Low-level algorithm for a software-emulated I^{2}C I/O module in general purpose RISC-V based microcontrollers

Roberto Molina-Robles*, Ronny García-Ramírez*,
Alfonso Chacón-Rodríguez*, Renato Rimolo-Donadio* and Alfredo Arnaud†
*Escuela de Ingeniería Electrónica, Tecnológico de Costa Rica
†Depto. de Ingenieria Electrica, Universidad Catolica del Uruguay
{rmolina, rgarcia, alchacon, rrimolo}@tec.ac.cr
aarnaud,@ucu.edu.uy

***Abstract*—To date, I^{2}C is a protocol that has a lot of use on microcontroller applications. Implementing the hardware for a I^{2}C controller affects directly chip size, cost and time development in VLSI projects. To save on these aspects, we have used hardware emulation by software to recreate the I^{2}C behavior through GPIO ports in a RISC-V microcontroller of our own. This paper proposes a flexible and practical low-level algorithm that replaces a physical I^{2}C controller, to be able connect an I^{2}C peripheral device to its driver code inside the microcontroller using GPIO ports as channels. An explanation of the application is described, along with the code structure and the logic behind the algorithm. At the end, this paper presents results with a brief analysis of its benefits and future improvements.**

***Index Terms*—I^{2}C bus protocol, hardware emulation, RISC-V, microcontroller, algorithms, architecture, I/O protocols, hardware-software applications.**

I. Introduction and related work

I^{2}C (Inter-Integrated Circuit) is a communication protocol [1] that has been used continuously through the years in digital applications since its conception. In today's applications, I^{2}C is widely used in communications, especially for microcontroller applications where several I/O devices needs to be driven. Fields like the medical industry, space technology, public security, automation controls, heterogeneous networks and telecommunications are fine examples where I^{2}C is constantly employed. Hence, many microcontroller design teams considerate using I^{2}C to drive peripheral devices. However, developing a microcontroller from scratch is a daunting task, where many difficult decisions and trade-offs happen at every step. In VLSI design, power consumption, development time, cost and chip size are topics of interest that usually affects the quantity of features and components that can be added to a final product. As any other hardware element, if an I^{2}C controller is added to a chip, all the aspects mentioned before are impacted negatively. In this kind of challenges, industry and academy researchers are always on the lookout for alternative solutions that can save on chip size and other resources.

Hardware emulation is a technique that has been around in electronics design for several decades, and we could say that it consists in using a hardware device to mimic the functionality of another, whatever the goal. For instance, [2] uses hardware emulation in a traffic network for "Internet of Things" applications. [3] brings a hardware software co-designed solution for a decoder of an x86 emulated system. In [4] it is described that hardware emulation on FPGA can be used to improve testing methods, and in [5] hardware emulation is used to measure and characterize network architectures. We have pondered for some time about the idea of using software-emulated hardware solutions to reduce the final chip size and the development time of our project: [6], [7], [8], [9] and [10]. As such, we decided to emulate by software an I^{2}C controller using GPIO (General Purpose Input-Output) ports, to be able to manipulate I^{2}C external devices without implementing the I^{2}C controller hardware into the microcontroller.

Delving into the state of the art of I^{2}C bus protocol, one can find solutions about applications of said standard. For example, in [11] and [12] an I^{2}C bus is used to interface I^{2}C peripheral devices to an FPGA. [13] presents an implementation on FPGA of an I^{2}C controller for secure data transmissions. In [14] an arbitrated multi-master multi-slave I^{2}C bus application is shown, and in [15] this communication protocol is used to form a network with actuators, sensors and microcontrollers. All these past papers show proper hardware implementations related to the I^{2}C communication standard. By adding the software component to the search formula, we can find articles like [16], where a software solution is included to build an I^{2}C bus analyzer. On other approaches, [17] use I^{2}C to collect information from a sensor network using an embedded system with Ubuntu as its operating system. However, there are similar implementations to what we wanted to achieve. For example, Texas Instruments showed on [18] a description about a software-emulated I^{2}C controller using, also, GPIO ports on a RISC family of microcontrollers. Microchip and Intel presented their software-emulated I^{2}C approaches in [19] and [20], respectively, in the corresponding assembly language for their chips. The implementation of our I2C controller emulated by software is similar to these last approaches, with the caveat that our application had to be written for our

2021 IEEE URUCON | 978-1-6654-2443-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/URUCON53396.2021.9647309

own RISC-V microcontroller, with a restriction on memory size since our main memory could hold up to 8 *KB* in total.

In summary, this paper proposes an alternative software algorithm for microcontrollers to emulate the I^{2}C communication protocol using incorporated GPIO ports, replacing the I^{2}C controller hardware implementation. The memory size cost of this algorithm is about 0.5 *KB* and its development time was moved to post-silicon phases. The development of this application was done in RISC-V 32I assembly code, using the open-source toolchain available online for said architecture [21].

II. The RISC-V Microcontroller Overview

As stated in the previous section, we have developed a general purpose RISCV-V microcontroller, [6], with the intend to use it for low-power medical applications [8]. We chose the 32I version as the ISA (Instruction Set Architecture) for our microcontroller and all its applications were created following this specification [22].

Three interfaces were incorporated to communicate with other type of I/O devices. First, an SPI (Serial Peripheral Interface) module to load the program from an external FLASH memory and boot the microcontroller. Second, an UART (Universal Asynchronous Receiver-Transmitter) module to communicate mainly with computers. And lastly, eight GPIO ports to drive different kind of devices or read sensors.

The I^{2}C controller algorithm in this article allowed us to virtually add a fourth interface at a low memory cost while using 2 GPIO ports. The application, described in the following sections, was written using the RISC-V 32I ISA. Figure 1 illustrates our microcontroller with these interfaces.

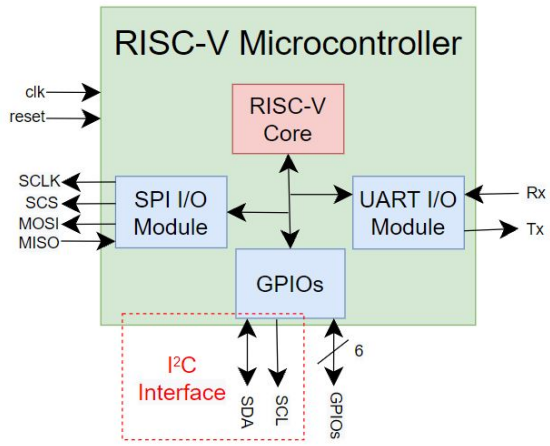

Figure 1. Overview of the RISC-V microcontroller and its interfaces.

III. Description of the Software Architecture and Methods

For our application, the emulation of the I^{2}C bus protocol had to be done in assembly language using GPIO ports and a limited memory space. Figure 2 illustrates the theory behind the concept for our application. As mentioned before, in the absence of the hardware controller for an I^{2}C device, we opted to develop an algorithm, written in RISC-V 32I assembly code, to load the commands of an LED Display from the main memory and send them over to the GPIO ports that emulated the I^{2}C.

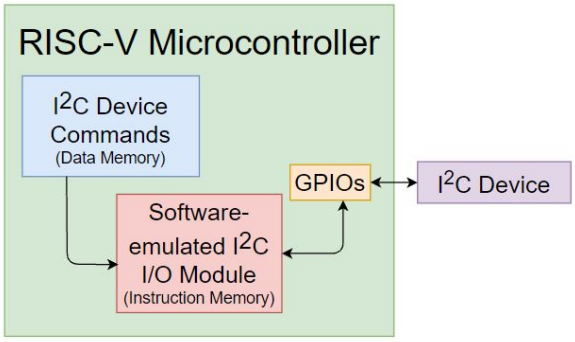

Figure 2. Concept idea and motivation for the development of the I^{2}C emulation application.

The proposed algorithm has a code structure described in Figure 3. This picture shows a representation of the code distribution inside the microcontroller's main memory. The code was allocated following the idea of a logically separated data and instruction memories. The data memory holds the commands of the I^{2}C peripheral device, namely, its driver, while the instruction memory holds the actual algorithm that emulates the I^{2}C standard. On the upper part of the main memory is located the microcontroller's interrupt service routine. Some space can also be reserved up there for a stack if needed. Since physical space and power consumption were important for our microcontroller, the main memory size was limited to 8 *KB*, hence, the totality of this application couldn't surpass that threshold.

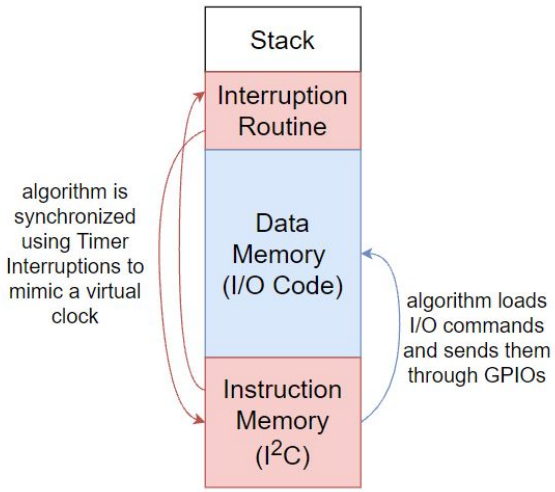

Figure 3. Code structure of the I^{2}C emulation application written in assembly code. The red arrows represent jumps in the code and the blue arrow represents reads on Data Memory to load the driver commands of the peripheral device to send them over the emulated I^{2}C.

Since this is a low-level software application and very close to the device hardware, a "time" concept was needed to a certain degree. Its algorithm was conceived in a similar way an FSM (Finite State Machine) is designed for digital control units. And just like a hardware implemented FSM, a clock was used to move through the states, though this clock was virtually implemented by software and its frequency could be adjusted to match the different speeds the I^{2}C bus standard permits. This FSM had to replicate a typical I^{2}C

Table I
FIGURE 4 STATE DESCRIPTION.

State Number	Description
1	Initial Setup (CSRs, interruptions, virtual clock freq.)
2	Load variables into register bank
3	Microcontroller setup and initialization.
4	Wait for timer interruption
5	Check I^{2}C transaction phase
6	I^{2}C device reset
7	Start I^{2}C transaction
8	Clear the SCL line
9	Set data bit or ACK on the SDA line
10	Set the SCL line
11	Hold and wait to complete a SCL cycle
12	Stop I^{2}C transaction
13	End-routine configuration

transaction. Figure 4 shows its graphical representation and Table I describes briefly what each of the states do in this algorithm.

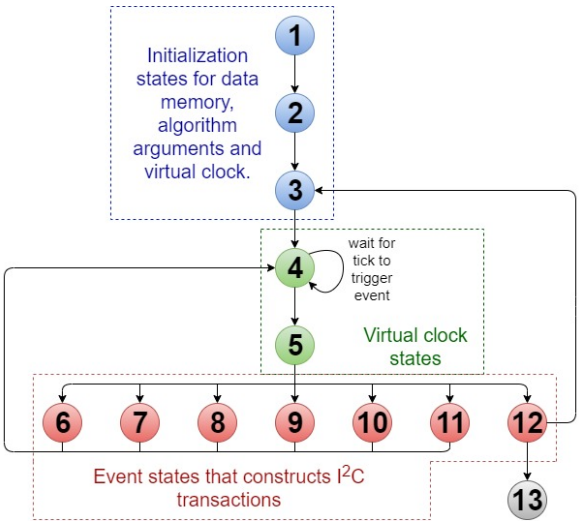

Figure 4. State diagram representation of the I^{2}C emulation algorithm.

The states 1, 2 and 3 (blue) are used to setup the microcontroller before the algorithm can use its resources. This setup includes tasks such as configuring the CSRs (Control Status Registers) and the interruption unit. Also, these states oversee the creation of local variables (managed at register-level) needed by the algorithm to adjust the virtual clock speed, internal counters, transfer packets, etc. States 4 and 5 (green) implement the virtual clock that marks the pace of the FSM and the evaluator that decides the route of the FSM after identifying the current phase of the transaction. These states take advantage of the microcontroller's internal timer to adjust the speed according to the application needs. States 6 to 12 (red) are the constructors of the I^{2}C transaction, each state represents a different phase in a single transfer.

Figure 5 illustrates a typical I^{2}C transaction, where commands or data bits are issued. Two lines are used for this transaction as specified by the I^{2}C standard: the SCL line created by the master (the microcontroller) to synchronize the transaction, and the SDA line where data bits or commands with address bits are sent back and forth. Both lines were implemented on GPIO ports and a third GPIO port was used for the I/O device's reset signal. For each transfer, eight address-commands/data bits are required plus an acknowledge bit to notify the sender if the capture was successful. In Figure 5 there are several numbers that matches the constructor states of Figure 4. Figure 5 helps understand the loops the FSM takes to emulate a complete I^{2}C transfer cycle. After a transaction has finished, the algorithm evaluates on a stop condition at state 12 if there are more packets to be sent and a new transfer is prepared, or if there are no packets left, the algorithm continues to the finalization routine to end the program or return to its caller.

IV. ANALYSIS AND RESULTS

After its implementation, the algorithm was analyzed to be able to understand better its advantages and features. Also, it helps us mark a roadmap for future updates. Several characteristics were studied to measure the algorithm capabilities like code size, adaptability to different I^{2}C devices and completeness of the I^{2}C standard.

Figure 6 shows an experimental result after the implementation of the proposed algorithm, in which three I^{2}C transactions occur: one command is issued, and two data bytes are sent afterwards. As explained in the previous section, two GPIO ports were used to emulate the SCL line and the SDA line from the I^{2}C bus, and a third GPIO port was used just to generate the I/O device reset. The acknowledge bits for each transaction, the start condition and the stop conditions are marked on the picture for a better understanding of the illustration. In the results section of [9], a picture is shown where an LED device was handled using this algorithm on a RISC-V microcontroller.

The code size was measured after compiling the program following Figure 3 code structure. Thus, the interrupt service routine implementation used 24 bytes and the instruction memory used 524 bytes. The data memory had a variable size since it contains the commands to drive the I^{2}C peripheral device. Considering that the whole main memory had a size of 8 *KB*, we could say that the I^{2}C emulation code only occupied around 1/16 segment of the available memory. In [18], the code size for a similar application was reported to be around 1 *KB*. In contrast, there is an improvement on this aspect versus that documentation.

The application was constructed with the capability to easily adjust the driver code depending on the connected I^{2}C device. Since the code to drive the I/O device is contained inside the data memory and separated from the emulated controller, it can be easily swapped before compiling with another code to command a different peripheral device. Hence, it could be said that the emulation algorithm can adapt to different I/O devices that use the I^{2}C protocol. However, there are two important drawbacks with its current implementation. First, the driver code is still needed inside the main memory before compiling and programming the microcontroller. And second, the current algorithm cannot function as a slave and receive information from an external device yet. These two aspects mark future improvements that can be included in newer versions of this application.

Regarding the completeness of the I^{2}C protocol, most features of the standard can be handled by the algorithm. The data transactions can be managed with the four speed grades dictated by the I^{2}C standard. The application can be adjusted to achieve these speed grades with an input argument that

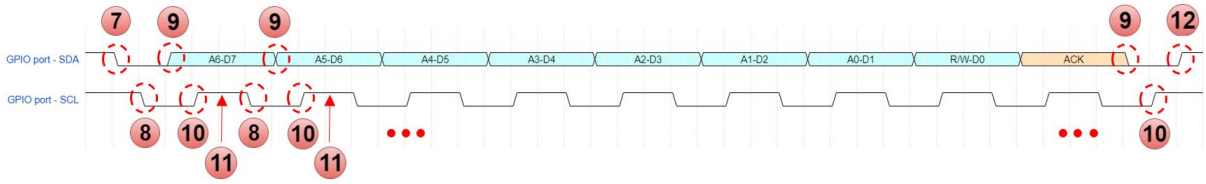

Figure 5. An I^{2}C transaction needed for peripheral addresses, commands and data. The numbers represent the states of Figure 4 diagram and Table I.

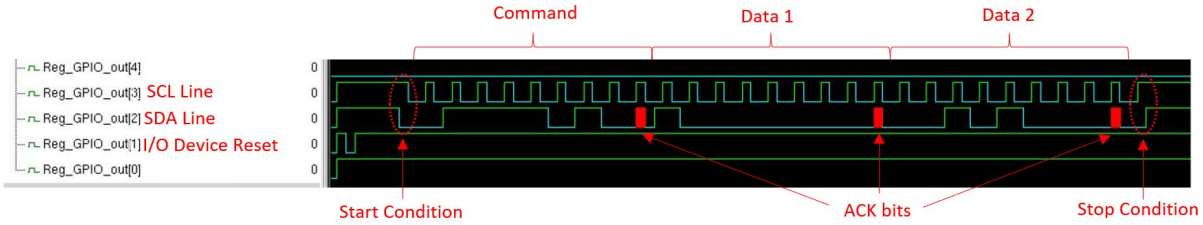

Figure 6. Resulting I^{2}C transaction after implementing the proposed algorithm.

modifies the frequency of the algorithm's virtual clock, given that the microcontroller's clock frequency permits such speeds. Equations 1, 2 and 3 describe this modifier.

$$1\ virtual\ clock\ tick = X\ \times \frac{1}{Freq.} \tag{1}$$

, where X is the number in the microcontroller's timer needed for a tick of the virtual clock inside the algorithm, and *Freq.* is the actual frequency of the microcontroller.

$$4\ virtual\ clock\ ticks = \frac{1}{SCL\ Freq.} \tag{2}$$

$$Speed\ Grade = SCL\ Freq. \tag{3}$$

, where *SCL Freq.* is the frequency of the SCL line and *Speed Grade* is the speed desired for the application. Let's assume that it is desired to use the I^{2}C bus standard-mode speed grade, whose transfer value is 100 *kbit/s*. If in one cycle of SCL line 1 bit is transferred through the SDA line, then the SCL frequency should be 100 *kHz* to transfer 100 *kbit/s*. From there, knowing the frequency of the microcontroller, it is possible to calculate the timer count to adjust the algorithm to the desired speed grade. There are features from the I^{2}C standard that are not included, like "multi-master" capability or "clock stretching", features that could be added in the future if the necessity arises.

As a final note about this application for I^{2}C emulation versus the proper hardware implementation of the I^{2}C controller. If chip size is considered relevant in a project, this application can be incorporated at the expense of a small memory fraction, given that the microcontroller has GPIO ports (or similar) that can reproduce the behavior of the I^{2}C. Also, the project cost and risk of developing the I^{2}C emulation is considerably lower than its hardware counterpart, since implementing the algorithm or fixing a bug can be done after fabricating the microcontroller.

V. Conclusions

This paper proposed a flexible low-level small-sized algorithm of a software-emulated I^{2}C controller for RISC-V microcontrollers to drive I^{2}C devices using GPIO ports as substitutes. Hardware emulation using software is an alternative for design teams when hard decision-making trade-offs arise between chip size, cost, time development and the necessity to include a feature, in this case, an I^{2}C bus controller. The paper described the logic behind this emulation scheme along with the motivation and background of its development. A detailed description of the construction of an I^{2}C transaction was also shown. This article also presented an experimental result of a proper I^{2}C emulation, along with an analysis about the benefits of this approach. Some future works we might consider as a result of this project are: improvement of the algorithm to include more features of the I^{2}C bus protocol, design of variations of the for other types of communication protocols and the development of an user interface to modify the arguments of the algorithm without the need of reprogramming the microcontroller.

References

[1] N. Semiconductors, "The i2c-bus specification and user manual," 2014, standard. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[2] Y. Kuwabara, T. Yokotani, and H. Mukai, "Hardware emulation of iot devices and verification of application behavior," in *2017 23rd Asia-Pacific Conference on Communications (APCC)*, 2017, pp. 1–6.

[3] S. Zhang, L. Jiang, X. Zhang, and X. Hu, "Hardware software co-design of pipelined instruction decoder in system emulation," in *2013 IEEE 4th International Conference on Software Engineering and Service Science*, 2013, pp. 149–153.

[4] T. Li and Q. Liu, "Cost effective partial scan for hardware emulation," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 131–134.

[5] M. Khamis, S. El-Ashry, A. Shalaby, M. AbdElsalam, and M. W. El-Kharashi, "A configurable risc-v for noc-based mpsocs: A framework for hardware emulation," in *2018 11th International Workshop on Network on Chip Architectures (NoCArc)*, 2018, pp. 1–6.

[6] R. Garcia-Ramirez, A. Chacon-Rodriguez, R. Molina-Robles, R. Castro-Gonzalez, E. Solera-Bolanos, G. Madrigal-Boza, M. Oviedo-Hernandez, D. Salazar-Sibaja, D. Sanchez-Jimenez, M. Fonseca-Rodriguez, J. Arrieta-Solorzano, R. Rimolo-Donadio, A. Arnaud, M. Miguez, and J. Gak, "Siwa: A custom risc-v based system on chip (soc) for low power medical applications," *Microelectronics Journal*, vol. 98, p. 104753, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0026269219303787

[7] R. Garcia-Ramirez, A. Chacon-Rodriguez, R. Castro-Gonzalez, A. Arnaud, M. Miguez, J. Gak, R. Molina-Robles, G. Madrigal-Boza, M. Oviedo-Hernandez, E. Solera-Bolanos, D. Salazar-Sibaja, D. Sanchez-Jimenez, M. Fonseca-Rodriguez, J. Arrieta-Solorzano, and

R. Rimolo-Donadio, “Siwa: a risc-v rv32i based micro-controller for implantable medical applications,” in *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*, 2020, pp. 1–4.

[8] A. Arnaud, M. Miguez, J. Gak, R. Puyol, R. Garcia-Ramirez, E. Solera-Bolanos, R. Castro-Gonzalez, R. Molina-Robles, A. Chacon-Rodriguez, and R. Rimolo-Donadio, “A risc-v based medical implantable soc for high voltage and current tissue stimulus,” in *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*, 2020, pp. 1–4.

[9] R. Molina-Robles, R. García-Ramírez, A. Chacón-Rodríguez, R. Rimolo-Donadio, and A. Arnaud, “An affordable post-silicon testing framework applied to a risc-v based microcontroller,” in *2021 IEEE Latin America Electron Devices Conference (LAEDC)*, 2021, pp. 1–5.

[10] R. Molina-Robles, E. Solera-Bolanos, R. García-Ramírez, A. Chacón-Rodríguez, A. Arnaud, and R. Rimolo-Donadio, “A compact functional verification flow for a risc-v 32i based core,” in *2020 IEEE 3rd Conference on PhD Research in Microelectronics and Electronics in Latin America (PRIME-LA)*, 2020, pp. 1–4.

[11] R. S. S. Kumari and C. Gayathri, “Interfacing of mems motion sensor with fpga using i2c protocol,” in *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, 2017, pp. 1–5.

[12] P. Bagdalkar and L. Ali, “Interfacing of light sensor with fpga using i2c bus,” in *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2020, pp. 843–846.

[13] V. S. Katkar, D. K. Shah, and S. S. Ashtekar, “Fpga implementation of i2c based networking system for secure data transmission,” in *2019 International Conference on Advances in Computing, Communication and Control (ICAC3)*, 2019, pp. 1–5.

[14] K. B. Bharath, K. V. Kumaraswamy, and R. K. Swamy, “Design of arbitrated i2c protocol with do-254 compliance,” in *2016 International Conference on Emerging Technological Trends (ICETT)*, 2016, pp. 1–5.

[15] T. Addabbo, A. Fort, M. Mugnaini, S. Parrino, A. Pozzebon, and V. Vignoli, “Using the i2c bus to set up long range wired sensor and actuator networks in smart buildings,” in *2019 4th International Conference on Computing, Communications and Security (ICCCS)*, 2019, pp. 1–8.

[16] J. Bruce, M. Gray, and R. Follett, “Personal digital assistant (pda) based i2c bus analysis,” *IEEE Transactions on Consumer Electronics*, vol. 49, no. 4, pp. 1482–1487, 2003.

[17] J. Ďuďák, S. Pavlíková, G. Gašpar, and M. Kebísek, “Application of open source software on arm platform for data collection and processing,” in *14th International Conference Mechatronika*, 2011, pp. 76–78.

[18] T. Instruments, “Software i2c on msp430™ mcus,” 2018. [Online]. Available: https://www.ti.com/lit/pdf/slaa703

[19] Microchip, “Using a pic16c5x as a smart i2c™ peripheral,” 1997. [Online]. Available: http://ww1.microchip.com/downloads/en/AppNotes/00541e.pdf

[20] Intel, “How to implement i2c serial communication using intel mcs-51 microcontrollers,” 1993. [Online]. Available: http://ww1.microchip.com/downloads/en/AppNotes/00541e.pdf

[21] RISC-V, “Risc-v gnu toolchain,” repository. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain

[22] ——, “The risc-v instruction set manual,” specification. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf