Műszaki Tudományos Közlemények vol. 21. (2024) 97–106.
DOI: Hungarian: https://doi.org/10.33895/mtk-2024.21.15
English: https://doi.org/10.33894/mtk-2024.21.15

EXAMINATION OF EMBEDDED SYSTEM ADAPTIVE CONFIGURABILITY

József VÁSÁRHELYI,[1] Roland BATÓK,[2] Dániel DRÓTOS[3]

[1] *University of Miskolc, Faculty of Mechanical Engineering and Information Technology, Institute of Automation and Infocommunication. Miskolc, Hungary, jozsef.vasarhelyi@uni-miskolc.hu*

[2] *University of Miskolc, Faculty of Mechanical Engineering and Information Technology, Institute of Automation and Infocommunication. Miskolc, Hungary. roland.bartok@uni-miskolc.hu,*

[3] *University of Miskolc, Faculty of Mechanical Engineering and Information Technology, Institute of Automation and Infocommunication. Miskolc, Hungary. daniel.drotos@uni-miskolc.hu*

Abstract

Field Programmable Gate Arrays or FPGA circuits have been the focus of applied research since their appearance. Following the introduction of these circuits in 1980, their year-by-year development and the expansion of application areas have opened up new areas of research. With the appearance of system-on-chip, SOC (System on Chip) solutions and adaptive processing units, they occupy an important role in industrial applications. The question is: when, why and what hardware is to change dynamically during system operation? Whether it is necessary to expand computing resources by expanding new circuit blocks or by inserting soft-core processor(s), this can be determined by examining the computing load of the embedded system during operation. This article presents the tests that were performed on the processors of the embedded systems, in order to determine the computational load of the processor and how the interrupt handling can be accelerated. The article also presents the examination of fuzzy interpolation.

Keywords*: embedded systems, dynamic function exchange, FPGA, robot control, fuzzy interpolation.*

1. Introduction

Field Programmable Gate Arrays or FPGA circuits have been in the focus of applied research since their appearance. The areas of application, which at that time were still limited to the implementation of two-level logical networks, are now suitable in the areas of telecommunications, real-time embedded systems, artificial intelligence, image processing and adaptive hardware using dynamic function exchange.

One of the important parameters of embedded systems is the dissipated power and computation speed rate. It is possible to reduce or increase these design parameters by several methods. For example, reducing the frequency of the microcontroller will reduce the dissipated power, but this also reduces the computation speed. The best solution is when the processor load is analysed while executing the application task and then by making the necessary modifications based on the results. One possible solution is a new approach to the interruption mechanism or the examination and hardware implementation of the control algorithm. Both solutions raise the following questions: Is it necessary to expand computation resources by adding new hardware blocks or by inserting new soft-core processor(s)?

This article is the extended version of the FMTŰ 2024 conference plenary presentation. It presents an examination of the computational load of the processor used in embedded systems, and then by presenting two examples -- interrupt management and fuzzy interpolation with state machine implementation. These two examples illustrate the possible application of adaptive operation and increasing the computation speed.

2. Literature review

2.1. Processor load analyses

The computer model formulated by John von Neumann in 1945, in his report "The First Draft of the Report on the Edvac" **[1]**, has been reviewed

by many articles over the years, presenting all its advantages and disadvantages.

The biggest problem was caused by the fact that the processor manufacturers left out the extra status bit used in the data storage by Neumann, which would have had the task of indicating the status of the stored data (program code or data) of the stored data.

The Neumann principles are as follows [2]:
- fully electronic computer,
- use of binary number system,
- existence of arithmetic unit (universal Turing machine),
- existence of central control unit,
- existence of internal program and data storage i.e. memory [2].

The operation of the Neumann model is most effective when a processor performs a single task [3]. This approach is generally true for embedded systems. In an embedded system, the processor has a well-defined task, especially when not using an operating system for controlling its operation. Problems with the model occur when the number of tasks differs from the number of processors present in the machine, i.e. the number of tasks is higher than the number of processors. Unfortunately, software developers still adhere to the classic Neumann model, so the operating system hides the hardware layer from the user, so it appears as if the processor is available for all the tasks [4].

Since the use of FPGA technology, we are concerned with two types of processor: soft-core and hard-core. The former is described in a hardware description language (VHDL or Verilog) and inserted into the FPGA design, while the latter is integrated on the same die with the programmable logic.

For embedded systems, which are implemented using low-resource FPGAs, it is especially important to examine the processor or microcontroller load during the execution of the user software.

In FPGAs, one can insert a soft-core processor using different methods. Either using Intellectual property modules (IP), which is one of the most common methods, or using the register transfer level (RTL) method. The use of the RTL soft-core processor has the advantage that all the internal signals of the central unit (CPU) are accessible, and the processor load test can be carried out. In contrast, in the case of IP-based design, the internal signals of the processor are not available, so the processor load test can not examine the internal signals.

In the case of FPGA-based systems, it is possible to increase the computing power of the processor by using other auxiliary accelerators and co-processors. Inserting these kinds of circuits during operation is possible with the dynamic function exchange. In this way the processor computation load can be reduced.

Analysing the computation load during operation of the processor is based on the examination of the system signals. Hardware errors can be determined by monitoring all signals [5]. The method tests the hardware operations and draws conclusions regarding machine malfunctions.

Another test method was used in the design phase, to determine the necessary hardware resources required for the optimal operation of the software in an embedded system. [6] [7]

Zhao et all. used a software tool to determine the errors in the running program. A target system was designed for this method [4].

During system design and load testing, the following aspects must be considered [8]:
- the measurement method must be independent of the software, which is executed by the embedded processor;
- the implementation of the measurement method must not modify the hardware architecture of the microprocessor;
- the measurement must provide real-time data, i.e. must detect changes in the system (processor) instantaneously.

Based on the above criteria, one can conclude that the measurement of the processor's computational load must be independent of the embedded system hardware and should not influence in any sense the system under test.

2.2. The execution of the interrupt process

As mentioned, the von Neumann computer model works well when the microprocessor performs only one task. Unfortunately, in embedded systems, the processor must perform several tasks at the same time. This is possible only if the processor divides its time between the tasks. This is only possible either using interrupt management or using an operating system, which allows task scheduling [9].

Interrupt Servicing (executed by Interrupt Service Routine -- ISR) appears as an additional task next to the main program. The processor must perform this task also [10].

The processor cannot immediately execute the ISR, since there is a latency in the interrupt acceptance, which elapses between the interrupt

request and the start of the interrupt service. There are several reasons for this dela:

- the processor executes critical code segment, which means the interrupt request is disabled;
- the processor executes a task, which has higher priority;
- the processor executes a long instruction that cannot be interrupted.

The time until the interrupt accept may also vary. One of the reasons for this is that the instruction pipeline must be emptied. Its duration depends on the type of instructions in the pipeline. The other reason is that after the ISR (interrupt handling program) start the interrupted program state (main or other ISR) environment (registers, return address, program status flags, etc) must be saved and at the end of the ISR service routine this information must be restored. The time needed for saving the interrupted program environment depends on how many registers the ISR uses. The time delay until the interrupt accept therefore is not constant and can fluctuates.

When a processor receives an interrupt signal, and if the interrupt is enabled, the processor suspends the running process and begins handling the exception. After the execution of the interrupt handling routine (ISR), the processor returns to the suspended task.

In the case of ARM processors, the ISR should be treated as an exception as described in the ARM documentations **[11]**. This process of handling the exception is briefly explained below:

- If an ISR event occurs and exception handling is enabled, the processor suspends the currently executed instruction (and/or discards the so-called long instructions and restarts their execution after returning from ISR),
- Then the context registers (8 registers each 32 bits) are saved in the stack memory specified by the stack pointer (for ARM MSP - Main Stack Pointer or PSP - Process Stack Pointer). The saved registers are as follows: xPSR (Program Status Register), PC (Program Counter - contains the return address), LR (Link Register R14) and registers R12, R3, R2, R1, R0 are also saved.
- After that, the processor switches to handler operation mode.
- Then the program counter (PC) is loaded with the ISR start address and the LR loads the return address.
- In the next step, the ISR number is loaded into the IPSR (Interrupt Program Status Register).
- Finally, the execution of the interrupt handling routine begins.

The process described above consumes processor time. It is important to consider the time elapsed between the ISR request and the start of exception handling. If the interrupt is repeated cyclically, the response time of the ISR increases with the latency of the start of the interrupt handling. Critical external events can delay the start of the interrupt service routine from the point of view of the service time of the interrupt handling.

How much time is required to save registers ("overhead") and to execute program PUSH/POP instructions in order to save other registers? In such cases, the processor deals with the save/restore address of the register (stack operations). If the address contained by stack registers point to external memory, this "overhead time" will be longer (executing external memory operations).

The maximum repetition frequency of interrupts can be calculated for a given processor considering the given clock signal frequency:

$$F_{MaxInt} = F_{CPU} / (C_{ISR} + C_{Overhead}), \quad (1)$$

where F_{MaxInt} is the maximum repetition frequency of the interrupt – i.e. ISR, F_{CPU} is the frequency of the system clock, C_{ISR} is the number of clock cycles required to execute the interrupt handling routine (ISR), and $C_{Overhead}$ is the number of clock cycles used to save and restore the registers (i.e. to save the program state and restore).

By fulfilling a frequent interrupt request, the processor executes the ISR. Therefore, it spends less time on executing the main program. The time required to complete the ISR (U_i) is:

$$U_i = (F_i (C_{ISR} + C_{Overhead})) / F_{CPU}, \quad (2)$$

where F_i is the frequency of interrupts.

From the above, it results that the main program is apparently executed with a much lower clock frequency:

$$F_{main} = (1 - U_i) * F_{CPU}, \quad (3)$$

where F_{main} is the apparent frequency of main program execution.

3. Processor computation load measurement

During processor operation one can distinguish two states: when the processor runs the programs i.e. executes instructions -- this is called the loaded state; while states are possible when the pro-

cessor is not active -- this is called an unloaded state. In the unloaded state, we distinguish:

- the sleep mode -- the processor does not execute instructions and it appears as if it is switched off - the dissipated power of the processor is very small;
- the idle state -- the processor waits for some event while executing instructions.

In the active state of the processor -- depending on the software activity -- different parts of the system are more active than other parts of the processor, for example, the arithmetic unit -- when it performs mathematical computations, shows high activity, or the memory manager unit also shows high activity when performing I/O operations. Based on this, the following processor operational categories are distinguished:

- base (unloaded state),
- calculation intensive,
- input - output operations.

This kind of classification was achieved by measuring the actual load of different parts of the processor. Using the measurements, one can identify the intensively used processor parts.

3.1. Processor load measurement method

The measurement method is supported by the internal structure of the used processor (**Figure 1**). Generally speaking, the processor is a state machine in which the storage elements (in the figure: Registers) store the current state of the system. The current state (q) changes at the rate of the clock signal. The value of the new state is generated by the so-called excitation network (in the figure FN1, FN2, etc.) as a function of the current state and possible external signals (v). The values of v, G, q are functions of time t (in this case, the clock signal controls the state changes). The stored values become the excitation value after a clock delay **[8]**, therefore:

$$q(t) = G(t-1) \tag{4}$$

External signals influence the values of the G function, so it is not necessary to test them directly. The G values are transferred to the q (next state) values after a clock delay, so the measurement can be performed either by examining the G or q signals.

The operation of individual parts of the CPU is indicated by the change in the value of their associated memory. So, the measurable characteristic of the q signals relevant to the investigation is the change in frequency of the q signals, that is, the frequency of the signal. However, the classification is not based on the instantaneous values of the signal frequencies, but on the time change of the frequency, the time function of the frequency: $fr(t)$.

3.2. Presentation of the System under test

The application on which the processor load measurement method was performed is an embedded system implemented on FPGA that controls a device that follows the movement of the Sun.

The application is a solar panel system that always sets the optimal operation depending on the movement of the Sun, i.e. it shows the maximum surface towards the Sun, which is perpendicular to the sun's rays (**Figure 2**).

The control system calculates the momentary orientation of the sun's rays (horizontal and north angle) as a function of time and geographical coordinates. Based on the calculated coordinates, it adjusts the solar panel so that it is perpendicular to the sun's rays, i.e. it moves the panel by con-

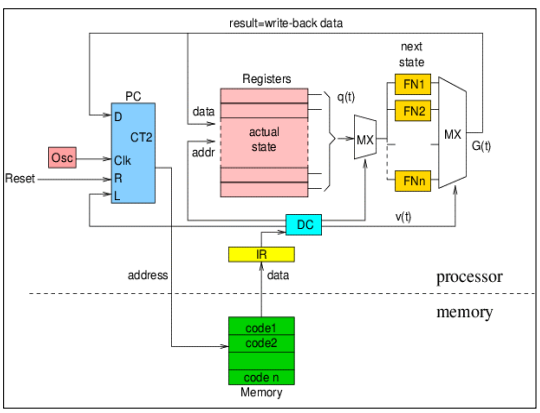


Fig. 1. *Block diagram representing processor structure.*

Fig. 2. *The equipment controlled by the test application*

trolling the activator motors. During the movement (adjustment of panels) depending on the computed result, the software performs a lot of I/O operations, similarly when the program updates the display of the user interface.

3.3. Measurement results

The measurement results are presented based on article [8] .

The system was modelled with an embedded system implemented on FPGA, using RTL-based ARM M0+ processor.

During the tests, the system model was run using a hardware simulator, which saves the signal values appearing at the outputs and inputs of the system elements in a so-called "dump" file. The “dump” file containing the output results was analysed with an appropriate software.

The frequency signal fr(t) variation in time is produced by a dedicated software that processes the output file of the simulation results. The algorithm built into the processing software corresponds to the operation of the digital circuit used for frequency measurement. During the hardware simulation, a large amount of data was generated. So a filter algorithm was incorporated into the processing software, which removed the data that was not relevant to the test. Due to the way the model was described, a lot of redundant information was obtained by having several signals duplicated by some other signals, so their values were identical. By removing this redundant information, the amount of data to be processed can be reduced.

With the graphic representation of the results, one can deduce which signal frequency changes show significant difference for some signals, at the time when the application software changes the processor state (load), i.e. when the computation algorithm starts or stops. Figure 3 shows the variation of some measured signals at the moment of starting the computation part of the application program.

Examining the signal correlation change shown in Figure 3, for example, in the case of the *alu_ext_trans_o* signal, one can notice that this signal shows a significant amplitude change, which means that the processor starts the computation, so its load increased.

Based on the modelling and simulation results, one can select the signals suitable for computation load measurement of the central unit. For the model, it is known when the application software performs a computation that will result in a load on the CPU. By examining the statistical characteristics of the fr(t) functions, one can select which signals shows a higher degree of correlation with the processor behaviours to be detected. Figure 4 shows the distribution of the correlation of the examined processor signals with the computation mode variation. One can conclude that some signals correlate well with the operating mode, so they can be suitable for indicating computation load periods.

From the number of signals for the measurement, the ones whose change can be detected most easily by the measurement module should be selected.

After modifying the structure of the embedded system, the selected signals of the processor were connected to the measurement module. These signals were analysed using the measuring system. After measuring the frequency variation and amplitude change of the signals using the monitoring module one can measure the system evolution.

The computation load of the processor can be deduced from the amplitude changes of the selected signals.

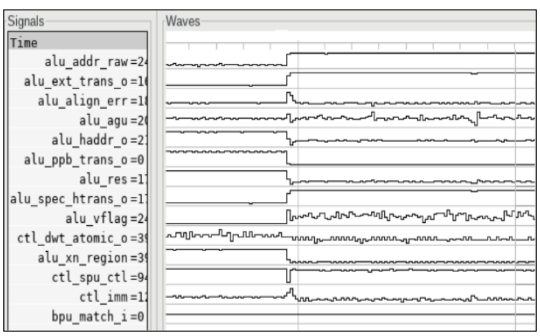


Fig. 3. *Graphical representation of several measured signals.*

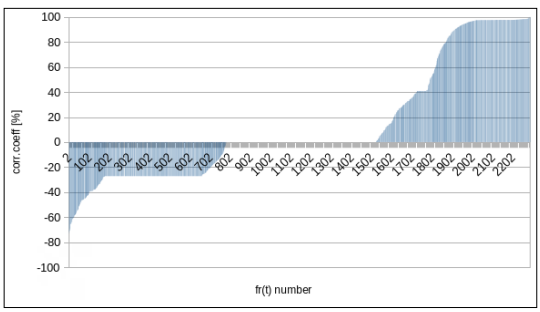


Fig. 4. *Correlation values of signals to CPU load.*

4. New method for interrupt process execution speed-up

The block diagram of the experimental system is presented in Figure 5. In the experiment, for implementation, ARM RTL soft-core (ARM M0 processors) were used.

The experiment is as follows: a timer generated an interrupt signal at a given frequency (10 KHz). The exception handling program counted the number of interrupts, i.e. increases in the value of a counter (software clock). The main program displayed the value of the counter on a display.

The P3 processor works in a traditional way. It executes the main program and handles the interrupt, i.e. executes a timed task, and then sends the value of the counter to the display. The execution time of the interrupt and exception acceptance latency of the traditional solution running on the P3 processor are compared with a system executing the task in parallel (P1 and P2). Processor P1 runs the main program, while processor P2 performs interrupt handling. When the exception handling is done, the P2 processor goes in to a suspended state (sleep mode).

When an interrupt signal occurs, the P2 processor wakes up and executes the exception handling program. The operation of this processor has been designed in such a way as to avoid the interrupt accept latency that arises during the execution of the usual interrupt handling process. So, in this way, there is no time variation. The time required to start the service is constant.

Therefore, the timer peripheral signal was not connected to the interrupt request input of the ISR signal of P2. The processor is in suspended state (sleep mode). The peripheral signal generated by the timer is connected to the wake-up input of the processor. When the signal starts the processor, ISR is served, which this time is listed as the main program of P2. Since the wake-up occurs on the rising edge of the ISR, servicing of the "ISR" starts immediately. There is no need for an interrupt acceptance process. There is no need to save the state of the main program because the ISR is the main program of the processor P2. Therefore $C_{Overhead} = 0$ 0 in equations (1) and (2).

At the end of the "ISR" service, the program jumps back to the beginning of the program, where an instruction suspends the processor P2. In sleep mode, it then waits for the next ISR signal The peripheral interrupt signal is stored by the processor interrupt handling circuit until the service starts.

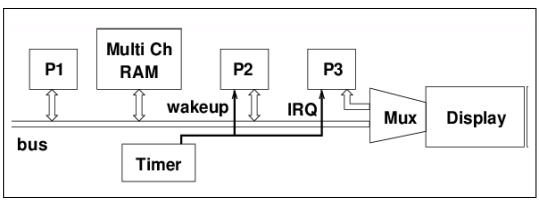
Figure 6 shows the result of the simulation of the parallel solution in comparison with the conventional solution. The event signal indicates the interrupt request of the peripheral; this pulse comes from the timer peripheral with a frequency of 10 kHz.

The *Wake Up* signal wakes up the P2 processor. The No sleep signal is generated by the processor and indicates that it continues program execution. This signal deletes the external storage. The *Service Start* signal is generated by the first "ISR service" instruction of the P2 processor, indicating that the service has started.

The *Event* signal interrupts the P3 processor program in the traditional way. The first instruction of the ISR function of the P3 processor generates the *ISR_Start* signal. This indicator signal was used to compare the starting point of the two ISRs, one executed on P2 and one executed on P3. Due to the exception handling process (saving registers, as mentioned previously, etc.), the P3 processor starts the ISR process later than the P2 processor.

5. Fuzzy Interpolation

A robot moving in a real environment must make decisions in different situations and react appropriately to each situation. In the case of behaviour-based control, the behaviours specified by the expert system are valid for given situations. The robot recognizes situations based on the data provided by its sensors and its internal states. Each defining condition has a behaviour that the robot performs [12], [13] and [14].



5. ábra. *Megszakításkezelés párhuzamosításának tömbvázlata*

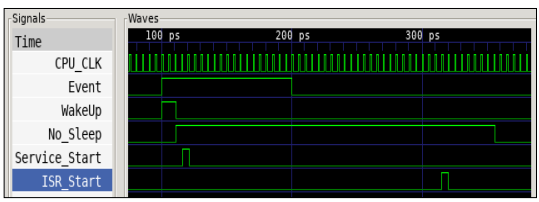


Fig. 6. *Simulation results of the interrupt handling.*

In the case where behaviour models are implemented with fuzzy logic, thanks to the fuzzy state machine, all possible behaviours are involved in the control system of the robot. Those behaviours that best fit the given set of observations will have the greatest impact on the robot's movements and its reactions [12–14].

Since robot movements are performed in a real environment, this can have different kinds of effects on the control decisions, it may happen that a set of observations does not relate to the robot's behaviour. In this case, the robot control system may become unable to take a decision. This can be improved by adding more fuzzy rules. But the increasing number of rules will result in the increased use of computing resources. Fuzzy rule interpolation provides a solution for this case [12–14].

Thanks to the interpolation procedure, it is enough to give to the expert system the most decisive rules, and in intermediate cases the interpolation procedure will provide the output control values [12–14].

Using the expert knowledge base, the system behaves almost correctly even when there is no rule of behaviour for the given situation. The fuzzy state machine uses the FIVE (Fuzzy Interpolation in Vague Environment) fuzzy interpolation procedure [12–14].

One application example of behaviour-based control is etorobotics , where the robot imitates the behaviour of an animal.

5.1. Fuzzy Interpolation in Vague Environment

The control system decision-making process of autonomous robots is carried out based on information from multiple sensors and the internal states. These input data form the observations of the fuzzy rules, and are also known as antecedents, while the outputs of the fuzzy rules are called *consequents*. A fuzzy rule can be written in the form of an implication of the form "IF ... THEN ... (ELSE)...". The set of fuzzy rules affecting the same output is called a *fuzzy rule base* [13][15] and [16].

In classic fuzzy systems, for the sake of stability, the rules must be designed in such a way that there exists a rule for every possible input combination. This ensures that the rules completely cover the state space. However, as the size of the task to be solved increases, the number of rules can grow exponentially, which results in a significant computational demand. Furthermore, generating a large number of rules is often problematic, as expert systems cannot always fully cover the problem due to mathematical complexity or lack of information. As a result, instead of a fully covering rule system, the control often works with a sparse rule base, in which some observations may not have a rule, so the output of the system will be uncertain [13], [15] and [16].

Sparse rule sets are more common than full coverage rule sets. Fuzzy interpolation methods offer a solution to the problem of sparse rule systems. These methods calculate the decisions belonging to the missing rules based on the existing rules. As a result of this method, it is sufficient for the expert system to enter only the most important rules. Thus, reducing the number of rules and the complexity of the system, as well as the interpolation depending on the method, will result also in the reduction of the computation requirement need [13], [15] and [16].

The Fuzzy Interpolation in a Vague Environment (FIVE) method places the rules in a vague space, where the Euclidean distance between the rules can be interpreted. They can be distinguished from each other based on the distance between the rules: a larger distance means a greater difference, while a smaller distance means more similar rules [13], [15] and [16].

Using the consequents weighted by the rule distance, Shepard interpolation calculates using inverse distances to create the decision of the rule base. The calculation steps of the FIVE methods are described below (see Figure 7):

- *Observation* is the information from the environment or internal state.
- With the help of *Linear Interpolation*, it assigns to the observation the value that specifies the extent to which the observation's value (μ) is a member of the given fuzzy set.
- The difference between the antecedent belonging to the rule and the μ value of the observation give the *Antecedent Distance*.
- The *Rule Distance* is the Euclidean distance of the antecedent distances.
- The *Shepard Interpolation* uses the distances of the rule to produce the Decision, i.e. is the consequent.

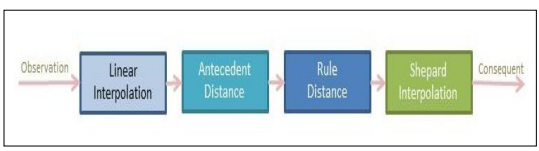


Fig. 7. *Computation steps of FIVE accelerator.*

5.2. FIVE hardware implementation

The most important requirement for the hardware implementation is tunability and parameterization during on-line operation. This means that one can modify the parameters of the FIVE controllers, including the antecedent and consequent values, while the system is operating.

To this end, parameters are stored in registers, for which an interface has been designed to allow the internal data of each module to be changed during a clock cycle.

The controller can be easily connected and adapted via variable bit width interface (output and input buses). This bus system also enables compatibility with the AXI interface, which is particularly useful for integrating FIVE hardware into a processor system.

The scalable resolution was realized with a modular structure and parameterizable bit width. The number of individual modules can be changed according to the amount required by the rules, so it flexibly adapts to the requirements of the system.

An important aspect is the portability of the code between FPGA platforms, so no manufacturer-specific elements were used in the code written in Verilog. This ensures that the code can be used on different platforms.

The circuit can be generated from FBDL (Fuzzy Behaviour Description Language) [17]. The connection and parameters of the modules can be obtained from the rule bases described in the FBDL language, which simplifies the planning and implementation of the system.

The FIVE method can produce results through four computation steps, which consist of two interpolation calculations and two distance calculations.

The processing of the data stream is illustrated in Figure 7 which contains the calculation steps of a rule base. The figure shows that, except for the Shepard interpolation module, the number of individual modules depends on the number of rules, antecedents and observations found in the rule base.

The number of linear interpolations is determined by the number of observations, while the number of antecedent distance calculations depends on the number of antecedents of each rule. Each rule has a rule distance calculation module, and the Shepard interpolation is calculated with a single module per rule base.

The calculation of linear interpolation is called "Universe" in the final implementation. In addition, also the hardware implementation was prepared with the Vivado High Level Synthesis tool, using unmodified and optimized versions of the basic μFRI library [18].

The FIVE FPGA implementation forms a synchronous computing chain. Each unit of the circuit produces the output value in response to changes in the input data. The implemented hardware works with variable bit width, by default the implementation was done with 8-bit unsigned integers for the input/output interfaces. The amount of data required for the computation can also be specified as a parameter.

The individual computation units and the inclusion unit also have constant and variable parts. The parts whose number varies based on the properties of the rules and rule bases (number of rules, number of observations, number of antecedents, etc.) were highlighted in the code and entered the Verilog code in a form that can be produced from the FBDL. The computing units ensures the pipeline operation for division and other operations. Each unit contains an internal input/output storage register, where the incoming data and the result are stored. According to the current design, all units produce results within one clock period, except for the Universe unit, which produces results in two steps.

5.3. Results

The implementation and simulation results of the system were produced with AMD Vivado version 2018.1.

The latency of the entire computation chain is 4 clock periods, as shown in Figure 8. The system can work with unsigned integers, and its maximum operating frequency is 4 MHz.

Figure 9 illustrates the operational linearity error of the FIVE pipeline accelerator. Based on the simulation, the largest absolute deviation from the expected value is 6, which corresponds to 2.34 % in the 8-bit range.

For the FIVE method implementation on FPGA, a frame made in Verilog hardware description language has been prepared, which is suitable for speeding up the computation of the FIVE meth-

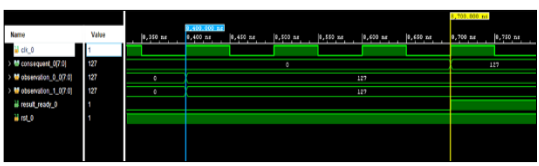


Fig. 8. *FIVE accelerator delay.*

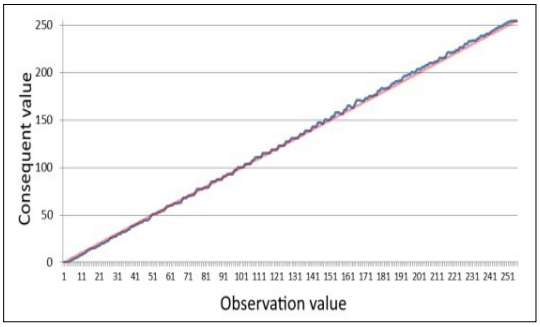


Fig. 9. *FIVE accelerator error.*

od as a hardware element that can be generated based on FBDL (Fuzzy Behavior Description Language). This module can be used independently or in association with an embedded processor system as an acceleration co-processing element. Complex behaviour of FIVE can be implemented by connecting several FRI_FIVE modules.

The FIVE IP structure also enables its implementation on Xilinx Adaptive Platform as a hardware acceleration unit operating in a software environment. In this case, the computing units are either connected to the processor system via the entire FIVE IP AXI interface. In this case the software component of the controller implementation module manages data collection and control tasks. A detailed description of the system is contained in article **[19]**.

References

[1] J. Neumann:, *First Draft of a Report on the EDVAC*. Moore School of Electrical Engineering, University of Pennsylvania, 1945.

[2] W Aspray: *John von Neumann and the Origins of Modern Computing*. MIT Press, Cambridge, 1990, p. 34-48.

[3] Drótos D., Vásárhelyi J.: *Interrupt Driven Parallel Processing*. In: 20th International Carpathian Control Conference (ICCC), Krakow-Wieliczka, Poland, 2019, 1–4. https://doi.org/10.1109/CarpathianCC.2019.8765909

[4] G. Zhao, S. Hassan, Y. Zou, D. Truong, T. Corbin: *Predicting Performance Anomalies in Software Systems at Run-Time*. ACM Transactions. Software. Engineering. Methodology. 30/3. Article 33 (July 2021). https://doi.org/10.1145/3440757

[5] H. Sayadi, N. Patel, S. Manoj P.D., A. Sasan, S. Rafatirad, H. Homayoun: Ensemble Learning for Effective Run-Time Hardware-Based Mal Ware Detection: A Comprehensive Analysis and Clas-sification, 55th ACM/ESDA/IEEE Design Automation Conference (DAC), 2018, pp. 1-6, https://doi.org/10.1109/DAC.2018.8465828

[6] J. Muskens, M. Chaudron: *Prediction of Run-Time Resource Consumption in Multi-Task Component-Based Software Systems*. In: Crnkovic I., Stafford J.A., Schmidt H.W., Wallnau K. (eds) *Component-Based Software Engineering*. CBSE 2004. Lecture Notes in Computer Science, vol. 3054. Springer, Berlin, Heidelberg 2004. https://doi.org/10.1007/978-3-540-24774-6 16

[7] B. H. Fletcher: *FPGA Embedded Processors*. In: Embedded Training Program Embedded Systems Conference San Francisco 2005 ETP-367, 2005, 37.

[8] D. Drótos, J. Vásárhelyi: *Microprocessor Load Measurement in an Embedded System*. 24th International Carpathian Control Conference (ICCC), Miskolc–Szilvásvárad, Hungary, 2023. 110–113. https://doi.org/10.1109/ICCC57093.2023.10178929

[9] G. M. Amdahl, Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In: AFIPS Conference, Proceedings, Vol. 30. 483–485. https://doi.org/10.1145/1465482.1465560

[10] U. Vishkin: *Is Multicore Hardware for General-Purpose Parallel Processing Broken?* Comm. ACM, 57, p. 35 (2014) The Future of Computing Performance: Game Over or Next Level?, https://www.nap.edu/read/12980/chapter/1, 31/08/2017. utoljára látogatott: 2024. 06. 06.

[11] A. N. Sloss, D. Symess, C. Wright: *ARM System Developer's Guide Designing and Optimizing System Software*. Elsevier, Morgan Kauffmann Publishers, ISBN: 1-55860-874-5, 2004, 702.

[12] Kovács S., Kóczy L.: *Application of the Approximate Fuzzy Reasoning Based on Interpolation in the Vague Environment of the Fuzzy Rulebase in the Fuzzy Logic Controlled Path Tracking Strategy of Differential Steered AGVs*. In: International Conference on Computational Intelligence, 1997. 456–467.

[13] Kovács S.: *Extending the Fuzzy Rule Interpolation FIVE by Fuzzy Observation*. In: Computational Intelligence, Theory and Applications, Springer, 2006. 485–497.

[14] S. Kovács, M. Gácsi, D. Vincze, P. Korondi, Á. Miklósi: *A Novel, Ethologically Inspired HRI Model Implementation: Simulating Dog-Human Attachment*. In: 2nd International Conference on Cognitive Infocommunications, 2011. 1–4.

[15] T. Tompa, S. Kovács: *Applying Expert Heuristic as an A Priori Knowledge for FRIQ-Learning*. Acta Polytechnica Hungarica, 17. (2020) 27–45.

[16] T. Tompa, S. Kovács, D. Vincze, Mihoko Niitsuma: *Demonstration of Expert Knowledge Injection in Fuzzy Rule Interpolation Based Q-Learning*. In: IEEE/SICE International Symposium on System Integration (SII), 2021. 843–844.

[17] I. Piller, D. Vincze, S. Kovács: *Declarative Language for Behaviour Description*. In: Emergent Trends in Robotics and Intelligent Systems. Springer, 2015. 103–112.

[18] R. Bartók, J. Vásárhelyi: *Two Methods for Autonomous Robot Obstacle Sensing and Application Programming Interface for Fuzzy Rule Interpolation*. 18th International Carpathian Control Conference (ICCC). IEEE, 2017.
[19] R. Bartók, J. Vásárhelyi: *Design of a FPGA Accelerator for the FIVE Fuzzy Interpolation Method*. International Journal of Computer Applications in Technology, 68/4. (2022) 321–331.