A Low-Area Direct Memory Access Controller Architecture for a RISC-V Based Low-Power Microcontroller

Hanssel Morales, Ckristian Duran, and Elkim Roa
Integrated Systems Research Group - OnChip, Universidad Industrial de Santander, Bucaramanga - Colombia
{hanssel.morales, ckristian.duran}@correo.uis.edu.co, efroa@uis.edu.co

***Abstract*—In this work, we present a low area DMA controller that enables low-cost SoCs where subsystems need constant memory access. Small interfaces and a unique FIFO handling read/write transactions are fundamental blocks in this design. As proof of concept, the testing system also includes a RISC-V RV32IM processor, a USB 1.1/2.0 PHY and a QSPI interface. We implemented a whole microcontroller using a TSMC 0.18μm technology node, where the DMA occupies 4.2% of the total area. The results show a total DMA area of 1997 gates using 4 information channels, which is 75.3% smaller area in comparison with recent low-area DMAs.**

***Index Terms*—Direct Memory Access, DMA, RISC-V, Microcontroller, IoT**

I. Introduction

Devices containing high-speed communication protocols and high information density sensors require constant reading and writing access to memory. Programmed drivers commonly use the main processor in most of the internet-of-things (IoT) applications to transport communication data. A driver issues interactions between the processor and the system memory with asynchronous tasks often demanding processor interruptions. Excessive bus transactions consume excessive power and interruptions cause delays that impact processor performance. In addition, required firmware may limit available memory for information writing and reading.

Numerous SoC implementations require direct memory access (DMA) in order to generate efficient data transport. DMA controllers (DMAC) avoid processor usage for asynchronous data, engaging the processor just to write a few configurations for each incoming request. It is becoming popular that low-cost SoC includes DMA controllers to operate I/O data interfaces [1]. Information transactions from low-speed peripherals could be efficiently implemented using DMA, as UART, I2C, and SPI should not use internal buffers to store asynchronous information. Fig. 1 shows different bandwidth requirements for different communication protocols. Protocols like USB 3.1 requires 10 Gbps and widely used USB 2.0 needs 480 Mbps.

In all cases, building a DMA controller should be capable of handling such information rates, and the main memory should increase word width and length depth to store information according to the protocol definitions. The DMAC consumes considerable power and area depending on the implemented design to fulfill these requirements. Defining a DMAC architecture is a challenging task for IoT applications as the main system frequency should increase for handling high information rates, increasing power consumption. Area is a primary concern because DMA controller logic should be capable to store pending information in caches or high-depth FIFOs while handling multiple peripheral requests.

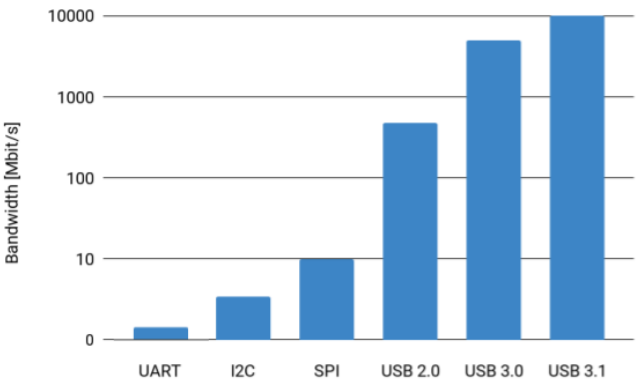

Fig. 1. Bandwidth requirement for different communication protocols.

In this work, we propose a low-area DMA controller with multiple information channel support. This design is applicable to low-power SoC thanks to the capability of enabling, disabling, and limiting transactions over a peripheral. Strategies used in this paper include reducing channel request bit width, less storage in FIFO width, and on-the-fly address calculations. We discuss comparisons with a recent DMAC IoT implementation with similar architecture and testing environment. Results show a possible implementation for meeting bandwidth requirements for high-speed USB 1.1/2.0 and super speed USB 3.0/3.1 without increasing architecture width. Finally, we summarize results showing gate area and ratio compared to the system implementation.

II. Related Work

A common DMA architecture is shown in Fig. 2 [2], [3], [4], [5]. The channel logic multiplexes the requests using an arbiter, thus defining a priority between peripherals. The FIFO stores these channel requests that will decode characteristics for bus transactions, handling the order of execution on the bus interface. When bus transactions finish, the DMA sends an acknowledgment to each channel, returning data from the memory in the case of reading transactions. DMA controller often separates reading and writing logic to ease usage and implementation.

In IoT applications, sensor control implements DMA to perform information storage, as proposed in [6]. In this case,

978-1-7281-0453-9/19/$31.00

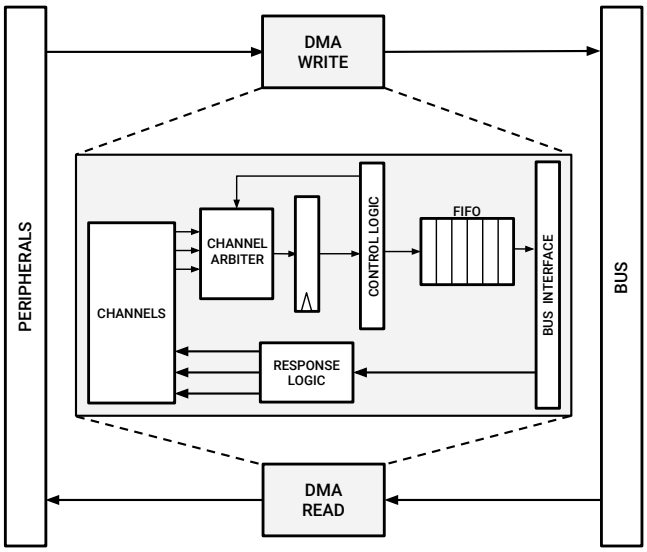

Fig. 2. Traditional DMA architecture.

DMA-controlled instructions replace the CPU instructions for data transfer. Although it states a power reduction, there is no discussion about the DMA construction because this work implements a commercial micro-controller to prove functionality. The DMA architecture can impact performance depending on the chosen micro-controller. Another implementation of low-power using DMA is exposed in [7], where a high-efficient DMA along with a media access controller (MAC) is implemented.

Recent IoT designs lack definitions about DMA architectures. The literature exposes only some details about the DMA construction. In [2], Lei *et al.* describe a first DMA construction defining a Wishbone bus implementation, where it is proposed an architecture that allows high-speed access through different clock domains and access to the internal processor memory. Another study of DMA technique, shown in [4], gives a solution to the bottleneck problem by using an address generation optimized for DMA controllers. In [5], Sharma *et al.* expose finite state machine decomposition to achieve a low-power AHB-based DMA controller.

In [3], Pullini *et al.* expose an architecture with reports of good area in function of connected peripherals, known as channels. Bandwidth is also reported using some protocols to implement this proposal. The results compare area reports with commercial microcontrollers. The implemented subsystem contains L2-caches, miscellaneous peripherals and a RISC processor. With this integration, a low area design is possible, as μDMA occupies up to 10.1% of the total subsystem area.

III. Proposed Architecture

Fig. 3 shows a scheme of the main architecture of the DMAC. Each peripheral can issue read and write transactions using single or multiple channels to perform memory access. A unique FIFO stores all granted transactions, where each channel is notified using a valid-ready handshake if the transaction is performed. The FIFO stores an ID to identify the channel to acknowledge, the type of transaction, and the write data for write operations. An arbiter (green) multiplexes the transactions among all channels with equal priorities. The channel interface drives few signal ports to save area. The signals that compose a channel interface are: read/write flag (R/W), write data (WDATA), and valid-ready signals.

For an N-bit implementation and M channels, the equation (1) defines the FIFO word length. Although common architectures, the FIFO does not store the addresses for each transaction. The FIFO stores the data using a double-channel RAM or using registers, depending on the construction parameter flags. Independent of the implementation, the FIFO is reduced to its minimum caused by the word width defined (1), storing fewer transaction characteristics and dynamically calculating operations on the bus with pre-configured channels.

$$length_{FIFO} = 1 + N + \log_2 M \tag{1}$$

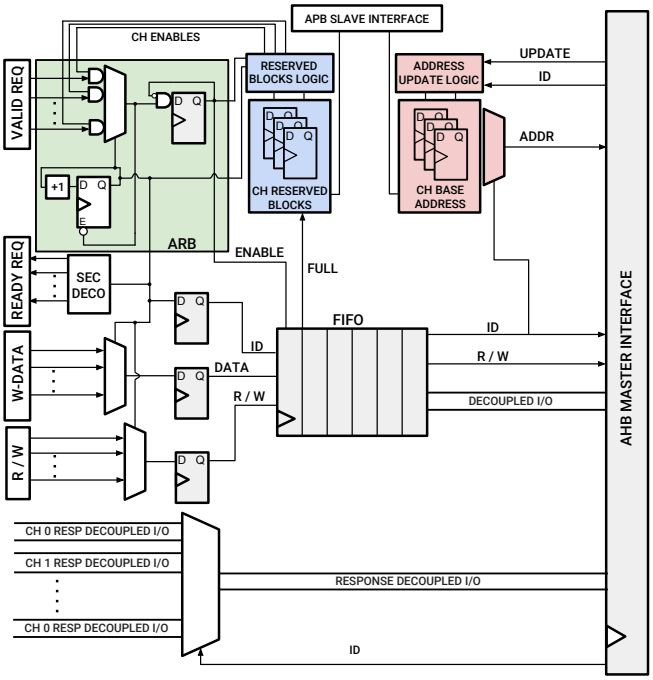

Fig. 3. Proposed DMA architecture.

The channels must be pre-configured using an APB-interface, storing parameters like starting address offset and the size of the data. Bus transaction address calculations use these parameters as inputs, and are represented in Fig. 3 as: the reserved blocks logic (blue) and the address update logic (red). Internal counters inside the address update logic calculate transaction addresses. For each particular channel, configuring target address and size of the data initializes calculations in these counters. Transactions are not stored in the FIFO if the reserved update logic (blue in Fig. 3) detects that reserved blocks hit zero. Reserved block detection program on a particular channel should perform reconfiguration to

avoid limitations to the peripheral. The target peripheral should notify the processor using interrupts, as this DMA controller does not trigger processor interruptions to allow application flexibility.

Pipeline stages perform operation phases on DMA transactions. Fig. 3 highlights in grey the pipeline stages. There are 3 pipeline stages shown in Fig. 4 that can be executed at time where the AHB interconnector is not busy. DMA pipeline stage fetches the transaction, which decides if the FIFO stores the transaction using the reserved update logic. The FIFO logic is the second pipeline stage, and its internal logic outputs the first arrived transaction. The address update logic manages the third stage, issuing the transaction to the AHB master interface. The AHB interface has an implicit pipeline stage, where acknowledgment and read transactions are transported to the response decoupled I/O channel.

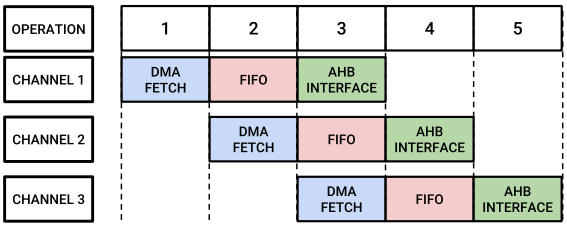

Fig. 4. DMA pipeline stage execution.

The microcontroller architecture implements the proposed DMA according to Fig. 5. The main processing unit is a RISC-V processor with RV32IM instruction set. USB 1.1/2.0 and QSPI interfaces implement fast communication features to test information transactions with this DMA proposal. The main bus has an AHB protocol and interconnector, attaching the main system memory (SRAM), the permanent memory (ROM), and a debug module for bus monitoring. The peripheral bus has APB protocol for low-speed peripherals used to configure registers in GPIO, QSPI and USB 1.1/2.0, and PWM. [8]. Chisel is the hardware description language (HDL) used to allow parameterization of the DMA and ease dynamic integration of the whole system [9].

IV. Implementation Results

The proposed circuit is synthesized using a TSMC 0.18μm technology node. Parameters of DMA as FIFO depth and a variable number of channels are fixed. Fig. 6 shows area results in function of the channels, where increasing the number of channels increases the number of gates in linear proportion. Table I shows a comparison with μDMA [3]. We perform the comparison as an area reduction between μDMA and this proposal using gate equivalent (GE) values. In the best case, our proposal shows a decrease of gates around 75.3% in 8-channel configuration.

In this design, increasing the number of channels does not impact the area in proportion. Due to the channels share all resources, the number of gates per channel ratio decreases as channels increase as shown in Fig. 7. A FIFO may need larger

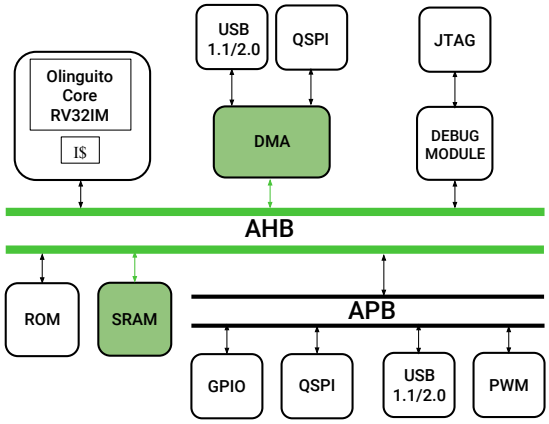

Fig. 5. Subsystem architecture.

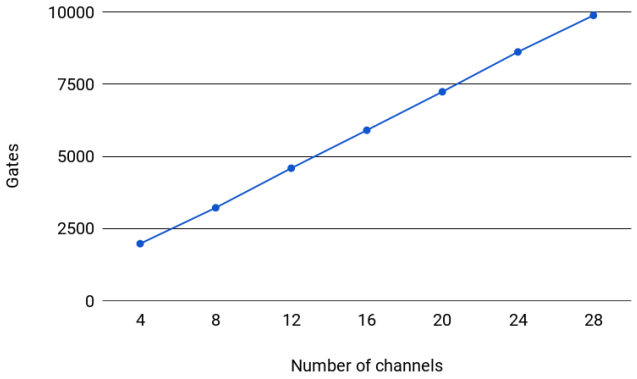

Fig. 6. Area in gates in function of the implemented channels.

depth when increasing the channels to store different transactions that can critically compromise the area. The processor can configure initially the DMA-enable registers or transaction limit counters to clamp resource usage into the channels. This avoids transactions overloads without compromising area usage caused by limited FIFO depth.

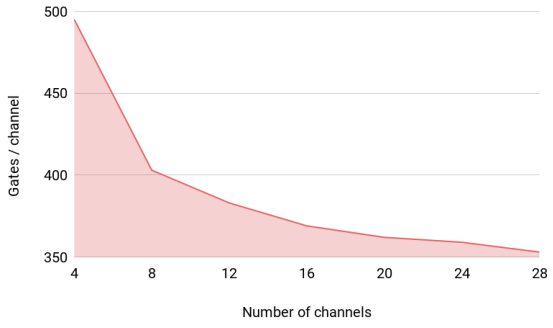

Fig. 7. Gate ratio in function of the number of channels.

Table II tabulates the largest bandwidth support of the proposed DMA. The current implementation in the microcon-

TABLE I
COMPARISON BETWEEN [3] AND OUR PROPOSAL.

Number of channels	This work [GE]	μDMA [GE]	Area reduction [%]
4	1997	8000	75.0
8 (best case)	**3220**	**13000**	**75.3**
12	4590	14500	68.3
16	5904	18000	67.2
20	7236	22500	67.8
24	8616	26000	66.8
28	9881	30240	67.3

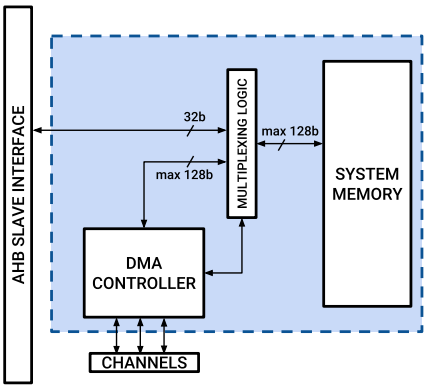

Fig. 8. System memory integration inside DMA module.

troller has a 32-bit data width at 48MHz. This is enough for supporting USB 1.1 and 2.0 according to Fig.1. For enabling USB 3.0 and 3.1, the DMA must support higher bus widths and frequencies. A comparison with [3] is performed also in Table II. Our proposal is intended to single-transaction bus support, therefore the bandwidth reported in μDMA is halved due to its RX and TX simultaneous support. Fig. 8 presents a main memory integration into the DMA architecture. The DMA module provides an AHB slave interface for regular bus read/write operations to the memory. In this approach, only the input and output data buses from the memory are increased, and the AHB slave interface has remained constant. For supporting high-speed interfaces, this implementation avoids increase into the system data bus width, where is compensated into the increase of DMAC internal memory. When writing or reading information to the memory, the DMAC puts a temporary wait state the AHB slave interface, blocking system memory operations if the processor execution process requires them. Using a frequency of 96MHz, and data widths superior to 64 bits, we are capable of meet bandwidth requirements for faster USB implementations.

Fig. 9 shows the final layout for a TSMC 0.18μm technology node. The figure highlights in pink the occupation of the proposed DMA into the microcontroller. The layout is performed with a fixed configuration of USB 1.1/2.0 interface in two endpoints. The number of required channels to handle such configuration is six. The proposed DMA occupied an area ratio of 4.2% compared to the microcontroller.

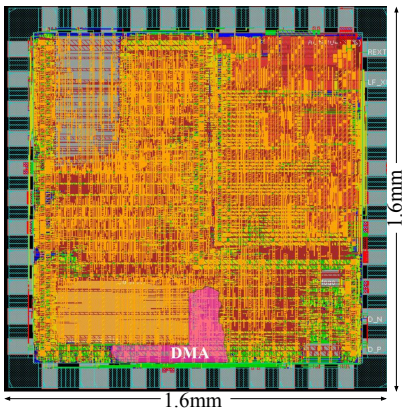

Fig. 9. Final layout of the implemented microcontroller, highlighting the area occupied by the DMA.

TABLE II
BANDWIDTH SUPPORT AND COMPARISON BETWEEN OUR DMA AND [3]

Memory Data Width	Bandwidth [Gbit/s]	μDMA Bandwidth [Gbit/s]
32	1.5@48MHz	1.6@50MHz (Single channel)
64	6.1@96MHz	N/A
128	12.2@96MHz	N/A

V. SUMMARY

In this work, we presented a low-area DMA architecture using a single FIFO. DMA channel configuration feature enables the design for low power applications. Shared resources and word width reduction enables low-are consumption. Results show up to 75.3% area savings compared to other area optimized DMA. In addition, integration with a RISC-V based microcontroller featuring a USB 1.1/2.0 and a QSPI tests the correct proposal behavioral. Finally, the DMA only uses 4.2% of the total chip area

REFERENCES

[1] *Atmel SAM D21E / SAM D21G / SAM D21J Datasheet"*, Atmel, 9 2015.
[2] L. Lei *et al.*, "Design and Implementation of DMA Transfers in WISHBONE Interface," in *2015 10th International Conference on Communications and Networking in China (ChinaCom)*, Aug 2015, pp. 612–616.
[3] A. Pullini *et al.*, "μDMA: An Autonomous I/O Subsystem for IoT End-Nodes," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sept 2017, pp. 1–8.
[4] C. Brunelli *et al.*, "A Dedicated DMA Logic Addressing a Time Multiplexed Memory to Reduce the Effects of the System Bus Bottleneck," in *2008 International Conference on Field Programmable Logic and Applications*, Sept 2008, pp. 487–490.
[5] C. Sharma and D. K. Chauhan, "High performance low power AHB DMA controller with FSM decomposition technique," in *2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI)*, Sept 2017, pp. 456–461.
[6] T. Enami, K. Kawakami, and H. Yamazaki, "DMA-Driven Control Method for Low Power Sensor Node," in *2015 IEEE Topical Conference on Wireless Sensors and Sensor Networks (WiSNet)*, Jan 2015, pp. 53–55.
[7] X. Zhang *et al.*, "A Low-Power Remotely-Programmable MCU for Implantable Medical Devices," in *2010 IEEE Asia Pacific Conference on Circuits and Systems*, Dec 2010, pp. 28–31.
[8] ARM, *AMBA Specification Rev. 2.0*, 1999.
[9] J. Bachrach *et al.*, "Chisel: Constructing Hardware in a Scala Embedded Language," in *DAC Design Automation Conference 2012*, June 2012, pp. 1212–1221.