A Low-Cost Semihosting Approach to Debug DSP Application

Tao Huang[1], Haoqi Ren[1], Zhifeng Zhang[1], Bin Tan[2], and Jun Wu[3](✉)

[1] Department of Computer Science, Tongji University, Shanghai, China
{2030798,renhaoqi,zhangzf}@tongji.edu.cn
[2] School of Electronic and Information Engineering, Jinggangshan University, Ji'an, China
tanbin@jgsu.edu.cn
[3] School of Computer Science, Fudan University, Shanghai, China
wujun@fudan.edu.cn

Abstract. Applications of digital signal processor (DSP) involve large amounts of data processing. In order to be able to improve the speed of DSP application development, it is necessary to be able to implement debugging functions that can support File I/O at a small cost of modification. This paper proposes a complete set of implementation methods of the semihosting debugging function on a DSP without an operating system, including hardware support and software algorithms. On hardware, this article adds only one self-trapping instruction to support semihosting debugging. In software, this article is based on the GDB File I/O extension protocol to design and implement the debugging agent software and C language library underlying file operations. And after optimizing library files and application source code, the I/O speed of the architecture can meet users' debugging needs. This solution can realize the semihosting debugging function for DSP chip at a low cost and has good performance. Therefore, provides more powerful debugging functions for DSP application development. In assembly level debugging, observing the execution of each assembly instruction can also verify the instruction execution of the chip itself. The addition of file operation can greatly validate the chip with larger data.

Keywords: DSP · Semihosting · GDB · Embedded debugging

1 Introduction

Nowadays, digital signal processing technology can be seen everywhere in people's daily lives, and digital signal processing algorithms often involve the real-time processing of large-scale data. So, in many devices, a separate DSP is always been used to complete the data-processing tasks, creating a huge market and application demand for DSP. In recent years, the development and application of deep learning have put forward higher demands on large-scale parallel computing [1]. Compared to general-purpose processors, DSP can process larger amounts of data simultaneously, but its control instructions are much simpler in order to reduce power consumption and silicon area. Therefore, how to quickly debug and verify it at various stages has become a difficult problem.

F. Gao et al. (Eds.): ChinaCom 2022, LNICST 500, pp. 33–43, 2023.
https://doi.org/10.1007/978-3-031-34790-0_3

The development and production of a chip often goes through multiple stages. As the number of transistors integrated on a single chip and the complexity of the on-chip system and microprocessor design continues to grow and increase, FPGA prototype verification before tape-out became an indispensable part of each design team [2]. Using FPGA and the debugging system as prototype can improve the verification speed [3, 4], but the digital signal processing application is characterized by the processing of a large amount of data, how to complete the verification of large-scale data during debugging is a challenge.

In general, large-scale data is often validated using file comparisons. In the field of debugging, you can simply divide the debugging method into two types, local debugging and embedded debugging [5, 6]. Local debugging is the most common. In this mode, the debugged program runs together with the debugger software on a PC, and the debugger debugs the program through the debugging interface provided by the operating system. The input and output of the program is managed by the operating system, and the result file can be easily obtained. Under embedded debugging [7], the debugger runs on the host PC and the program being debugged runs on the target machine. In the field of embedded debugging, in order to input and output files, the semihosting mechanism is required. Semihosting mode [8] refers to the delivery of input/output requests from application code to the host which running the debugger, using the host's file I/O system for input/output functionality. As for DSP, its control instructions are generally relatively simple, so usually it is difficult to run a fully functional and powerful operating system, let alone a file system. To debug it for file I/O, the only choice is the semihosting approach.

This article primarily describes the design of a semihosting debugging approach that requires only minimum hardware support and the associated hardware and software design and implementation. Hardware, the scheme implements a self-trapping instruction for the DSP to support semihosting debugging and connects to the host based on the JTAG interface. On the software, we base on the open-source debugger GDB to complete debugging agent software and corresponding C language libraries.

This article is organized as follows: The second part describes the relevant work in the field of debugging. The third part describes in detail the semihosting design we proposed. The fourth part is application optimization and validation. The fifth part is a summary of this article.

2 Related Work

Embedded debugging systems are divided into three categories: hardware debugging, software debugging and simulator debugging [9]. Software debugging requires the corresponding system and software support on the embedded hardware, and simulator debugging is a complete simulation of the embedded hardware on the host. The most widely used is hardware debugging, which can achieve completely realistic and reliable debugging effects, and does not require hardware support for the operating system.

There are two kinds of hardware debugging: in-circuit emulator (ICE) and on-chip debugger [10–13]. In-circuit emulator is a set of computer systems specifically designed to simulate the target CPU or MCU [14, 15]. The system generally contains an emulator motherboard, the processor embedded in the motherboard has exactly the same function

as the processor to be debugged, and in order to achieve the purpose of debugging, its hardware has been specially modified. Using ICE to debug is essentially a hardware debugging method that been implemented by using the corresponding ICE to replace the processor [16–18].

ICE has not been used on a large scale due to its excessive cost and poor scalability. On the contrary, the on-chip debugger has been widely used due to its low hardware overhead and cost. On-chip debugging is to add a hardware circuit module specifically for debugging, which is called the on-chip debugger, inside the processor at the beginning of the processor hardware design [19–21]. In embedded debugging field, most users only use some basic debugging functions, such as breakpoints, single steps, register access, memory access, and so on. Therefore, on-chip debugging has become the first choice for most users.

The classic semihosting mechanism in the field of debugging is proposed by ARM [22], who implements or simulates the half-master mechanism in both hardware debugging and simulator debugging. However, the semihosting mechanism in ARM hardware debugging requires more hardware support. It must use an emulator that supports the semihosting mechanism for debugging [23], which has higher requirements for hardware versions and cumbersome development process.

Based on the above related considerations, this paper adopts the on-chip debugging method in order to implement the semihosting function on DSP quickly and at low cost. And in order to reduce the dependence on the emulator, this article will use software to provide appropriate support for the semihosting mechanism as much as possible.

3 Semihosting Approach

SWIFT processor is a high-performance DSP chip developed by Tongji University, which uses VLIW (Very Long Instruction Word) [24] technology and self-developed instruction set, and has high parallelistic vector computing capability. Consider power consumption and silicon area, the SWIFT instruction set omits a large number of control instructions. In order to be able to more easily and quickly verify and debug applications on the SWIFT processor, we need to support semihosting debugging with minimal changes.

3.1 GDB-Based Semihosting Architecture

The SWIFT DSP has a debugging system consists of a debug module, a JTAG tap on hardware and a debug agent (proxy), GDB on the software, as shown in Fig. 1.

GDB (The GNU Project Debugger) is a powerful open-source debugger. The structure of the GDB target side is relatively separate from other parts, which reduces the difficulty and effort of porting and adapting the new hardware architecture. GDB has thus become the first choice for many new chips. GDB is controlled by IDE in the debug system, execute what the user wants. Not only is GDB capable of debugging programs on the host, its simple RSP protocol also provides the basis for remote debugging. GDB using RSP protocol to send command and data to the debug proxy and get the result from it.

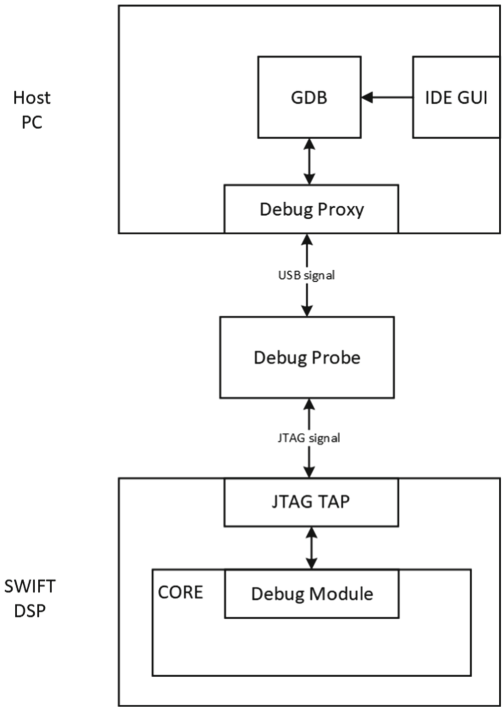

Fig. 1. The embedded debugging system Architecture for SWIFT DSP.

The debug proxy gets the GDB command and drive the Debug Probe to connect with the target machine, which is SWIFT DSP. Debug Probe in this system is simply a convertor translate USB signals to JTAG signals. JTAG TAP and Debug module control the core and using JTAG interface to communicate with the host.

The RSP protocol GDB used has an extension protocol, File-I/O Remote Protocol Extension, which can implement remote File I/O function. The File I/O Remote Protocol extension allows remote target to complete various system calls over GDB using the host's file system and console I/O. The target's system call is converted to a remote protocol packet to GDB, which then performs the desired action on the PC host system and returns the response packet to the remote target. This can simulate file system operations on targets without file system.

Based on File-I/O Remote Protocol Extension, this debug system can achieve semihosting operations with several modifications.

One semihosting operation based on GDB is performed as the procedure below:

1. Processor requires file manipulation.
2. Processor stalls and sends a request to debug module.
3. The request is sent to GDB via debug probe.
4. GDB performs the file operation.

5. GDB get the result and feed back to processor.

In traditional semihosting architectures, a debug probe is required to support semihosting mode. When the processor makes a request outward through the debug module, the probe will interact with the hardware to obtain the necessary information after receiving the request. This process is transparent to the user, which limits the user's hardware choice and prevents the user from making corresponding changes and optimizations on the software. In order to avoid hardware limitations, this paper's approach only uses the probe made of ordinary USB-JTAG communication chip. And the software does the necessary job.

3.2 Hardware Support

As the semihosting procedure shown in Sect. 3.1, DSP has to have the ability to stall autonomously and send the request to JTAG interface. To help DSP implement semihosting function with minimal changes, we added a instruction TRAP for SWIFT DSP. The way TRAP instruction works is shown in Fig. 2.

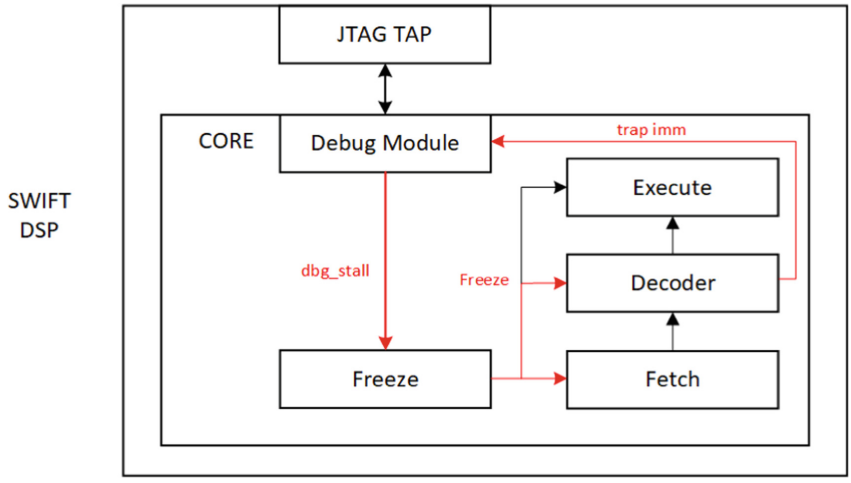

Fig. 2. How TRAP works in SWIFT DSP.

The immediate value followed by the TRAP instruction indicates which system call was made. The decoder of DSP decodes the TRAP instruction into a signal trap with an immediate value. Signal trap is passed into the debug module, causes the signal "dbg_stall", which generates several freeze signals to stall the core. The freeze signals are connected to the pipeline of SWIFT DSP. In order to stop in the right position, we only freeze the fetch module and the decoder of the processor to let the former instructions execute normally.

At the same time, the immediate value is also passed into the debug module, which is placed in the first half of the DRR (debug reason register). DRR was originally used to indicate what has happened that caused SWIFT DSP to stop. But the number of events

only used the lower 14 bits. We use the high 18 bits of DRR for the immediate value of instruction "TRAP". The debug proxy learns the status of the core by polling for the registers of the core.

3.3 Software Support

The debug proxy controls the interaction between the debug probe and GDB. The debug proxy knows that DSP has been paused and gets the immediate value of TRAP by the value of the debug reason register. But the system call requires various parameters, taking read as an example, (1) is the open function supported by GDB. (2) is the packet content GDB need and (3) is what GDB feed back to proxy after performed the system call.

$$\text{int open(const char} * \text{pathname, int flags);} \tag{1}$$

$$\text{`Fopen,pathptr/len,flags,mode'} \tag{2}$$

$$\text{`Fretcode,errno,Ctrl-C flag;call-specific attachment'} \tag{3}$$

Although instruction TRAP can indicate the system call, it is not able to provide the value of parameters. To get these parameters we have to refer to the context. In that case, the basic file operating functions in the standard C library have been written as shown in Fig. 3.

```
int myopen(const char *pathname, int length, int flags,
int *errorno)
{
  int a;
  __builtin_dsp_trap(1);
  return a;
}

int open(const char *pathname, int flags)
{
  int length = strlen(pathname) + 1;
  return myopen(pathname, length, flags, &errno);
}
```

Fig. 3. The "open" and "myopen" function in C standard library

We design the parameter format of the function "myopen" according to the parameter information required by the RSP protocol of GDB and the format of the reply. In the function "open" of the standard library, the parameter value required by protocol is first calculated, and then the important function "myopen" is called.

```
myopen:                                          # @myopen
  addi  GR30, GR30, -256
  load32  GR2, GR30, 67
  load32  GR3, GR30, 66
  store32 GR4, GR30, 63
  store32 GR5, GR30, 62
  store32 GR3, GR30, 61
  store32 GR2, GR30, 60
  trap  1
  load32  GR2, GR30, 59
  addi  GR30, GR30, 256
  ret GR31
  nop
  nop

open:                                            # @open
  ...
  call  strlen
  ...
  call  __errno
  nop
  nop
  store32 GR2, GR30, 3
  ...
  store32 GR18, GR30, 2
  call  myopen
  nop
  nop
  ...
  ret GR31
  nop
  nop
```

Fig. 4. The assembly code of “open” and “myopen” function

The assembly code generated by compiling these functions by compiler is shown in Fig. 4. The GR30 is the stack register for SWIFT. After entering the function “myopen”, the processor first modifies the value of the stack register and therefore enters the corresponding function stack area. Then compiler want the parameters are stored in a contiguous chunk of memory. So, it starts to collect them from memory or registers.

Finally, the parameters are stored in the memory address of GR30+63/62/61/60. Therefore, proxy only needs to read the value of GR30 to get the address where all parameters are stored in memory. When the processor encounters a trap instruction to stop, the proxy drives debug probe to read the memory to get the value of the parameter. Then proxy gets all the information it needed and combined the data in RSP protocol format and sent it to GDB.

GDB performs the corresponding operation on the host and returned the result to proxy. The proxy needs to give feedback to the processor by writing the returned data to memory. According to the assembly of the function “myopen”, the address value of the global variable “errno” is stored at address of “GR30+60”. When the proxy

encounter a GDB execution failure, the reason for the failure can be stored to the corresponding address. After executing the TRAP instruction, the processor loads the data at "GR30+59" into GR2 and returns it to the upper function as the end. Therefore, the return value "fd" or failure code "−1" after GDB executing the function "open" can be placed at address of "GR30+59". In this way, the SWIFT processor finishes executing the function "open". Similarly, we can implement the common file I/O functions supported by GDB, such as close, read, write, etc.

This method relies on how the compiler compiles functions such as "myopen", and more specifically on the size of the function stack. However, functions such as "myread" and "mywrite" are used in the standard library, and the standard library is provided to the users in the form of a static library, and the assembly in the static library is fixed and will not be modified by the user. The compiler does not frequently modify the size of the function stack. Also, the position of these parameters could be listed in the configure file of the toolchain. It allows us to modify the stack without recompile the proxy.

4 Validation and Optimization

To validate the semihosting system, we use a development board containing two Xilinx VU440 FPGA chips to build the prototype verification system. The structural diagram of the development board is shown in the Fig. 5. SWIFT DSP has ultra-long vector parallel processing capability that cannot be implemented on a single VU440 chip, so we use time-division multiplexing to logically split it and implement it on two FPGAs. Due to the board level delay, the consequent main frequency can only be around 20 MHz. And the JTAG clock must lower than the main frequency of DSP. We use the FT2232HL development board as the debug probe, the structure diagram is shown in the Fig. 6. The board uses the FT2232H chip, which supports a USB2.0 high-speed interface that converts USB signals to a variety of serial signals, including JTAG. The manufacturer FTDI also provides the basic driver library file for the chip. Facilitates the writing of debugging proxy.

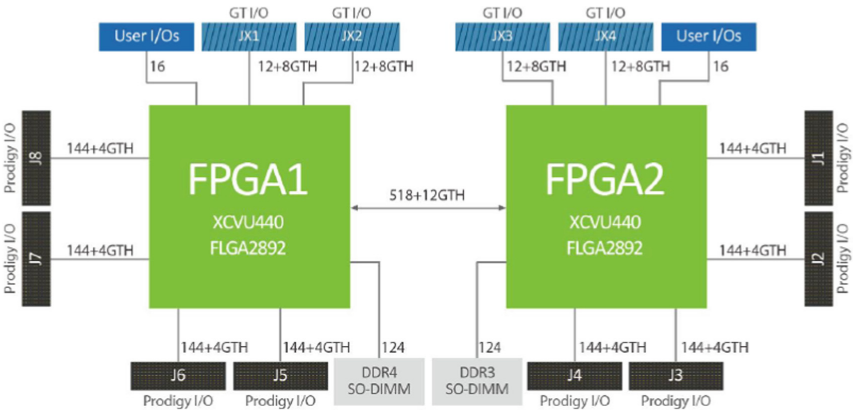

Fig. 5. The architecture of FPGA development board

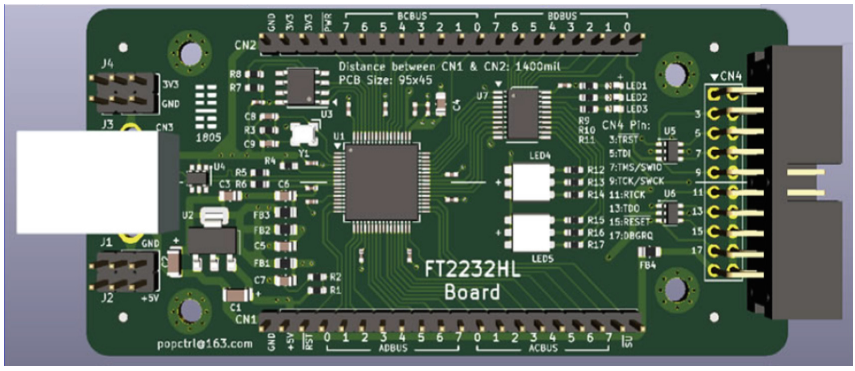

Fig. 6. The appearance of the FT2232HL development board

We tested this prototype verification system with a commonly used digital signal processing program that includes FTT algorithm. Without optimization, the running time is unbearable, taking 14 min to complete according to Table 1. This is because the bottleneck of program speed is file operation.

Table 1. The running time of a digital signal processing program in 20 MHz FPGA

Before optimization	After optimization
14 min 20 s	9 s

In SWIFT's C language library, the underlying layer of commonly used input and output functions is implemented with read and write functions, such as printf, fread, fwrite, etc. In DSP applications, programmers are used to call input and output functions frequently, and the amount of data for I/O is very large, as the amount of data in a single I/O operation is very small. Each time SWIFT executes an I/O operation, it has to go through the transmission and conversion process from the processor's debug module to the debug probe, proxy, and GDB. Its rate is much lower than that of local I/O. Too many I/O operations can greatly slow down the speed of the SWIFT processor, and caused the result of 14 min in Table 1.

To improve its speed, we optimize it in our program. In the C language standard library, the FILE struct is used to flag a file, and we add a data member "char buf[MAXBUF]" to the FILE struct to cache data read in from the remote host. When there is no data in the buffer, calling the input function will directly read the MAXBUF size data from the remote host. And when there is data in the buffer, the program reads the data from the buffer and returns.

The optimization of the output function follows a similar principle, reducing the number of I/O and increasing the amount of data for a single IO. For example, the implementation of the function "printf" can first call "sprinf" to store data in contiguous memory and then output it to the remote host.

As seen in Table 1, after optimization, the running time is reduced to 9 s. The file operation is no longer the bottleneck as the main frequency is only 20 MHz in FPGA. The running speed after optimization can meet users' debugging need.

5 Conclusion

In this article, we proposed a complete and simple semihosting debugging approach based on the SWIFT DSP. The solution does not rely on too much hardware, only requires the processor to support self-trapping instructions, and can use the most versatile USB to JTAG serial port chip as a debug probe, reducing the cost of developing and using. On the software side, we used libraries and proxy to read the relevant parameters of I/O functions and write back the results, which is convenient for modification and debugging. Libraries and applications can be I/O optimized to get close to the speed of traditional semihosting methods.

We used FPGA to build a prototype system, FT2232HL board to connect FPGA and PC host. After code optimization, the running speed of program met most user's need in debugging. Given the speed of the DSP clock and JTAG clock, it could be called a satisfactory performance.

The approach is based on an open-source debugger, which greatly reduces the development workload of the semihosting function and can quickly provide the available semihosting function to various processors. Providing more powerful functions for application debugging can accelerate the speed of application development and adaptation, and provide great help for a new DSP chip to seize the market.

Acknowledgment. The authors thank the editors and the anonymous reviewers for their invaluable comments to help to improve the quality of this paper. This work was supported by the Key-Area Research and Development Program of Guangdong Province under Grant 2018B010115002, National Natural Science Foundation of China under Grants 61831018 and U21A20452, the Outstanding youth project of Natural Science Foundation of Jiangxi Province 20212ACB212001, and the S&T plan projects of Jiangxi Province Education Department GJJ201003.

References

1. Kuschnero, V.M., Schaedler, M., Bluemm, C., et al.: Advances in deep learning for digital signal processing in coherent optical modems. In: Optical Fiber Communication Conference (2020)
2. Austin, T., Larson, E., et al.: SimpleScalar: an infrastructure for computer system modeling. Computer **35**, 59–67 (2002)
3. Nadal, J., Baghdadi, A.: Parallel and flexible 5G LDPC decoder architecture targeting FPGA. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **29**(6), 1141–1151 (2021). https://doi.org/10.1109/TVLSI.2021.3072866
4. Weng, O., Khodamoradi, A., Kastner, R.: Hardware-efficient residual networks for FPGAs (2021)
5. Berger, A.S.: An overview of the tools for embedded design and debug (2020)
6. Berger, A.S.: Best practices for debugging embedded software (2020)

7. Hossain, F., Iry, J., Kulkami, N., et al.: Method and system for remote debug protocol proxying for production debugging; selective session and user routing for debugging in multi-tenant cloud computing infrastructure. US (2014)
8. Prado, B., Dantas, D., Bispo, K., et al.: A virtual prototype semihosting approach for early simulation of cyber-physical systems. In: 2018 IEEE Symposium on Computers and Communications (ISCC). IEEE (2018)
9. Chance, G., Ghobrial, A., Mcareavey, K., et al.: On determinism of game engines used for simulation-based autonomous vehicle verification (2021)
10. Lee, H., Hyunggoy, Oh., Kang, S.: On-chip error detection reusing built-in self-repair for silicon debug. IEEE Access **9**, 56443–56456 (2021). https://doi.org/10.1109/ACCESS.2021.3071517
11. Mitra, S., Barrett, C., Lin, D., et al.: Post-silicon validation and debug using symbolic quick error detection (2018)
12. Cao, Y., Hao, Z., Palombo, H., et al.: A post-silicon trace analysis approach for system-on-chip protocol debug. In: 2017 IEEE 35th International Conference on Computer Design (ICCD). IEEE (2017)
13. Merten, M., Huhn, S., Drechsler, R.: A codeword-based compactor for on-chip generated debug data using two-stage artificial neural networks. In: 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp. 1–6. IEEE (2021)
14. Petrović, P.B.: A new electronically controlled floating/grounded meminductor emulator based on single MO-VDTA. Analog Integr. Circuits Signal Process. **110**, 185–195 (2021)
15. Minati, L., Mancinelli, M., Frasca, M., et al.: An analog electronic emulator of non-linear dynamics in optical microring resonators. Chaos Solitons Fractals **153**, 111410 (2021)
16. Kumar, V., Dubey, S.K., Islam, A.: A Current-Mode Memristor Emulator Circuit (2020)
17. Barboni, L.: A passive circuit-emulator for a current-controlled memristor (2020)
18. Zhu, M., Wang, C., Deng, Q., et al.: Locally active memristor with three coexisting pinched hysteresis loops and its emulator circuit. Int. J. Bifurcation Chaos (2020)
19. Berger, A., Barr, M., et al.: On-chip debug. Embed. Syst. Program. (2003)
20. Berger, A.S.: On-chip debugging resources (2020)
21. Backer, J., Hely, D., Karri, R.: Secure design-for-debug for systems-on-chip. In: Test Conference. IEEE (2015)
22. Yiu, J.: Input and output software examples - ScienceDirect. The Definitive Guide to ARM® CORTEX®-M3 and CORTEX®-M4 Processors (Third Edition), pp. 583–604 (2014)
23. Yiu, J.: Getting Started with the ARM RealView Development Suite. Elsevier Inc. (2011)
24. Uzan, D., Kahn, R., Weiss, S.: Perceptron based filtering of futile prefetches in embedded VLIW DSPs. J. Syst. Architect. **110**(3), 101826 (2020)