Minimalistic SDHC-SPI hardware reader module for boot loader applications

Paulino Ruiz-de-Clavijo, Enrique Ostúa, Manuel-J. Bellido, Jorge Juan, Julián Viejo, David Guerrero

Departamento de Tecnología Electrónica, E.T.S. Ingeniería Informática, Universidad de Sevilla, Spain

Keywords:

Field-Programmable Gate Array
Secure Digital
Boot loader
Serial Peripheral Interface
System on Chip
Embedded systems

A B S T R A C T

This paper introduces a low-footprint full hardware boot loading solution for FPGA-based Programmable Systems on Chip. The proposed module allows loading the system code and data from a standard SD card without having to re-program the whole embedded system. The hardware boot loader is processor independent and removes the need of a software boot loader and the related memory resources. The hardware overhead introduced is manageable, even in low-range FPGA chips, and negligible in mid- and high-range devices. The implementation of the SD card reader module is explained in detail and an example of a multi-boot loader is offered as well. The multi-boot loader is implemented and tested with the Xilinx's Picoblaze microcontroller.

1. Introduction and related work

FPGAs offer a fast and cost-effective path for embedded system design in general and are playing an important role in the field of embedded systems due to its flexibility. Despite the flexibility of the FPGA platform, any embedded system has to provide a solution to the problem of system initialization. FPGAs reconfiguration implies two main parts: logic reconfiguration and embedded system initialization, since most of them require an initialization process. Different solutions can be applied depending on the complexity of the system. In the most simple systems, including those lacking a processor and those using simple soft-core microcontrollers, the whole design and the software code can be included into the programming bitstream. The main drawback of this approach is that the full system have to be re-configured even for minor software updates and the full size of the bitstream must be handled. More complex systems typically include a proper microprocessor (hard or soft-core) together with a dedicated off-chip RAM and ROM memory. These systems frequently use some kind of boot loader, that is to say, a set of software routines that are executed upon system power-up.

The logic reconfiguration process for volatiles FPGAs can be carried out in several ways. Taken Xilinx FPGAs as an example, it must be commented that they are SRAM-based devices with multiple reconfiguration methods, each one suitable according to an end application [1]. Digilent Inc. manufacturer builds Xilinx FPGAs-based boards including several reconfiguration methods and external peripherals. Nexys4 board from Digilent Inc. [2] is an example of an off-chip DDR RAM and an SD card reader, among others off-chip resources. The board also includes several boot loading schemes that allow the reconfiguration from internal ROM, from SD card and from USB mass storage, among others. For Nexys4, the bitstream uploading from SD card is carried out by an extra microcontroller added to the PCB (PIC24FJ128GB106 [3]). In custom designs, adding a microcontroller increases the complexity and causes inconvenience in final PCB. Moreover, the microcontroller used in Nexys4 has large firmware (128 KBytes of flash memory and 16 KBytes of RAM) due to the fact that it manages a FAT32 file system to look up the bitstream file.

The next step after the reconfiguration process is initialization. The system initialization code in embedded processors is typically stored in a ROM-like memory, which is read by the embedded processor at boot time. Initialization code involves, at least, some peripheral checking routines and a first-stage boot loader. The main processor in small systems implemented in reconfigurable hardware is sometimes a very simple soft-core microcontroller with very little resources and the only storage available is provided by block RAM devices. Thus, both the boot loader and the initialization code utilize valuable storage and processor resources that can usually be changed only by a full system re-programming. In this situation, an excessive boot loader footprint either impacts on the space available for the application firmware or it simply does not fit in the FPGAs internal RAM.

Systems with off-chip dedicated RAM and ROM may afford implementing a boot loader with multiple boot loader stages to overcome these limitations. A single-stage boot loader may be stored in a non-volatile memory or in the FPGAs bitstream and it may trigger the initialization process. However, every change requires a full re-programming process. In most of cases, these systems implement the boot loader with at least two stages [4]. The first stage may either comprise loading the complete system code from some storage device or just loading a second-stage boot loader with extended capabilities. This first stage (also called pre-loader) is a small piece of code that is not often updated, as the most firmware updates are usually carried out in the second stage. Multiple approaches and implementations propose different methods and/or methodologies [5–7] for the boot loader process depending on the application. The common principle is a boot loader with two important features: low-footprint area and the ability to upload firmware updates [6].

Furthermore, the advances in flash memory technology have made the integration of mass storage devices in electronic embedded systems possible. Flash memory is widely used due to the maturity of the technology, low cost and high capacities. An accepted standard for flash memory is Secure Digital (SD) card, being the most used removable medium for data storage applications in embedded systems. SD cards contain the main flash memory controller. Thus, they are integrated into a system by implementing the SD card protocol described in the SD Card Specification [8]. The SD card protocol is complex but there exists serial peripheral interface (SPI) as an alternative communication mode. The SPI mode is suggested in the SD Card Specification as an alternative for those systems, in cases where the full SD card protocol is too heavy.

System on Chip (SoC) technology is based on general purpose processors embedded on a chip, and a software-based SD card access technique is used in most applications whenever the mass storage is required. The SD card software for these systems is implemented in two different ways depending on the SD card required performance: either by developing the full SD card protocol or by using the SPI SD card mode. The full SD protocol uses a 4 bits parallel transference for data that increases the speed versus the single serial data bit used in SPI mode.

The related work in [9] uses the SPI for interfacing an SD card with a MSP430 microcontroller and presents a driver written in C language for SD cards. Since the SPI mode of the SD card protocol is a subset of the full SD protocol, the size of the driver is reduced, but the solution sacrifices the performance. Other approaches implement an SD card controller in hardware [10–12]. These alternatives do not use a general purpose microcontroller. Designs like [10] implement an SD card controller in Verilog hardware description language (HDL). Nonetheless, since the purpose of the implementation is to achieve fast read-write operations, the result is a complex set of Finite-State Machines (FSM). The related work [11] also introduces a full SD controller that aims low power consumption applications. Open designs available in OpenCores [12] provide solutions for SD cards interfacing that incorporate the full SD protocol. Generally, those designs are suitable to be used in complex microprocessor-based architectures and are supported by an additional software driver.

This paper proposes a hardware SD card reader for High-Capacity (HC) SD cards in SPI mode (SDHC-SPI). The SDHC-SPI reader module is developed for FPGA-based designs with a low hardware footprint. The module can be used to implement flexible boot loader functions in hardware that may completely replace software boot loader stored in a system ROM. After this introduction, Section 2 will describe the internal features of the SDHC-SPI reader module and Section 3 will analyze the implementation of a sample multi-boot loader system for Xilinx's Picoblaze microcontroller [13]. Section 4 will include the functional validation of the system and implementation results and finally, Section 5 will summarize the main conclusions.

2. SDHC-SPI reader module

SD cards are initialized following a serial protocol based on a sequence of command-response operations. The SPI mode can only be set just after the SD card is powered on and following a concrete sequence of commands, with some time restrictions. Once the SPI communication mode is enabled, SD cards only support a subset of the full SD protocol.

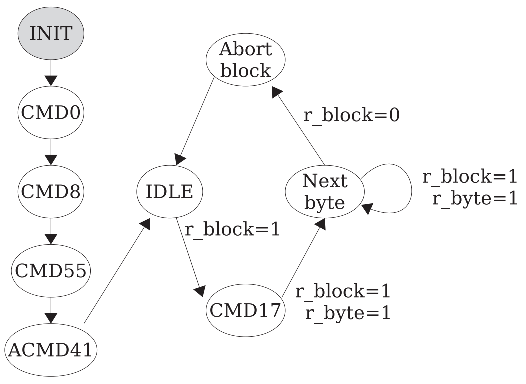


Fig. 1. Top-module finite-state machine diagram.

The initialization commands must be sent in correct order, the responses must be checked after each command and a waiting time limit must be considered for responses. The initialization process is fully described in [8] for all SD cards types: SD, SDHC and SDXC. Since only SDHC is supported in this particular implementation, the initialization process is greatly simplified.

The overall operation algorithm of the module is depicted in Fig. 1 considering time restrictions from [9,14]. In this figure, the FSM controller for SD card starts at INIT state. It puts the SD card into SPI mode after powering up and during the first reset command (CMD0). SPI mode cannot be changed while the SD card is powered on. Basically, this first state sets the SD card into SPI mode sending 80 clock cycles at 80 kHz with chip select (CS) signal asserted. The initial clock frequency should be less than 400 kHz therefore, it should give the card enough time to initialize its internal logic before sending the first command [14].

The FSM controller initialization process will succeed, if the IDLE state in Fig. 1 is reached. The commands sent are summarized as follows:

- CMD0: It is a command to reset the card and send it to idle state.
- CMD8: It is a command used to verify the conditions of the card operation.
- CMD55-ACMD41: It is a command pair used to start the internal card initialization and check when the card is ready to accept other commands.

The command pair CMD55-ACMD41 is sent in a loop until it overcomes successfully. Once this state is reached, the SD card holds in an idle state waiting for read or write commands. The FSM also keeps in an IDLE state (see Fig. 1) waiting for requests to read data. To attend the data read requests the proposed design only uses the CMD17 command, which reads data in the SD card in a single block mode.

It must also be noted that the SD protocol in SPI mode is protected by a CRC. Previous commands are 6 bytes long, being the last byte a 7-bit CRC with a final stop bit. The response of read command (CMD17) generates a 16-bit CRC code appended at the end of data retrieved. A possible implementation for CRC7 and CRC16 units uses 7 plus 16 flip flops connected with some logic, as it is presented in [14].

2.1. SDHC-SPI reader module overview

According to the FSM described in the previous section, the SDHC-SPI developed aims to minimize the hardware footprint. The design's key points are:

- The core only uses the subset of the SDHC commands depicted in Fig. 1. Thus, only SDHC cards are supported, which are the most common SD cards in use at present.
- CRC7 and CRC16 units are not included. The CRC16 field received

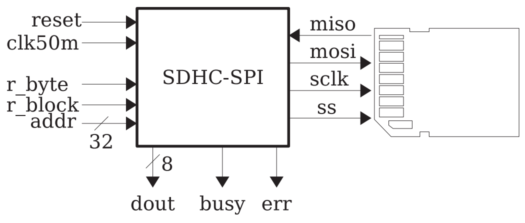


Fig. 2. Top view module.

after block read operations is ignored.

- Internally, the SD card commands (6-byte codes) are stored in a small ROM.
- No input buffer is used. The 512-byte block of the read operations are fetched under demand and directly from the SD card byte by byte.
- Only the single-block read mode of SD card protocol is used (CMD17).
- At top level, a simple handshake protocol is utilized to simplify integration into higher-level designs.

The top view of the module has the inputs and outputs shown in Fig. 2. From a high-level view the module performs two tasks: SD card detection and initialization, and data block reading. The SD card initialization process starts with signal *reset* and SD data block read is triggered by the signal *r_block*. The *busy* signal keeps asserted (set to logical one) in both operations while the operation is performed. The operation ends when the *busy* signal is de-asserted (set to zero). Once the *busy* signal is de-asserted, the operation result can be tested during the next clock cycle by checking the *err* signal. The *err* signal is only asserted when the operation fails.

Fig. 3 represents a time diagram of the SD card initialization process, which is automatically triggered when the *reset* signal is de-asserted and ends when the *busy* signal is de-asserted by the module. The signal *err* acts as a result indicator: if the initialization process fails or the SD card is not detected, the signal *err* is asserted, otherwise the card is detected and initialized successfully.

The second available operation is a single-block read from the SD card. Single block read operations in SDHC cards utilize CMD17 and return a fixed size of 512 bytes plus 2-byte CRC. The SDHC read command assumes that the card is divided into 512-byte blocks and each block is addressed by a 32-bit integer number. The address is sent as an argument into the read command CMD17. The SDHC-SPI interface for reading is driven by *r_block* and *r_byte* signals and must be asserted in the correct order. Fig. 4 shows the time diagram of the reading process. The process starts when *r block* is asserted and the block address is captured from the *addr* input signals. Then, the module sends CMD17 to SD card and polls it waiting for the response. Once the SD card has the block ready to sent, the *busy* signal is de-asserted and data can be retrieved asserting *r_byte* signal multiple times. The overall behavior is like 512-byte stream received sequentially from *dout* bus. In detail, for

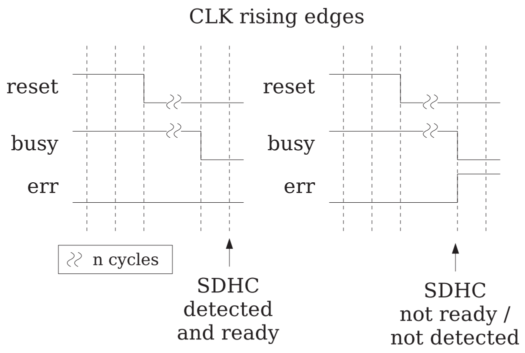


Fig. 3. SD card initialization / detection time diagram.

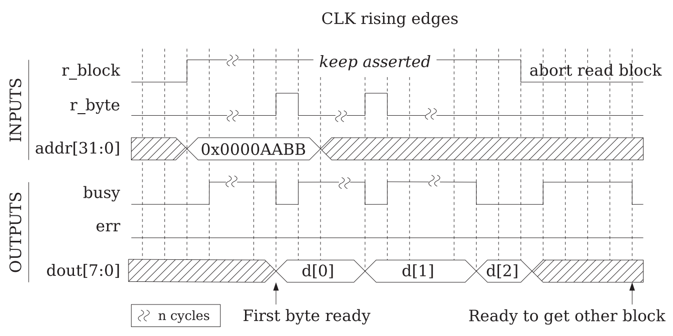


Fig. 4. Read block time diagram.

each pulse on *r_byte*, the module sends one byte of the 512-byte block, as it is depicted in Fig. 4 with $d[0]$, $d[1]$ and $d[2]$, for instance. Fig. 4 also shows the possibility of partial block read, from byte 0 to byte 2 ($d[0]$, $d[1]$ and $d[2]$), de-asserting the *r_block* signal when the full block is not required.

Read operations with unsuccessful result can happen when the module reaches a timeout waiting for the SD card response. In this situation, the signal *err* is asserted and no other read operations can be executed until the SD card is re-initialized.

2.2. SDHC-SPI reader module implementation

The algorithms to control the SD card are completely developed in hardware using the FSM technique and coded in VHDL. Fig. 5 shows the internal block diagram of the reader module where the main blocks are the Main FSM Unit and the SDCMD Unit. Main FSM Unit implements the algorithm of the Fig. 1 controlling the SDCMD Unit. Main FSM Unit requests to send SD commands to SDCMD through signals *w_cmd*, *w_addr* and *w_byte* and waits for the command execution ends. The signals *addr* and *din* are the argument for the SD commands and the SD command result is given back through signal *dout* for its analysis.

The SDCMD Unit sends 6-byte command sequences to the SD card through the SPI. The SPI is a custom design unit since in the communication with SD cards some important details must be considered to successfully complete the initialization process for most SD cards. Time restrictions for the SD cards initialization process are studied in detail in [14] as well as in the driver presented in [9] for SD-SPI protocol highlights, showing some important time rules to communicate with most SD cards. However, both are software-oriented solutions and in hardware implementation there are some issues to be considered in SPI custom implementation. In software implementation of SPI like [9], the *SS* (slave select) line is asserted and de-asserted in spaced instructions. Thus, the pulse in SS is wide, specially when the microcontroller (shown in that work) runs at 8 MHz. In a hardware controlled SPI, the *SS* signal can be asserted and de-asserted in two successive clocks cycles, and some SD cards do not detect such a small pulse. In consequence, our SPI hardware implementation includes some specific functionality as a glitch filter on the *SS* signal.

The SPI unit is also able to change the clock frequency of the SPI protocol because SD cards initialization process requires several frequencies (80 kHz, 400 kHz). In the reader module, the read operations

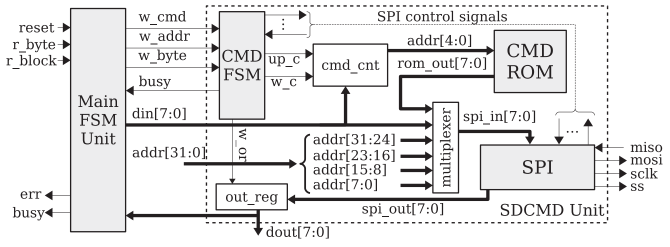


Fig. 5. SDHC-SPI Internal structure.

have fixed the SPI clock frequency to a constant value set by the designer in the source code. The choices available for SPI clock are $clk50m/2$, $clk50m/4$, $clk50m/64$ and $clk50m/512$, where the *clk50m* input signal (Fig. 2) is a 50 MHz input clock.

Another significant unit is CMDROM, as it stores the 6-byte sequences of the SD CMDs. Since the SDHC-SPI only uses four SD card commands, the ROM size is only 24 bytes long. The CMD's last byte is the CRC7 also stored in ROM, but it is only required for both CMD0 and CMD8. It is not necessary for the rest of commands when the SD card is in SPI mode. An FSM-controlled counter is used to address the command in the ROM. After each command is sent, the FSM waits for the response and the module keeps on working while the response matches the expected one.

3. Picoblaze boot loader

The present contribution proposes the use of SDHC-SPI to develop a hardware boot loader that can retrieve the program code from an SD card, without requiring boot loading software run by the main processor or microcontroller. It is useful both to replace fixed storage like ROM and also to allow quick and easy changes in the embedded code, that can be replaced with the appropriate tools by means of any desktop computer. The solution is specially beneficial for soft-core microcontrollers with little ROM and RAM storage available. The program code is typically small in these systems, therefore the low speed of the SPI interface is not likely to be a problem. A specific boot loader adapter is required for each microprocessor, but the adapter's complexity and footprint is low either compared to the SDHC-SPI or to the whole system. This section presents a sample hardware boot loader for Xilinx's Picoblaze [13] microcontroller.

Picoblaze is an 8-bit microcontroller optimized for Xilinx's FPGA devices and distributed as an IP core. Picoblaze has a read-only memory of 1024-instruction words for program code of 18-bit wide per instruction. The instructions are compiled within the FPGA design and automatically loaded during the FPGA configuration process. A software boot loader for Picoblaze is not feasible because the code memory is not writable from its instruction set.

Filling the RAM is fast with this memory capacity, despite SPI protocol performance. The program ROM is synthesized as a fixed block RAM in Xilinx devices. The boot loader only needs to use the *write* signal of the block RAM to fill it with the program code. We propose the example of system architecture depicted in Fig. 6(a) to upload the code in a different way from the stock solution. The *Boot Loader Adapter Unit* mainly consists of an FSM that controls the Picoblaze *p_reset* signal while the program is being transferred from the SD card to the block RAM. The SDHC-SPI retrieves a byte stream from the SD card, but the memory bus size of Picoblaze is 18-bit wide as previously mentioned. A memory bus adapter is required to make the 8-bit stream compatible with 18-bit memory words. The memory adapter packs each 18-bit instruction into 3 bytes and the two most significant bits (MSB) of each byte are not used. The packed instructions are directly stored and retrieved from the SD card. Table 1 displays an example of a test program assembled to binary code and converted to be stored into a byte stream using the aforementioned method.

Table 1
Binary code packed from 18 bits to 3 bytes.

Instruction	HEX code	18-bit binary code	SD card data
load s0,AA	0×000AA	000000 000010 101010	0×00 0×02 0×2A
output s0,04	0×2C004	101100 000000 000100	0×2C 0×00 0×04
jump 0	0×34000	110100 000000 000000	0×34 0×00 0×00

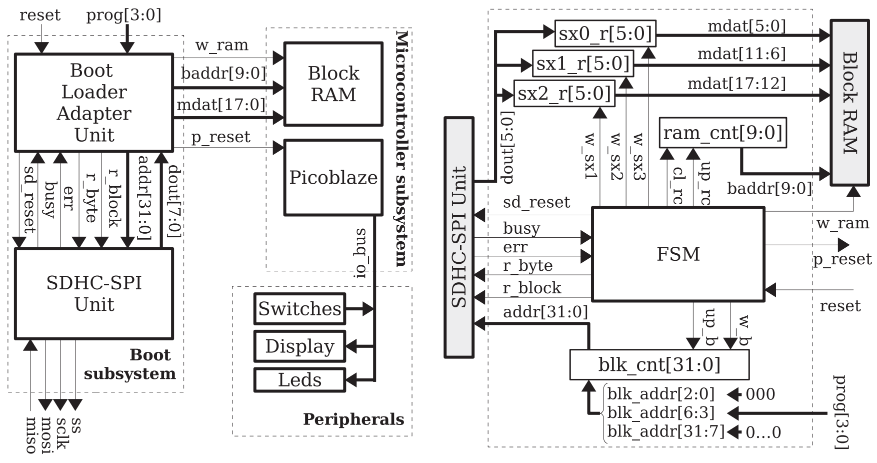
A simple software tool written in python is used to convert the HEX codes returned by the Picoblaze assembler into the packed format and write them to the SD card. The boot loader adapter is depicted in Fig. 6(b) where the registers *sx0_r*, *sx1_r* and *sx2_r* are used to convert between the data formats. An FSM controls the data transfer from SDHC-SPI to the registers and from the registers to the Block RAM.

The sample boot loader also includes a multi-boot operation. Several programs can be stored in different blocks of the SD card and the final loaded program can be selected from a set of manually-operated switches connected to signal *prog*. The software tool supports the multi-boot operation and can processes several assembler programs generating a single file that can be written into an SD card directly. The boot loader adapter load the *blk_cnt* counter (Fig. 6(b)) with the SD card address corresponding to the selected program in the signal *prog*.

4. Results

A complete system including these modules and the Picoblaze microcontroller is designed and implemented in order to test the proposed SD card reader and boot loader modules. The system is primarily tested on the Basys2 development board from Digilent [15], which includes Xilinx Spartan-3E XC3S100E [16] FPGA chip. This is a low-grade chip intended to prove the feasibility of the proposed design even in a limited resources scenario. The system is also tested on a Xilinx Virtex-5 XC5VLX50T [17] high-grade FPGA chip and in a modern low-grade Xilinx Artix-7 XC7A35T chip [18] in order to show the small footprint of the hardware implementation in state-of-the-art devices.

The subsequent section will summarize the functional and implementation results of the reader and boot loader modules together with the complete sample system.



(a) Top view (b) Boot loader adapter

Fig. 6. Picoblaze boot loader system.

4.1. SDHC-SPI reader module verification

Both simulation and execution on the prototype implementations are used to verify the SDHC-SPI. For this purpose, a set of test benches are designed. The simulation test benches read the contents of the memory from a simulated SD card module coded in VHDL language. A software tool is specifically developed to automatically generate this VHDL code from a given memory content in raw format.

The implementation platform is the already mentioned Digilent's Basys2 board. Various peripherals on the board are used to control and check the correct operation of the SD card reader: push buttons, switches, LEDs and a 7-segment display. Controllers for the peripherals are also developed, as needed. In all the tests, the SD card is preloaded with some raw content. Such tests are executed and then checked using the board's peripherals. Below, we list the three tests carried out:

- Test 1: The SD card is initialized and the first byte is read from the SD card and displayed in hexadecimal. Additional bytes can be read and displayed by pressing a control button.
- Test 2: A whole SD block (512-byte length) is read from the SD card calculating the *xor* checksum byte by byte. The final result is shown in hexadecimal on the display.
- Test 3: A sequential number of SD blocks is read and an *xor* checksum of all bytes is calculated and displayed. The final value can be verified to make sure it is the right one.

However, some details must be considered to run the tests. The SD card is filled in all tests in raw mode with pre-established size random data. The SD card block to read is manually selected in test 1 by the eight input switches available in the prototype board. The results displayed are verified in test 2 by matching them with the output of a software tool. A button on the board is used to run the test on the next block, while the current block number is also being displayed. Then, the number of sequential blocks must be set in test 3 in VHDL code before the test is synthesized. As in test 2, a software is used to calculate the *xor* checksum of the complete sequence of blocks as well.

4.2. Picoblaze boot loader verification

The multi-boot loader for Picoblaze system has been deployed into the Basys2 prototype board. The boot loader verification basically consisted of booting the Picoblaze microcontroller with three different test programs stored on an SD card and selected by the switches available on the board. First, the SD card is filled with the binary data generated by the software tool mentioned in Section 3, containing the binary code of each program concatenated and properly aligned. The programs for the multi-boot system are prepared in two steps. In the first one, the programs are assembled with the Picoblaze assembler software, available with the Picoblaze tool kit. That software outputs a file with the memory contents in hexadecimal format, which are translated into the 3-byte format shown in Table 1. In the second step, each program is aligned to 4096-byte blocks in order to optimize the SD card operation. Both steps are carried out automatically by a python script that generates an SD card image that, in turn, can be directly dumped on the device. The whole system has been tested by selecting and running the three test programs stored on the SD card. The process has worked correctly in all the tests with several repetitions and using various SD card devices.

As noted in Section 3 the full packed code memory of Picoblaze occupies 1024×3 bytes, therefore the boot loader needs to read 6 blocks of the SD card (512 bytes per block). The time required by the Picoblaze boot loader system to load the program into memory (load time) has been measured using a logic analyzer for the different SPI clock frequencies available in the system (see Section 2.2) and they are shown in Table 2. In an FPGA implementation, this load time adds to the configuration time used by the platform to load the bitstream from ROM in order have a fully functioning system (power-on time). In our case, the Basys2 board takes 610 ms to be configured from ROM thus, the overhead in the power-on time introduced by the boot loader system, ranges from 1% to 50% of the total power-on time depending on the SDHC-SPI frequency configuration (Table 2). Since any modern SDHC device should be able to work at the maximum considered frequency of 25 MHz, it can be said that the power-on time overhead introduced by the boot loader system is negligible in most cases and of the same order of magnitude as the configuration time in the worst case.

Table 2
Program load elapsed time.

SPI clock frec.	Load time	Power-on time overhead (%)
25 MHz	6.84 ms	1.1%
12 MHz	10.52 ms	1.7%
780 kHz	81.70 ms	11.8%
97 kHz	614 ms	50.1%

4.3. Implementation results

Resource utilization is discussed in this section. Two designs are taken as reference, the *SPI-Master SD/MMC Controller* [19] and the *logiSDHC Secure Digital Host Controller* [20]. The former is an open hardware design and the latter is a licensed IP core. Both are SD host controllers that include more complete functionality like writing support and data buffers for better performance, but these capabilities do not suppose any benefit for boot loader applications. Tables 3 and 4 summarize the resources utilization of the full boot loader system and its main parts for a XC3S100E and a XC5VLX50T chip. These are examples of low-grade and high-grade chips respectively. The glue logic at top level is not included in tables, hence the difference between the sum of the parts and the full system. The SPI-Master has also been synthesized for these FPGAs and its results are also included in the tables as reference. The synthesis process has been executed with the following options: *Optimization Global - Area*, *Optimization Effort - High* and *Place & Route Effort Level - High*.

Considering the resources utilization for the low-grade chip in Table 3, the complete boot loader system (SDHC-SPI plus boot loader adapter) takes 24% of the resources. The SDHC-SPI takes most of the resources, since the boot loader adapter is more simple in comparison, but it is only one third of the SPI-Master occupation. Finally the sample Picoblaze system takes 15% of the resources leaving a 60% of the chip resources available for further extensions. In comparison, the SPI-Master alone takes about the 60% of the total chip resources leaving very little room to build a boot loader solution and specific application on top of it.

Regarding the high-grade chip in Table 4, the resource utilization in LUTs and slices is reduced by one third compared to the low-grade chip, which is coherent with the fact that the XC5VLX50T chip contains 6-input look-up tables [17] while the XC3S100E chip contains 4-input look-up tables [16]. Combined with the additional logic available, the full system takes less than 4% of the resources in the high-grade chip, whereas the boot loader solution (SDHC-SPI plus boot loader adapter)

Table 3
Resources utilization in Xilinx Spartan3E-100 - XC3S100E.

Core	Slices	Slices Reg.	LUTs	Slices (%)
SPI-Master [19]	590	583	822	61.4%
SDHC-SPI	179	111	270	18.6%
Boot loader adapter	55	82	78	5.7%
Picoblaze + Peripherals	144	127	231	15.0%
Full system	383	320	588	39.8%

Table 4
Resources utilization in Xilinx Virtex 5 - XC5VLX50T.

Core	Slices	Slices Reg.	LUTs	Slices (%)
SPI-Master [19]	364	571	685	5.05%
SDHC-SPI	116	111	211	1.61%
Boot loader adapter	37	82	75	0.51%
Picoblaze + Peripherals	84	127	182	1.17%
Full system	245	320	478	3.40%

Table 5
Resources utilization in Xilinx Artix-7 - XC7A35T.

Core	Slices	Slices Reg.	LUTs	Slices (%)
SPI-Master [19]	178	474	525	2,18%
logiSDHC [20]	636	1143	1578	7.80%
SDHC-SPI	63	88	211	0.77%

only takes about 2% of the resources. Again, the SDHC-SPI controller takes about one third of the SPI-Master resources.

The logiSDHC source code is not available, but its data sheet inform about resources utilization in a modern Artix-7 low-grade FPGA chip, the XC7A35T [18]. Both the SDHC-SPI and the SPI-Master have been synthesized for this chip using the Xilinx's Vivado framework with the option *AreaOptimized high*. Results are shown in Table 5. It must be noted that the SPI-Master and the logiSDHC include two block RAMs whereas SDHC-SPI only uses one ROM (CMDROM unit) which is already included in the LUTs count. In general, small ROMS like CMDROM (24 bytes long) are synthesized by Xilinx tools on LUTs in order to maximize performance and save block RAM resources.

Again SDHC-SPI resource requirements (and the whole boot loader solution therefore) are about one third of the SPI-Master and only a 10% of the logiSDHC module. Even though a modern FPGA chip can easily cope with a moderately complex SD card host, it can be seen that the proposed boot loader solution will have a negligible impact in resource utilization in any modern application.

5. Conclusions

A hardware SD card reader module for configurable FPGA devices has been presented. It has stood for implementing a complete multi-boot loader solution for the Picoblaze microcontroller in hardware using VHDL code, completely discarding the need for software boot code. The proposed solution is specially useful in low-memory embedded systems and in higher-capacity systems that can benefit from the flexibility of storing the boot code or firmware in removable, low cost, massive storage like SD devices.

Both the SD card reader and boot loader adapter have been implemented and then integrated into a sample Picoblaze-based system that has suitably run in all the tests. The boot loader solution presents a low-hardware footprint that makes it adequate even for low-grade FPGA chips ($\approx 24\%$) and a nearly negligible impact ($\approx 1-2\%$) in high-grade and modern low-grade devices.

References

[1] M. Li, M. Xie, G. Liu, X. Liu, A SPI FLASH-based FPGA dynamic reconfiguration method, in: 2013 IEEE International Conference on Microwave Technology Computational Electromagnetics, 2013, pp. 379–382.
[2] Digilent, Inc., Nexys4 DDR FPGA Board Reference Manual (2016).
[3] Microchip Technology Inc., DS39897C PIC24FJ256GB110 Family Data Sheet (2009).
[4] C. Gu, Building Embedded Systems: Programmable Hardware, Apress, Berkeley CA, 2016, Ch. Power On and Bootloader, pp. 5–25.
[5] D. Hartono, M.S. Ng, Z.N. Lim, S.W. Lee, V.V. Yap, C.M. Tang, A scalable bootloader and debugger design for an NoC-based multi-processor SoC, in: 2015 Proceedings of the 3rd International Conference on New Media (CONMEDIA), 2015, pp. 1–5. http://dx.doi.org/10.1109/CONMEDIA.2015.7449150.
[6] I. Pratt, S. Zhong, Bootloader design considerations for resource-constrained microcontrollers in RFID reader designs, in: 2014 IEEE RFID Technology and Applications Conference (RFID-TA), 2014, pp. 50–55. http://dx.doi.org/10.1109/RFID-TA.2014.6934199.
[7] A. Marchiori, Q. Han, A two-stage bootloader to support multi-application deployment and switching in wireless sensor networks, in: 2009 International Conference on Computational Science and Engineering, Vol. 2, 2009, pp. 71–78. http://dx.doi.org/10.1109/CSE.2009.50.
[8] SD Specifications Part 1 Physical Layer Simplified Specification Version 5.0, Tech. rep., Technical Committee SD Card Association (2016).
[9] F. Foust, Secure Digital Card Interface for the MSP430, Michigan State University, 2004.
[10] O. Elkeelany, V. Todakar, Data archival to SD card via hardware description language, IEEE Embed. Syst. Lett. 3 (4) (2011) 105–108, http://dx.doi.org/10.1109/LES.2011.2168804.
[11] P. Zhou, T. Wang, X. Wang, Y. Wang, Hardware Implementation of a Low Power SD Card Controller, in: Proceedings of the IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC 2014).
[12] OpenCores. URL ⟨http://www.opencores.org⟩.
[13] Xilinx Inc., PicoBlaze 8-bit Embedded Microcontroller User Guide for Extended Spartan-3 and Virtex-5 FPGAs. Introducing PicoBlaze for Spartan-6,Virtex-6, and 7 Series FPGAs (2011).
[14] J.A. Aseem Vasudev, Interfacing SD Cards with Blackfin Processors, Tech. rep., Analog Devices (2010).
[15] Digilent, Inc., Basys2 FPGA Board Reference Manual (2016).
[16] Xilinx Inc., Spartan-3 FPGA Family Data Sheet, DS099 Product Specification (2013).
[17] Xilinx Inc., Virtex-5 Family Overview, DS100 v5.1 Product Specification (2015).
[18] Xilinx Inc., 7 Series FPGAs Data Sheet: Overview, DS180 v2.4 Product Specification (2017).
[19] S. Fielding, spiMaster IP Core Specif. (2008) (URL ⟨http://opencores.org/project,spimaster ⟩).
[20] Xylon d.o.o., logiSDHC Secure Digital Host Controller, Data Sheet (2014).