to the starting column address of the corresponding FPGA partition. As long as neighboring (from left to right) configuration columns are affected, the configuration port increases the FAR automatically. If the bitstream needs to skip some configuration columns, several *write config data* sequences, which individually setup the FAR to the appropriate starting configuration column, are embedded into the bitstream. The configuration data within one such *write config data* sequence is called configuration block in the following.

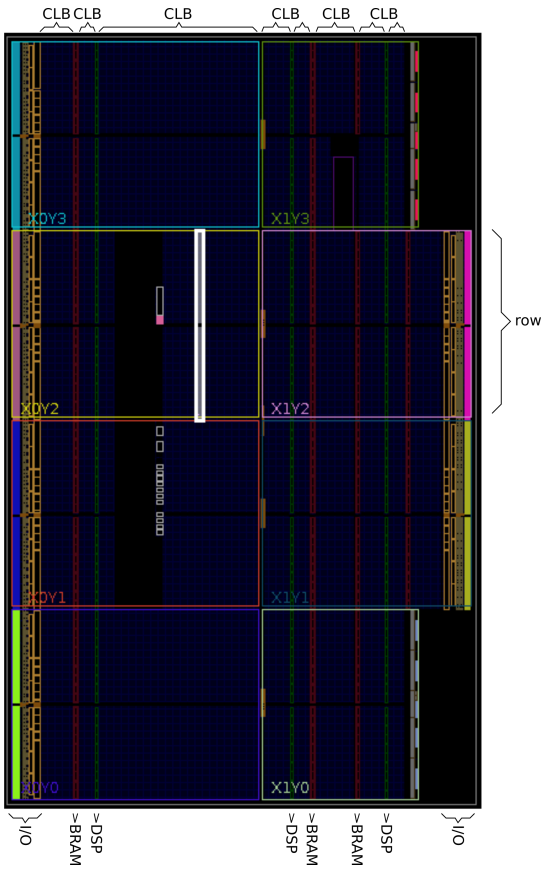


Fig. 1. Artix7 example Fabric Layout

III. INITIAL VALUES, READBACK AND READBACK CAPTURE

A. Initial Values

If a design does not only contain combinational logic, but additional storage elements, it is good design practice to include a set/reset signal that allows to bring the design into an intended starting state. When designing for FPGAs, the intended starting state can be embedded into the bitstream. In consequence, after configuring the FPGA with such a bitstream, the design is already in the intended starting state without applying a user set/reset. To get into the details, examine the following VHDL code snippet, describing an 8 Bit register.

```
-- register signal declaration
signal regVal : std_logic_vector(7 downto 0) := "01101001";
[...]
-- process for register inference
process(clk)
begin
  if rising_edge(clock) then
    if reset = '1' then
      regVal <= (others => '0');
    else
      regVal <= Data_in;
    end if;
  end if;
end process;
```

Two different initializations can be seen here. One at line 8/9, which is the common way to initialize the register via a user controlled set/reset line. The other one at line 2, is applicable to FPGA bitstreams. After configuring an FPGA with a bitstream generated from this code snippet, regVal will contain the initial value 01101001, even if clock remains disabled.

This initialization is achieved by embedding the initial values of the storage elements (namely CLB Flip-Flops, distributed RAM[1] and block Ram) into the configuration data. For block RAM this is directly done via the BRAM content configuration frames. For distributed RAM it is also directly done, given that the *GLUTMASK_B* Bit in the CTL0-register is set (see Table II, data word 00000400 in the *CTL0-setup* sequence). For CLB Flip Flops an additional step is required. After writing the configuration bitstream to the FPGAs configuration memory, the initial values of the CLB FlipFlops need to be loaded into the corresponding Flip Flop. The CLB Flip Flop initialization is done by issuing a GRESTORE to the FPGA, which is in essence an assertion and deassertion of an FPGA global set/reset line. Be aware that this has nothing to do with a set/reset line used in the user design. As the GRESTORE mechanism operates differently for different FPGA families, subsequent sections provide additional information for the specific FPGA families.

B. Readback and Readback Capture

Till now, this paper only discussed the mechanisms to write configuration data (and thereby initial values) into an FPGA. Xilinx FPGAs offer the possibility to read the currently used configuration back. This mechanism is provided for verification issues. This readback is done via bitstreams, which are structured nearly the same way as when performing a configuration write. The main difference is that the *write config data* sequences (see Table II) have to be replaced by *read config data* sequences as shown in Table IV.

interface direction	data	explanation	step summary
write	30008001	write 1 word to CMD	read config data sequence
	00000004	RCFG command	
	20000000	nop word	
	30002001	write 1 word to FAR	
	00420500	address for FAR	
	20000000	nop word	
	28006000	large value write to FDRO	
	5000559d	value for FDRO (21917)	
write	*n times nop word*		
read	????????	config data word 1	
	...	further config data words	
	????????	config data word 21916	

TABLE IV
READ CONFIG DATA SEQUENCE

The differences are 1) instead of starting with a *WCFG* command, a *RCFG* (read configuration) command is issued; 2) instead of specifying the number or words to read by interacting with FDRI (frame data register input), use FDRO (frame data register output); 3) between setting up the amount of data words to be read (write to FDRO) and the readback of the data words, a number of n *nop words* has to be inserted (for 7-Series $n = 32$, for Ultrascale-Series $n = 64$) and finally 4) instead of writing the configuration data, it is read back.

In addition to exchanging *write config data* sequences with *read config data*, the final *CRC check* has to be replaced by a *reset CRC* sequence. Furthermore, a *SHUTDOWN* command has to be inserted into the readback bitstream right after the *sync sequence* and a *START* command has to be inserted right before the *desync* sequence.

If it is necessary to not only read back a designs (circuitry) configuration, but also the current values of the storing elements (also called *context*) an additional step is required. This step is summarized as *capture* sequence. The capture sequence differs for the different FPGA families. Therefore the *capture* sequences details are further explained in the family specific sections.

C. 7-Series Specifics

The minimal PR-bitstream shown in Table II will not start with the appropriate initial values of the CLB-FlipFlops (distributed RAM (CLB LUTs) and block RAM is correctly initialized), as the bitstreams lacks the *GRESTORE* command (see Section III-A). For 7-Series devices the global set/reset line of the FPGA will be asserted and de-asserted by the *GRESTORE* command. Without any further preparations, the global set/reset would be applied to the entire FPGA and not only to the intended reconfigurable partition. For this purpose, the *CFG_CLB* type frames exist (see Section II-B). The purpose of theses CFG type frames is to mask/unmask the effect of the FPGA global set/reset line. In consequence, a PR-bitstream for a 7 series device which uses the *GRESTORE* command always contains a *write config data* sequence, configuring the entire *CFG/CLB* mask of the overall FPGA. Therefore the *CFG_CLB* configuration block always has the same size for a specific FPGA, independent of the partial reconfigurable area itself.

Whether a 7-Series PR-Bitstream contains the *GRESTORE* command or not is dependent on the *RESET_AFTER_RECONFIG* flag of the corresponding *pBlock*s. If *RESET_AFTER_RECONFIG* is set for the reconfigurable partition, the PR-bitstream will contain the *GRESTORE* command and initializes the Flip-Flops of the CLBs. For further details on *pBlocks* and *RESET_AFTER_RECONFIG* see [9].

Since Vivado 2016.1, *blanking*[2] bitstreams are no longer required. However, the *blanking* functionality is now embedded into 7-Series PR-bitstreams by applying two (slightly different) configurations to each configuration frame. In consequence the PR-bitstreams contains two *write config data* sequences with the same starting FAR and equal length. The first one is the *blanking* configuration, the second one the intended

[1]Distributed RAM is based on the LUTs of a CLB.

[2]A blanking bitstream was recommended to be loaded in between the changing of the configuration of a partial reconfigurable area for Vivado Version before 2016.1. These blanking bitstream avoided unintended (but sporadic) glitches when exchanging PR-configurations.

configuration, also containing the desired initial values for the storing elements. On a readback or readback capture procedure one will only get the final configuration sequence (the second *write config data* sequence).

D. Ultrascale-Series Specifics

With the introduction of the Ultrascale Series, the capabilities within PR-regions of the Xilinx FPGAs changed drastically, regarding the design elements that can be used inside reconfigurable modules. For example, it is not possible to include clock management logic like (BUFG, MMCM, PLL etc.) inside reconfigurable modules (see [9] Chapter 6 and 7) for the 7-Series (or earlier) FPGAs. As a further consequence to the changes introduced by the Ultrascale-Series, the *CFG_CLB* type frame and *GRESTORE* related masking mechanisms of the 7-Series (see previous section) are no longer required. Furthermore, Ultrascale-Series PR-bitstreams always contain the *GRESTORE* command, as *RESET_AFTER_RECONFIG* is always activated (and is not deactivateable) for Ultrascale reconfigurable partitions.

When performing a PR-design flow, for each reconfigurable variant of a reconfigurable partition, one will get two *.bit-files (and also *.bin-files). A configuration file and a clearing file. The configuration bit.file is written into the FPGA, when the reconfigurable module is to be used. *The clearing bit-file prepares the device for the delivery of any subsequent partial bitstream for that reconfigurable partition by establishing the global signal mask for the region to be reconfigured. Although the existing module is technically not removed (the current logical module remains), it is easiest to think of it this way [9].*

When analyzing both, the configuration and the clearing bit-file of an Ultrascale reconfigurable module, it is found that both essentially are a concatenation of two bitstreams.

The configuration bit-file is composed of the configuration bitstream (containing the designated design and also the embedded initial values for the storing elements) and the lock-bitstream (explanation will follow later). The configuration bitstream finishes with the *DGHIGH/LFRM* command which is *activating all interconnects [9]*[3].

The clearing bit-file is composed of the unlock-bitstream (the inverse of the lock-bitstream inside the configuration bit-file) and the disconnect bitstream. The disconnect bitstream essentially contains the *AGHIGH* command: *places all interconnect in a High-Z state to prevent contention when writing new configuration data [9].*

The lock- and unlock bitstreams set (lock) or unset (unlock) specific bits in specific configuration frames of the reconfigurable partition. When further analyzing the position of the designs storing elements (distributed RAM and CLB FFs) with Vivado floorplanner it can be seen that these configuration frames are targeting exactly the CLBs of those storing elements. Without getting into the details[4], it can be concluded that this implements some kind of locking mechanism, that prevents overwriting the current configuration.

[3]More information on how this is working in detail cannot be found withing Xilinx guides and would therefore only be unprovable guessing.

[4]A further discussion of this analysis would be far beyond of this paper and is therefore omitted here.

IV. Context Save And Restore

The summary of Section II and III can be as follows:

- Xilinx FPGAs provide mechanisms to start a design with dedicated initial values in the storing elements via the *GRESTORE* command. These initial values are embedded in the configuration blocks of the bitstreams.
- The readback capture mechanism offers the possibility to get the current state of the configured storing elements back from the FPGA.

As a conclusion, a context save and restore of the entire FPGA or only a reconfigurable partition should be achieved by the following steps:

1) Load the initial bitstream for the design into the FPGA.
2) Let the design do its job for a while. This usually changes the internal state (context) of the design.
3) *Context Save:* Readback (capture) the current internal state of the configured design.
4) Use the FPGA (reconfigurable partition) for another design, otherwise context save and restore would not be required.
5) *Context Restore:* Load an updated bitstream (updated with the saved values of the storing elements) of the original design back into the FPGA.
6) Let the design progress doing its job.

This outlined way of performing a context save and restore is neither documented nor supported by Xilinx for their FPGAs. Furthermore, Xilinx recommends to not manipulate bitstreams on your own, as this might severely damage the FPGA.

The above context save and restore steps are straightforward from an academically niggling perspective. However, the devil is in the details, and the details differ from FPGA family to FPGA family. Therefore the following sections present a brief summary of the details for the 7-Series and Ultrascale-Series devices.

A. General Overview

Independent of the used FPGA family, the following points have to be taken into account, when trying to perform context save and restore:

1) The PR-bitstreams are neither compressed nor encrypted.
2) During configuration and readback capture, all clocks feed into the reconfigurable partition need to be disabled (usually via clock gating). Also respect the PR design-guidelines found in [9].
3) Take care of the padding frames and data words when writing (ending with padding data) and reading (starting with padding data) configuration data (see [6] chapter 6 or [8] chapter 10 for details on configuration data padding).
4) Be aware that the *GLUTMASK_B* bit in the *CTL0* register is set (compare *CTL0 setup* sequence in Table II).

Otherwise the values of distributed RAM will not be initialized or read back correctly.

5) Be aware to do a readback capture and not only a pure readback (see Section III-B). Otherwise you will not get the correct values of the designs storing elements.
6) Take care when using manipulated or self generated bitstreams as you might damage or destroy your FPGA.

B. 7 Series Details

According to Section III-C three kinds of *write config data* sequences can be found in an unmodified 7-Series PR-bitstream for each reconfigurable partition. These three kinds are the (exactly one) *CFG_CLB* configuration block, the (one or more) *blanking* blocks and the (equal in numbers and corresponding starting frame address) *configuration* blocks. The context save bitstream to extract the current state of the reconfigurable partition is performed with the readback capture bitstream shown in Table V.

sequence name	summary
sync sequence (see Table I)	readback capture setup
reset CRC sequence (see Table II)	
IDCODE check sequence (see Table II)	
write CFG_CLB block sequence	
reset CRC sequence (see Table II)	
SHUTDOWN command sequence	
reset CRC sequence (see Table II)	
GCAPTURE command sequence	
CTL0 setup sequence (see Table II)	
5x nop word	
for each configuration block (of the original Bitstream): read config data sequence (see Table IV)	capture phase
restore CTL0 sequence (see Table II)	readback capture cleanup
5x nop word	
START command sequence	
park FAR sequence (see Table II)	
reset CRC sequence (see Table II)	
dsync sequence (see Table I)	

TABLE V
7-SERIES READBACK CAPTURE

During the *capture phase* the configration data and additionally, the current values of all storing elements, are read back. The *CLB* type configuration blocks can be used directly to update the corresponding configuration blocks of the (original) PR-bitstream. However, *block RAM content* type configuration blocks need additional treatment before updating them back into the original PR-bitstream. This *block RAM content* type configuration frame treatment was first described in [5] for Virtex-6 devices. When comparing the original (*block RAM content* type) configuration frames with their readback counterparts, certain bits are modified (set to '1'), even when the block RAM contents have not been updated during design execution. According to [5]: *it can be assumed that these bits define for each BRAM, whether the memory contents should be restored.* So before updating the *block RAM content* type configuration frames back into the original PR-bitstream these bits have to be unset (to '0'). As the update equation of [5] is valid for Virtex-6 devices only, the 7-Series update scheme is given in this paper:

Withing each configuration frame, bit number 18 (words are indexed from 31 downto 0, so bit number 18 has index 17) of the following words need to be unset to '0' (count words starting with 0): 4, 14, 24, 34, 44, 55, 65, 75, 85, 95.

Finally, the updated bitstream now contains the current context of the partition. For a context restore, just configure the reconfigurable partition with the updated bitstream.

C. Ultrascale Series Details

Trying to apply the context save and restore mechanism, presented for the 7-Series devices, to Ultrascale-Series devices does not work. The reason, without getting too much into the details, is Xilinx changed the way how the initial values-*capture-GRESTORE* mechanisms work internally. As a consequence, readback captured *CLB* type configuration frames, which contain CLB Flip-Flop contents cannot be used to configure the FPGA again. Experience shows that this will not work as it breaks the functionality of the entire design of the reconfigurable module.

However, it is also necessary for Ultrascale devices to perform a readback capture, to get the current values of the storing elements. As the readback capture mechanism differs between 7-Series and Ultrascale Series, an Ultrascale Series readback capture bitstream is shown in Table VI For a detailed explanation of Ultrascale Series readback capture of the entire FPGA, see [7].

sequence name	summary
sync sequence (see Table I)	readback capture setup
SHUTDOWN command sequence	
reset CRC sequence (see Table II)	
NULL command sequence	
5x nop word	
CTL0 setup sequence (only set GLUTMASK_B bit)	
capture sequence (set CAPTURE bit in CTL1)	
8x nop word	
for each configuration block (of the original Bitstream): read config data sequence (see Table IV)	capture phase
unset CAPTURE bit in CTL1	readback capture cleanup
restore CTL0 sequence (see Table II)	
3x nop word	
START command sequence	
park FAR sequence (see Table II)	
reset CRC sequence (see Table II)	
dsync sequence (see Table I)	

TABLE VI
ULTRASCALE SERIES READBACK CAPTURE

The main differences to 7-Series readback capture are that there is no longer a *CFG_CLB* block and no *IDCODE check* required. Furthermore, there is no GCAPTURE command any longer, it is replaced by the *CAPTURE* bit residing in *CTL1* register.

As mentioned earlier, it is not possible to use a readback captured configuration frame, which contains the current values of storing elements to update the original bitstream. Fortunately, it is possible to get the position of each individual (storage element) bit of a design within a bitstream. This information can be found in *.ll (logic location) files, which can be generated as a secondary product by the *write_bitstream*

command when generating the *.bit-files. The following text snippet presents a single *.ll-file entry:

```
Bit 19602275 0x0042029f 195 Block=SLICE_X6Y3
  Latch=AQ Net=count_reg[12]
```

The line provides detailed information for the bit with index 12 of the design element *count_reg*. It is placed inside *Slice_X6Y3, Latch AQ*. It is the 195th bit of the configuration frame with FAR 0x0042029f. 19602275 is the bit offset, starting from FAR 0. The logic location file contains such an entry for each context relevant storing element (Flip-Flop, distributed RAM and block RAM) of a design.

By taking into account the information found in the logic location files, it is now possible to perform a context save:

1) Perform a readback capture of the desired partition (TableVI).
2) For each storing element (one line inside the logic location file):
   a) Extract the current value of the storing element out of the readback capture results.
   b) Update the corresponding bit in the original configuration bitstream.

Finally, an updated configuration bitstream is generated. This bitstream can now be used to perform a context restore by simply configuring it into the FPGA. This update strategy also applies to *block Ram content* type configuration frames. Therefore the special treatment of dedicated bits inside the readback captured *block Ram content* type configuration frames (see previous 7-Series related section) is unnecessary.

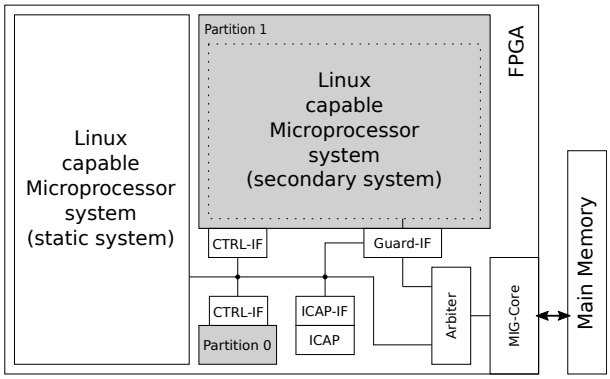
The above presented solution for the Ultrascale Series context save and restore can also be applied to 7-Series devices. Older publications like [11] (from 2005) imply that this solution is applicable to older FPGA families back to Virtex-II FPGAs.

V. Proof of Concept Tests

To prove the correctness of the context save and restore mechanisms, presented in the previous sections, a test system containing two partitions has been designed, as shown in Figure 2. The test system is composed of a microprocessor and several peripherals running Linux on top of it. It is based on our Partial Reconfigurable Heterogeneous System (PRHS) Framework presented in [3]. The previously presented context save and restore mechanisms, in addition with several bit/bin-file management functionalities were implemented in a Linux device driver that interfaces with the FPGA device-series specific ICAP-controller to perform configuration and readback of the partitions. The system and the respective reconfigurable modules are implemented and tested with Vivado 2018.3 on the Nexys4DDR (Artix 7), VC707 (Virtex-7) and VCU108 (Virtex Ultrascale) FPGA evaluation boards.

A. Basic Testing

The first partition is a tiny one (partition 0 in Figure 2), to just hold either a simple counting state machine (for CLB-FF testing), a distributed RAM or a block RAM design. This allows to easily test the applicability of the describes context save and restore mechanism for all of the three aforementioned fundamental storing elements of the FPGA.



The *CTRL-IF* allow to control the clock and user set/reset signals for the partition. It also allows to interact with the partition via specific communication registers or Fifos. The *ICAP-IF* provides control and interface between the microprocessor system and the ICAP. The *Guard-IF* enables the static system to limit and supervise the main memory access of any design residing in partition 1.

Fig. 2. Overview on test system.

It is possible, to configure one of the reconfigurable modules, operate it (let the counter count, change contents of the distributed or block RAM), save the context, exchange the partitions design by one of the others, operate them and change back (context restore) to the original design. These experiments show, that all of the the basic storage elements are fully supported by the context save and restore mechanisms.

B. Testing a Complex Design

The second partition (partition 1 in Figure 2) of the test design is large enough to hold another microprocessor based computing system running its own Linux instance as operating system. The test was performed according to the following sequence:

1) Configure the second computing system into partiton 1, and start Linux on top of the PR-designs microprocessor.
2) Suspend the execution of Linux (on the partial reconfigurable system). Perform a context save of the reconfigurable partition and (as it is part of the overall systems context) also save the content of the second systems main memory content (which resides external to the FPGA).
3) Configure another (in this test scenario blank) reconfigurable module into the partition (to overwrite the ”old” context). Clean the main memory portion of the secondary system.
4) Perform a context restore of the previously configured microprocessor system. Additionally, restore the corresponding main memory contents. Afterwards, resume the execution of the secondary system.

The above sequence was performed several times during execution of the secondary system (e.g. during Linux bootup or while executing a benchmark). Results show, that the presented context save and restore mechanisms also work for complex systems like an entire processing system, which contains large amounts of CLB Flip-Flops, distributed and block RAM. E.g. on the Artix7 implementation, the reconfigurable module of the secondary partition contains approximately 1MBit of relevant context, according to the corresponding logic location file. Most of this context is contained in the microprocessors Memory Management Unit and Caches.

VI. Conclusion

In this paper the technical aspects and details to perform an officially unsupported context save and restore of reconfigurable partition within modern Xilinx FPGAs (7-Series and Ultrascale-Series devices) are summarized. The context save and restore is based on the configuration readback and state capture mechanisms of the FPGA. Two different possibilities to perform such a context save and restore are presented, whose applicability depends on the FPGAs features and internal functionality. The possibilities have been tested for different FPGAs and are proven to be working as intended, when obeying several technical hints and design guidelines, also presented in this paper.

References

[1] Alban Bourge, Olivier Muller, and Frederic Rousseau. Generating efficient context-switch capable circuits through autonomous design flow. *ACM Transactions on Reconfigurable Technology and Systems*, 10:1–23, 12 2016.

[2] K. Dang Pham, E. Horta, and D. Koch. Bitman: A tool and api for fpga bitstream manipulations. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 894–897, March 2017.

[3] Marcel Eckert. *FPGA-Based System Virtual Machines*. PhD thesis, Helmut-Schmidt-Universitt, Holstenhofweg 85, 22043 Hamburg, 2014.

[4] Marcel Eckert, Dominik Meyer, Jan Haase, and Bernd Klauer. Operating system concepts for reconfigurable computing: Review and survey. *International Journal of Reconfigurable Computing*, 2016.

[5] Markus Happe, Andreas Traber, and Ariane Keller. Preemptive hardware multitasking in reconos. In Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, pages 79–90, Cham, 2015. Springer International Publishing.

[6] Xilinx Inc. *7 Series FPGAs Configuration User Guide (ug470)*.

[7] Xilinx Inc. *Configuration Readback Capture in UltraScale FPGAs (XAPP1230)*.

[8] Xilinx Inc. *Ultrascale Architecture Configuration User Guide (ug570)*.

[9] Xilinx Inc. *Vivado Design Suite User Guide - Partial Reconfiguration (ug909)*.

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

[11] H. Kalte and M. Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 223–228, Aug 2005.

[12] Dirk Koch, Christian Haubelt, and Jürgen Teich. Efficient hardware checkpointing: concepts, overhead analysis, and implementation. In *FPGA*, 2007.

[13] Markus Köster, Heiko Kalte, and Mario Porrmann. Relocation and defragmentation for heterogeneous reconfigurable systems. In *ERSA*, 2006.

[14] Wesley J. Landaker, Michael J. Wirthlin, and Brad L. Hutchings. Multitasking hardware on the slaac1-v reconfigurable computing system. In Manfred Glesner, Peter Zipf, and Michel Renovell, editors, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 806–815, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[15] Shaoshan Liu, Richard Neil Pittman, and Alessandro Forin. Minimizing partial reconfiguration overhead with fully streaming dma engines and intelligent icap controller (abstract only). page 292, 01 2010.

[16] Shaoshan Liu, Richard Neil Pittman, Alessandro Forin, and Jean-Luc Gaudiot. Minimizing the runtime partial reconfiguration overheads in reconfigurable systems. *The Journal of Supercomputing*, 61(3):894–911, Sep 2012.

[17] A. Morales-Villanueva, R. Kumar, and A. Gordon-Ross. Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable fpgas. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1505–1508, March 2016.

[18] H. Simmler, L. Levinson, and R. Männer. Multitasking on fpga coprocessors. In Reiner W. Hartenstein and Herbert Grünbacher, editors, *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, pages 121–130, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[19] H. G. Vu, S. Kajkamhaeng, S. Takamaeda-Yamazaki, and Y. Nakashima. Cprtree: A tree-based checkpointing architecture for heterogeneous fpga computing. In *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, pages 57–66, Nov 2016.

[20] Hoang-Gia Vu, Takashi Nakada, and Yasuhiko Nakashima. Efficient multitasking on fpga using hdl-based checkpointing. In Nikolaos Voros, Michael Huebner, Georgios Keramidas, Diana Goehringer, Christos Antonopoulos, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pages 590–602, Cham, 2018. Springer International Publishing.