An Open-Source Soft-Microcontroller Implementation Using an ARM Cortex-M0 on FPGA

Vitor Finotti[1], **Bruno Albertini**[1]

[1]Escola Politécnica – Universidade de São Paulo (USP)
Avenida Prof. Luciano Gualberto, 380 – 05508-010 – São Paulo – SP – Brazil

{vfinotti,balbertini}@usp.br

Abstract. *There is a myriad of projects that could be deployed on FPGA for architectural exploration. However, open-source platforms are scarce, and one with embedded software and operating system support to the application-specific hardware could not be found in the literature. We present an open-source soft-microcontroller architecture based on an ARM Cortex-M0, adaptable to different amounts of cores or new components, supporting an end-to-end deployment from code compilation using arm-gcc to loading the binary into the HDL memory cores. The proposed design is validated through simulation and implementation on a KC705 development kit, demonstrating busy-wait polling, DMA transfer, and deterministic real-time processing through FreeRTOS.*

1. Introduction

Field-programmable gate arrays (FPGAs) reconfigurable nature, associated with IO high-speed modules and specialized hardware present in modern FPGAs contributed to their popularization as a solution for applications targeting low-latency, high-bandwidth, and energy efficiency, from state-of-the-art projects [Contardo et al. 2015] to cloud datacenters [Caulfield et al. 2016]. However, many routine procedures and algorithms that are simple to implement using compiled machine instructions for general purpose processing architectures such as central processing units (CPUs) would require overly complicated state machines to be implemented in FPGAs using hardware description language (HDL). This problem can be overcome by two different approaches: soft-core microcontrollers, where a microcontroller is instantiated in FPGA fabric using reconfigurable blocks, and System-on-Chip (SoC) platforms, where a hard-core microcontroller is included in the same die as the FPGA. In both cases, a processor interprets machine instructions loaded from memory to perform arithmetical and data transfer operations.

Because SoC processors circuits are optimized for running machine instructions, they can achieve higher clock rates than soft-cores alternatives, resulting in higher speed [Jayakrishnan and Parikh 2019]. However, choosing a hard-core architecture implies in reducing the flexibility and reconfigurability of the design, since the processor is immutably printed at the silicon through photolithography. Besides, SoC solutions are vendor- and platform-dependent, restricting in reusing parts of the design in other projects.

Meanwhile, the choice of soft-cores is often limited to vendor-provided solutions, solving only partially the issues of flexibility and code reuse.

With that concerns in mind, we propose an open-source generic architecture built around an ARM Cortex-M0 processor, providing a vendor-agnostic HDL design that

can be used in several FPGA models. We based our project in the work developed by [Martos and Baglivo 2011], where a Cortex-M0 processor runs a simple script that periodically blinks an LED. Although innovative, this work was limited to the interaction between the processor, the memory, and a pattern detector. We extend that project's concept by introducing several elements essential to modern embedded systems design, such as interconnect, direct memory access (DMA) controller, and real-time operating system (RTOS) compatibility. We also replaced proprietary software whenever possible, preferring open-source cores and tools.

2. Related work

The two approaches for running compiled machine instructions in FPGAs are soft-core microcontrollers and SoCs. The latter is comprised of architectures such as Zynq [Xilinx 2021] or Intel SoCs [Intel 2021], which offer higher performance at the cost of less flexibility and higher cost since the microcontroller is printed in the silicon die as a fixed component. In addition to that, SoCs generally have a vendor-dependent ecosystem for HDL design and programming.

When implementation flexibility and final costs are project constraints, soft-core microcontrollers may be an interesting alternative to SoCs. While vendor-provided soft-microcontrollers such as Microblaze [Xilinx 2020] or Nios II [Intel 2020] are occasionally used in industry and academy, they share some of the SoCs' restrictions in terms of development flexibility and vendor dependence. A few vendor-agnostic projects such as aeMB [OpenCores 2021a], NEORV32 [Nolting 2020], or ZipCPU [Gisselquist Technology 2021] could be used to overcome those restrictions. However, they use non-standard instruction sets which may require some effort to be learned and used. ARM-based soft-microcontrollers are either completely proprietary [ARM 2020], based on older instruction sets [OpenCores 2021b], or a proof of concept with implementation restrictions [Martos and Baglivo 2011]. Our work presents an infrastructure around a Cortex-M0 processor, providing good levels of HDL customization while using a modern instruction set of the popular ARM architecture.

3. Main HDL cores

In this section, we detail the main HDL cores that build up our ARM soft-microcontroller. We made an effort for keeping the project as much open-sourced as possible, using cores made available by the HDL community and creating new ones when needed. The main HDL language used was SystemVerilog, with occasional use of VHDL where advised by the vendors due to implementation efficiency. The architecture of our platform is illustrated in Figure 1.

3.1. ARM Cortex-M0 obfuscated core

As the soft-microcontroller central microcontroller unit (MCU), we chose the Cortex-M0 processor core made available by the ARM DesignStart program [ARM 2020]. Despite other cores being offered at the program, the Cortex-M0 was chosen for being the smallest processor available (decreasing the required FPGA size for its implementation and increasing the number of platforms capable of supporting it). Besides, it is made available as an obfuscated core implemented using pure Verilog code. The other models, in

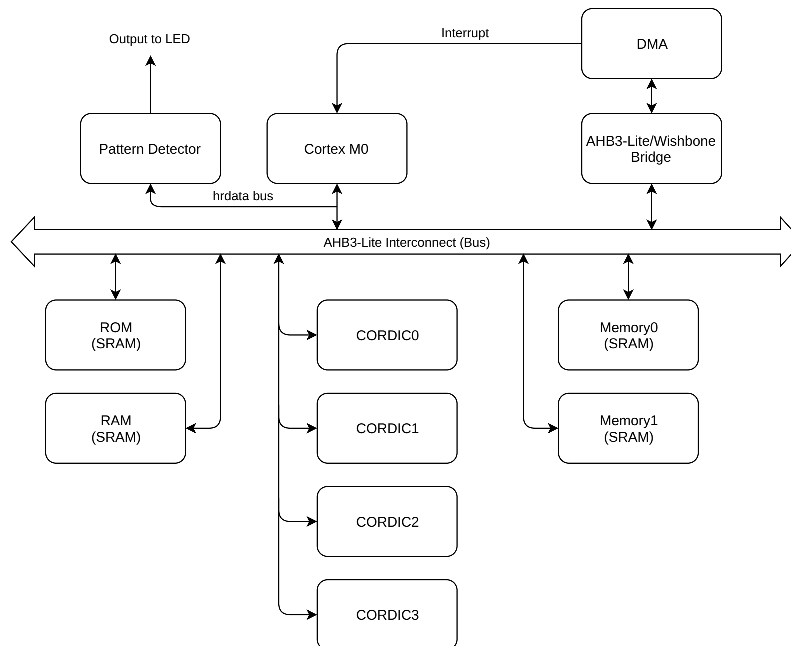

Figure 1. Schematic of our soft-microcontroller HDL design.

addition to occupying a larger space in FPGA due to their increased complexity, are implemented using the Xilinx block design (.bd) format, making it incompatible with platforms from other vendors and reducing portability and version control.

The Cortex-M0, as every other Cortex-M core, includes the CPU itself, which is responsible for executing machine instructions, an interrupt controller (NVIC), and a debugger interface. It communicates with the peripherals and memory through the interconnect, which behaves like a bus matrix that connects master devices, such as the MCU, to slave devices, such as the memories. In some cases, it allows simultaneous data transfers to occur between master and slaves. The communication approach adopted in this design is called `memory-mapped I/O` in which, from the perspective of a master device such as the MCU, all peripherals and memories are addressed by pointers so, when writing and reading from them, the master is writing to and reading from memory addresses.

Unlike the Cortex-M3 and Cortex-M4 models, the Cortex-M0 works almost exclusively with a 16-bit instruction set called ARM Thumb instruction set, which is widely used in the embedded variants of the ARM 32-bit processor. In the original ARM processors, all instructions were 32-bits (4-bytes), and, subsequently, the Thumb instruction set was introduced as a 16-bit variant aiming to improve memory utilization. At first, processors supporting Thumb instructions could switch (at the procedure level) between Thumb and 32-bit instruction sets. More recently, ARM has introduced a family of “M” processors that execute only Thumb instructions (plus a limited set of 32-bit instructions). These “M” processors include the Cortex-M4, Cortex-M3, Cortex-M1, and Cortex-M0. Of these, the Cortex-M0 has the smallest set of instructions. It is notable, that every ARM processor is capable of executing programs written with the Thumb-M0 instruction subset. For embedded applications, processors based on the ARM “M” core provide both high-performance and low-cost [Brown and Himebaugh 2016].

Since not all ports were required for performing most applications intended for this project, a wrapper was created to only externalize the relevant signals, setting the remaining ports to either ground or V_{CC} depending on the default expected value.

3.2. AHB3-lite interconnect

The MCU communicates with memory and peripherals through a *bus matrix*, which enables data to be routed across a fixed set of connections. From the MCU or, in general, any master device on the bus, there is no difference between peripherals and memories, since both are addressed by pointers. These devices are referred to as *memory-mapped* since all their interfaces are part of the memory address space of the master device. This model is illustrated at Figure 2, where three main components - a generic processor (CPU), a memory, and a generic I/O device - communicate using three different sets of signals: N-bits of address, M-bits of data, and some number of control lines. Generally, the CPU may initiate two different types of operations – read transfers and write transfers – with either the Memory or the I/O device as specified by the address.

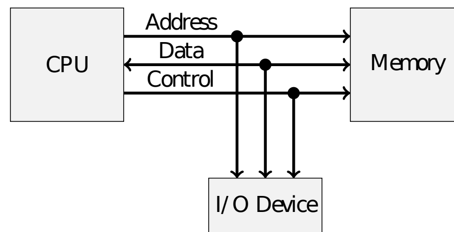

Figure 2. Memory mapped I/O model. Obtained from [ARM 2009].

In the case of the ARM Cortex-M0, address, data, and control signals are specified by the AHB3-lite protocol, which defines not only the signals but how they must behave to perform read and write transfers. To implement the communication between master and slave devices with AHB3-Lite protocol, we chose the Roa Logic AHB-Lite Multi-layer Interconnect core [RoaLogic 2020b]. This core is a fully parameterized interconnect fabric soft IP for AHB3-lite communications that target high performance and low latency. It allows a virtually unlimited number of AHB3-lite bus master and slave devices to be connected without the need for bus arbitration to be implemented by the bus masters. Instead, slave-side arbitration is implemented for each slave port within the core, providing support for priority and Round-Robin based arbitration when multiple bus masters request access to the same slave port. According to the vendor, arbitration is typically completed within 1 clock cycle.

Through the interconnect, we define the base addresses and addresses masks, so the interconnect can relate the address during a transfer initiated by a master core.

3.3. AHB3-lite general purpose SRAM

To infer the memory units, we used the AHB3-lite Memory core [RoaLogic 2020a] as a base for our implementation. This core infers an static random-access memory (SRAM) that supports read and write operations.

Despite technically being volatile memory, since it is an HDL coded core that is loaded by a bitstream on the FPGA platform, we can also infer the initial configuration of the SRAM, so that instructions for the MCU can be loaded into it. Since this

feature was not available in the original AHB3-lite Memory implementation, modifications were made to allow the loading of a text file with the initial values in each memory address during synthesis. The loading of this file and its format were implemented following the FPGA manufacturer guidelines, and the modified core is available in the soft-microcontroller project repository [Finotti 2020a].

3.4. Pattern detector

The pattern detector is a simple core that toggles its output bit when a specific pattern appears at the input bus. We conveniently connect the input bus to the *hrdata* master data bus, to provide an easy way of detecting when a pattern is read by the master. The idea here is to define a routine in the program executed by the MCU in such a way that the specific pattern to be detected is periodically read by the master, so the output bit can drive an LED or another external component to indicate the program is running.

3.5. AHB3-lite CORDIC

CORDIC (for COordinate Rotation DIgital Computer) is a simple and efficient algorithm used to implement several trigonometric functions, hyperbolic functions, square roots, multiplications, divisions, exponentials, and logarithms. Its simplicity and efficiency favor its use when computing efficiency is required or when hardware multipliers are a scarce or inexistent resource. We used the sequential CORDIC core created by [Thibedeau 2020] as a base for our implementation. For compatibility with the AHB3-lite bus, we implemented a wrapper around the original core, creating a process to translate the commands and signals used to operate the core into the AHB3-lite protocol.

In our design, we mainly use the CORDIC core as a dummy processing using with the only purpose of doing some calculations to validate our system integration and operation. To better interface with the available data and control registers, a structure in C was written to behave as a driver, linking register names to the corresponding memory offset from the base address of the core. Each CORDIC core instance will have one of this structure beginning at the base address of the instance as specified at the AHB3-lite interconnect.

3.6. AHB3-lite/Wishbone conversion interfaces

While AHB3-lite is the protocol used by the Cortex-M0 MCU present in our design, several other protocols are available for in-chip communication, including the popular Wishbone protocol, which purpose and applicability are similar to AHB3-lite protocol. Given the similarity between these two protocols, we wrote a bridge between them in a way to create a compatibility between Wishbone-exclusive and AHB3-lite-exclusive cores. This bridge essentially translates the signal activation routine between the two protocols, converting the two signal unidirectional flows: AHB3-lite-to-Wishbone and Wishbone-to-AHB3-lite.

3.7. AHB3-lite DMA

The DMA controller is a specialized peripherals core that is under program control and is accessed by reading from and writing to addresses in the peripheral address space. DMA controllers can be programmed to transfer data between memory and peripherals synchronously with either external or temporal events, acting as a master device at

the interconnect bus. Since DMA controllers are quite flexible and versatile about their operating modes, we may use them as a way of reliving the MCU from expensive memory transfer operations, leaving the MCU to perform more meaningful tasks or sleep to reduce energy consumption while the transfer takes place. We implemented an AHB3-lite compatible DMA controller [Finotti 2019a], using the Wishbone DMA developed by [Usselmann 2001] as a base. Since the original implementation was non-functional, we debugged and fixed it in a way to allow the expected behavior. Finally, to use the DMA controller with the AHB3-lite protocol present in our design, we wrapped the wishbone DMA controller with the AHB3-lite/Wishbone conversion interfaces detailed at Section 3.6.

4. Software components

4.1. ARM GCC, building options and Makefile

As detailed on Section 3.1, the ARM Cortex-M0 processor used in our soft-microcontroller works almost exclusively with a 16-bit instruction set called ARM Thumb instruction set. To convert a program written in a high-level language such as C into a series of instructions understandable by the MCU, we need a set of tools which shall translate the text files containing high-level code into object files, and then link them together into an executable file containing machine code. These tools, which are known as compiler and linker, are generally provided by the MCU manufacturer, as there is a close relationship between the MCU hardware architecture and the machine code instructions supported by it. Furthermore, while a good compiler will result in efficient implementations of the high-level program, bad compilers will result in poor performance or even in a non-functional program. For this project, the GNU ARM Embedded Toolchain were be used for code compiling, linking, and debugging due to its comprehensive support to ARM processor architectures, the popularity of GCC, and all the advantages present in an open-source tool.

4.2. FreeRTOS

To keep up with the increasing complexity of the design, we decided to use a real-time operating system (RTOS) to run the tasks to be performed by the processor. Among the possible RTOS solutions available in the market, we chose to use FreeRTOS due to its popularity, simplicity, and because it is distributed freely under the MIT open source license. FreeRTOS is a market-leading RTOS for microcontrollers and small microprocessors, designed to be small, simple, and easy to port to different architectures. The kernel itself consists of only three C files, providing methods for multiple threads or tasks, mutexes, semaphores, software timers, and thread priorities management. Also, a tick-less mode is provided for low power applications.

We created a FreeRTOS port compatible with our platform that can run tasks and interface with the boards' peripheral components.

4.3. Compiled code extraction for RAM initialization

Once the code is compiled and linked into an executable file, we must integrate it into our HDL design in such a way that the MCU can access the program instructions and begin its operations when the FPGA is powered up. That is accomplished by a python script that

reads the binary and converts every byte into ASCIIfied binary text, dumping it into a text file. This text file is used to infer the initial values of the general-purpose SRAM detailed in Section 3.3, which will be used as the ROM memory of the soft-microcontroller.

5. Experiments and results

This section presents the results obtained at the ARM Cortex-M0 soft-microcontroller implementation. The HDL designs were designed to experiment with the essential features of embedded devices. In each of them, a C program was compiled and loaded in the appropriate memory addresses so that the Cortex-M0 could execute a routine to validate the experiment in question. The program execution and the HDL hardware behavior were validated through simulation in Vivado before the implementation in real-hardware on a KC705 board. A utilization report of the resources usage after HDL implementation in Vivado is shown in Table 1, while power characteristics are depicted in Table 2.

Table 1. Resources utilization report on a Kintex-7 (part XC7K160TFFG676-2).

Resource	Utilization	Available	Utilization %
LUT	15432	101400	15.22
LUTRAM	8705	35000	24.87
FF	3287	202800	1.62
DSP	3	600	0.50
IO	11	400	2.75
BUFG	6	32	18.75
PLL	1	8	12.50

Table 2. Power report on a Kintex-7 (part XC7K160TFFG676-2).

Total On-Chip Power	0.233W
Junction Temperature	25.4 ^{o}C
Thermal Margin	59.5 ^{o}C (30.5 ^{o}C)
Effective ΘJA	1.8 ^{o}C/W

5.1. Bare-metal blinky experiment

This experiment proposes to execute a simple LED blinking program, where the system periodically toggles a GPIO that drives an LED. The complete code is available at [Finotti 2019b]. The simplicity of the experiment allows us to focus on the system itself, validating all stages responsible for its operation. On the software side, we have the compilation of the program in binary code, the linking with the startup and vector table files, and the code extraction from the resulting ELF executable into an ASCII file, which will be used to initialize the memory HDL core. On the synthesizable hardware side, we have a Cortex-M0 processor, an AHB3-Lite interconnect, a RAM and ROM memories, and a pattern detector responsible for identifying the pattern on the bus that will cause the LED toggle.

The program coded for this experiment increments a variable until a manually tuned value so that the toggle period is 2 seconds. Upon reaching the specified value, a

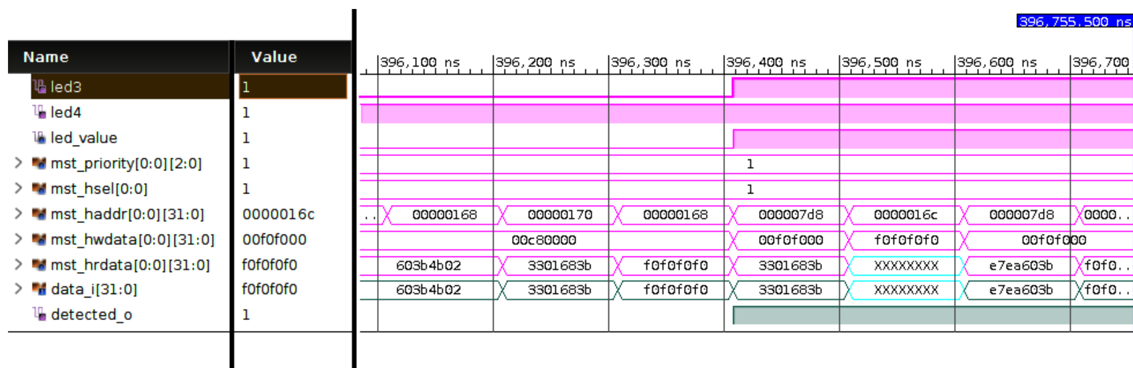

Figure 3. Blinky bare-metal code waveform running on ARM soft-microcontroller. Notice *led3* toggling during simulation.

second variable is assigned the pattern `f0f0f0f0`. As a consequence of this assignment, the processor reads this pattern from ROM before writing it to an address on the RAM, causing this pattern to appear on the read data bus. The pattern detector, when detecting this value, toggles the LED. A code snippet with the main part of the program is shown on Listing 1.

Listing 1. Code snippet of LED toggling routine.

```
while(1)
{
  counter=0;
  for(ii=0; ii<period; ii++)
  {
    counter++;
  }
  trap=LedToggle; // memory access pattern (toggle LED)
  trap++; // force toggle value to change value
}
```

The simulated hardware waveform, which is depicted on Figure 3, shows the exact moment when the pattern `f0f0f0f0` appears on `mst_hrdata[0:0][31:0]` (Master read-data bus). This value is read by the pattern detector input `data_i[31:0]` and, in the clock cycle immediately after that, the pattern detector toggles its output `detected_o`, which goes to `led_value` and `led3`. After that, the pattern also appears on `mst_hwdata[0:0][31:0]`, which corresponds to the write operation of the variable assignment.

5.2. Bare-metal busy-wait experiment

In this experiment, the busy-wait functionality is demonstrated by configuring four CORDIC cores to perform arithmetical operations and busy-waiting their conclusion before proceeding to the same routine specified at Section 5.1. The complete code is available at [Finotti 2019c]. The four CORDIC cores are created as slaves and connected to the AHB3-lite interconnect in different hardware addresses. The busy-waiting is illustrated by Listing 2, where a single CORDIC core is polled before proceeding to the conventional LED toggling routine. In our simulation and implementation, four CORDIC cores are set similarly.

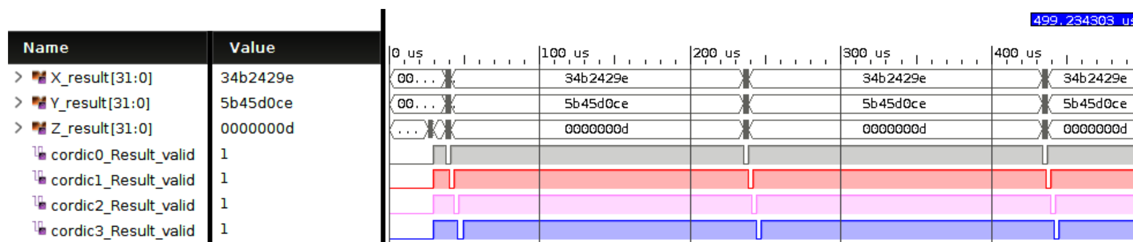

Figure 4. Busy-wait bare-metal code waveform running on ARM soft-microcontroller.

Listing 2. Code snippet where busy-wait is done for a single CORDIC core operation before proceeding to LED toggling routine.

```
// Calculate sine and cosine of pi/3
CONTROL_START0 = 0;
X0 = 1073741824; // 1 in Q1.30 notation
Y0 = 0;
Z0 = 715827883; // angle pi/3, since 2pi = 2^32
CONTROL_START0 = 1;

// Wait for calculations results
while(CONTROL_DONE0 != 1)
{}

// count until period and put the pattern that toggles the LED on the
    bus
counter=0;
for(ii=0; ii<period; ii++)
{
    counter++;
}
trap=LedToggle; // memory access pattern (toggle LED)
trap++; // force toggle value to change value
```

After the simulation, which is shown on Figure 4, we can see how the four different CORDIC cores proceed with their calculation before the toggle. Throughout the simulation we can observe that, for each of the CORDIC cores, the signals `X_result[31:0]`, `Y_result[31:0]` and `Z_result[31:0]` change from a stable value and start varying at the same time `Result_Valid` signal is set to `False`, indicating the moment the core begins its operation. On the waveform, it is notable the moment each core begins its operations, with CORDIC0 starting first followed by CORDIC1, CORDIC2, and CORDIC3. Immediately after the last core finishes its calculation, the program proceeds to the LED toggle routine, which causes `led3`'s value to change before the program routine is executed again.

5.3. FreeRTOS DMA experiment

This experiment tested Direct Memory Access (DMA) data transfer using an AHB3-lite DMA controller core between two generic memories while keeping CORDIC operations and LED toggling. The complete code is available at [Finotti 2020b]. To keep up with the increasing complexity of our experiments, FreeRTOS was introduced in the software

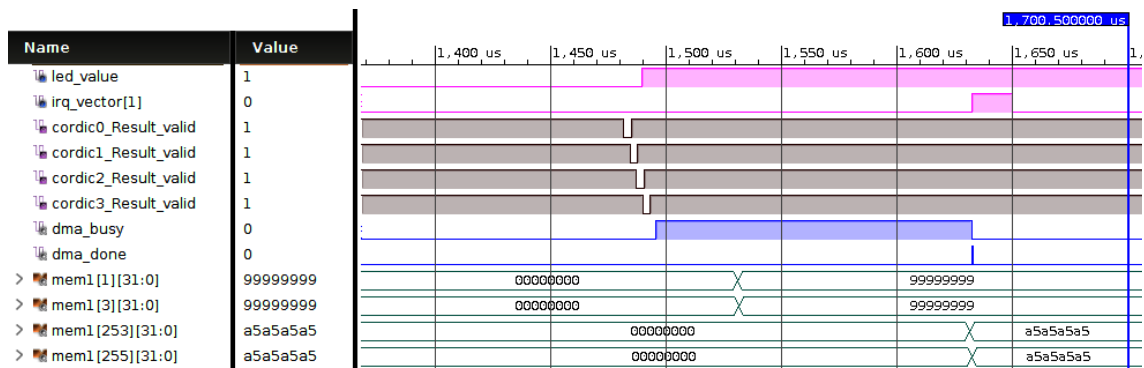

Figure 5. DMA operation on FreeRTOS waveform running on ARM soft-microcontroller.

side of the application, translating the bare-metal routine into tasks to be managed by the scheduler. A FreeRTOS port was also created to bridge the software side and the hardware was implemented in HDL, therefore creating a complete workflow. The HDL design is composed of the Cortex-M0 processor, four CORDIC cores, two memories, and a DMA core.

To test a DMA transfer, we created a FreeRTOS task that starts all CORDIC cores, toggles the pattern responsible for toggling the LED, and configures the DMA core to transfer data from one RAM to the other. After that, the tasks wait for a defined period and then consumes a semaphore, entering in a blocked state. This procedure is shown on Algorithm 1.

Algorithm 1 Main FreeRTOS task routine that sets CORDIC cores, DMA transfer and LED toggle.

```
 1: loop
 2:     Configure and start CORDIC0 core
 3:     Configure and start CORDIC1 core
 4:     Configure and start CORDIC2 core
 5:     Configure and start CORDIC3 core
 6:     Assign toggle pattern to variable        ▷ Pattern detector toggles the LED here
 7:     Configure DMA                            ▷ Transfer data from mem0 core to mem1 core
 8:     vTaskDelay(period)                       ▷ Delay between LED blinks
 9:     xSemaphoreTake()                         ▷ Semaphore that waits for DMA "done" interrupt
10: end loop
```

Once the main task finishes running, it waits indefinitely until the semaphore is given, which happens in an interrupt task triggered by an interrupt generated once the DMA core finishes its transfer operation.

The resultant waveform of this experiment is shown on Figure 5, where it is possible to see the moment where all 4 CORDIC cores begin and finish their operation, and where the DMA core enters a busy state, indicating it began the memory transfer. We can see where mem1 begins to receive the data "99999999_{hex}" in mem1[1] and mem1[3], and where it finishes, at mem1[254] and mem1[255]. Once the operations finish, the interrupt is raised at bit irq_vector[1], staying raised until it is cleaned by the interrupt task.

6. Discussions

This project presented an open-source generic architecture build around an ARM Cortex-M0 processor, which can be easily adapted for different purposes and applications. We implemented an end-to-end framework, from software compilation and linkage using ARM GCC to the conversion to a binary compatible with the memory cores used in the HDL design.

We validated the design both in simulation and synthesis in a comprehensive set of experiments, covering relevant aspects for embedded system applications. This design poses an alternative to vendor-specific solutions soft-microcontroller solutions, using the popular ARM architecture instead, or to SoC platforms, where the fixed microcontroller higher performance comes at a cost in flexibility and reconfigurability.

Our design might be used for FPGA applications demanding microcontroller functionalities, without the commitment of using SoC architectures or vendor-exclusive soft-cores. It might also be used in educational environments since, through simulation, one might analyze the behavior of the hardware at the registers level, observing how the internal registers of the Cortex-M0 and any of the peripherals behave over time. The possibility of compiling an elaborate program using our GCC-based framework, or using FreeRTOS in the design represents an invaluable resource for a comprehensive understanding of embedded systems, from software compilation to hardware execution.

References

ARM (2009). Cortex-M0 Devices Generic User Guide. https://developer.arm.com/documentation/dui0497/latest/. Last accessed on Sep 07, 2020.

ARM (2020). ARM DesignStart Program. https://www.arm.com/resources/designstart. Last accessed on Sep 07, 2020.

Brown, G. and Himebaugh, B. (2016). Computer Structures with the ARM Cortex-M0. https://legacy.cs.indiana.edu/~geobrown/c335book.pdf. Last accessed on Sep 07, 2020.

Caulfield, A. M., Chung, E. S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.-Y., Lo, D., Massengill, T., Ovtcharov, K., Papamichael, M., Woods, L., Lanka, S., Chiou, D., and Burger, D. (2016). A Cloud-Scale Acceleration Architecture. page 13.

Contardo, D., Klute, M., Mans, J., Silvestris, L., and Butler, J. (2015). Technical Proposal for the Phase-II Upgrade of the CMS Detector. https://cds.cern.ch/record/202088. Last accessed on Mar 01, 2021.

Finotti, V. (2019a). AHB3-Lite DMA core. https://github.com/vfinotti/ahb3lite_dma. Last accessed on Sep 07, 2020.

Finotti, V. (2019b). Blinky C example code for Cortex-M0. https://github.com/vfinotti/cortex-m0-blinky-c. Last accessed on Sep 07, 2020.

Finotti, V. (2019c). Busy-wait polling C example code for Cortex-M0. https://github.com/vfinotti/cortex-m0-busy-wait-c. Last accessed on Sep 07, 2020.

Finotti, V. (2020a). Cortex-M0 implementation on a Kintex-7 FPGA. https://github.com/vfinotti/cortex-m0-soft-microcontroller. Last accessed on Sep 07, 2020.

Finotti, V. (2020b). DMA example code with FreeRTOS for Cortex-M0. https://github.com/vfinotti/cortex-m0-freertos-dma-c. Last accessed on Sep 07, 2020.

Gisselquist Technology, L. (2021). ZipCPU: A small, light weight, RISC CPU soft core. https://github.com/ZipCPU/zipcpu. Last accessed on Jun 03, 2021.

Intel (2020). Nios® II Processors for FPGAs - Intel® FPGA. https://www.intel.com/content/www/br/pt/products/programmable/processor/nios-ii.html. Last accessed on Sep 07, 2020.

Intel (2021). Intel® FPGA Products - FPGA and SoC FPGA Devices and Solutions — Intel. https://www.intel.com.br/content/www/br/pt/products/details/fpga.html. Last accessed on Jun 03, 2021.

Jayakrishnan, V. and Parikh, C. (2019). Embedded Processors on FPGA: Soft vs Hard. page 8.

Martos, P. I. and Baglivo, F. (2011). Implementing the Cortex-M0 DesignStart Processor in a Low-end FPGA. page 4.

Nolting, S. (2020). The neorv32 risc-v processor. https://github.com/stnolting/neorv32. Last accessed on Jun 03, 2021.

OpenCores (2021a). aeMB. https://opencores.org/projects/aemb. Last accessed on Jun 03, 2021.

OpenCores (2021b). Amber ARM-compatible core. https://opencores.org/projects/amber. Last accessed on Jun 03, 2021.

RoaLogic (2020a). AHB-Lite Memory. https://github.com/RoaLogic/ahb3lite_memory. Last accessed on Sep 07, 2020.

RoaLogic (2020b). AHB-Lite Multilayer Switch. https://github.com/RoaLogic/ahb3lite_interconnect. Last accessed on Sep 07, 2020.

Thibedeau, K. (2020). VHDL-extras Library. https://github.com/kevinpt/vhdl-extras. Last accessed on Sep 07, 2020.

Usselmann, R. (2001). WISHBONE DMA/Bridge IP Core. https://opencores.org/projects/wb_dma/. Last accessed on Sep 24, 2020.

Xilinx (2020). MicroBlaze Soft Processor Core. https://www.xilinx.com/products/design-tools/microblaze.html. Last accessed on Sep 27, 2020.

Xilinx (2021). Zynq-7000 SoC. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. Last accessed on Jun 03, 2021.