electronics

MDPI

Article

A RISC-V Processor Design for Transparent Tracing

Iván Gamino del Río, Agustín Martínez Hellín, Óscar R. Polo *, Miguel Jiménez Arribas, Pablo Parra, Antonio da Silva, Jonatan Sánchez and Sebastián Sánchez

Space Research Group, Ctra. Madrid-Barcelona Km. 33.600, Edificio Politécnico, Universidad de Alcalá, Alcalá de Henares, 28805 Madrid, Spain; ivan.gamino@uah.es (I.G.d.R.); agustin.martinez@uah.es (A.M.H.); miguel.jimeneza@uah.es (M.J.A.); pablo.parra@uah.es (P.P.); antonio.dasilva@uah.es (A.d.S.); jonatan.sanchezs@uah.es (J.S.); sebastian.sanchez@uah.es (S.S.)

* Correspondence: o.rodriguez@uah.es

Received: 21 September 2020; Accepted: 5 November 2020; Published: 7 November 2020

Abstract: Code instrumentation enables the observability of an embedded software system during its execution. A usage example of code instrumentation is the estimation of "worst-case execution time" using hybrid analysis. This analysis combines static code analysis with measurements of the execution time on the deployment platform. Static analysis of source code determines where to insert the tracing instructions, so that later, the execution time can be captured using a logic analyser. The main drawback of this technique is the overhead introduced by the execution of trace instructions. This paper proposes a modification of the architecture of a RISC pipelined processor that eliminates the execution time overhead introduced by the code instrumentation. In this way, it allows the tracing to be non-intrusive, since the sequence and execution times of the program under analysis are not modified by the introduction of traces. As a use case of the proposed solution, a processor, based on RISC-V architecture, was implemented using VHDL language. The processor, synthesized on a FPGA, was used to execute and evaluate a set of examples of instrumented code generated by a "worst-case execution time" estimation tool. The results validate that the proposed architecture executes the instrumented code without overhead.

Keywords: processor architecture; trace mechanism; critical software characterization; worst-case execution time; risc-v vhdl ip core; instruction set architecture; software instrumentation; hybrid analysis; real-time trace

1. Introduction

On the field of real-time systems, embedded software must meet a set of non-functional requirements that involve the interaction with the environment (reaction requirements) and the platform (execution requirements) [1]. Given the criticality of this type of systems, the verification of non-functional requirements is of the utmost importance, demanding the use of appropriate tools and techniques [2]. One of such techniques is real-time trace [3], which enables the observability of the embedded software. It uses a dedicated hardware interface as a trace port in the deployment target in order to provide, in real time, information about the software execution context. A logic analyser, or other dedicated device, is connected to the trace port so the provided information can be captured and analysed off-line. Real-time trace can be combined with code instrumentation as a simple way to enable the observability of an embedded software system during its execution. The code instrumentation mechanism consists of selective insertion of trace instructions at the code locations where the observation is required. The benefits of this technique are the flexibility to determine what information is traced and where to insert the trace instructions. The main drawback is its intrusiveness, as it introduces an overhead in the execution time of the instrumented code. Despite this, its advantages

of flexibility and ease of application have led to its widespread use for embedded system analysis, as shown in [4–7].

Non-intrusive hardware-based solutions were designed as described in [8–10] as an alternative to code instrumentation. However, these solutions lack the flexibility of code instrumentation, and generate a large amount of trace information whose real-time analysis is difficult to manage.

One of the types of analyses where code instrumentation is used is the estimation of Worst Case Execution Time (WCET) [11]. This analysis is mandatory for critical embedded software systems, where correct operation depends on the compliance of real-time requirements [12]. To attain this, the software system requires a deterministic behaviour, so that the WCET of the code can be obtained and a real-time schedulability analysis can be performed [13]. There are multiple ways to obtain the WCET of code. A successful approach presently is referred to as hybrid analysis [14], which combines the static analysis of the source code with the analysis of the time-stamped traces generated by the execution of the software on the target. The static analysis determines the specific points in the source code which need to be traced. If instrumentation techniques are used to obtain the traces, the static analysis must define where the tracing code is added. Finally, with the support of the specific processor hardware and using a logic analyser, the instrumented code is executed and execution times are uniquely obtained at each of the selected points. The code instrumentation, however, introduces a runtime overhead that modifies the behaviour of the software system under analysis, creating what is known as the "probe effect" [15].

The design and implementation presented in this paper introduces an internal processor structure which eliminates the runtime overhead of the code instrumentation. By using an ad hoc structure processor design and adding a specific trace instruction to the Instruction Set Architecture (ISA), the solution enables the following features: (1) fetching two consecutive instructions in a single cycle; (2) decoding two instructions in parallel in order to detect if one of them is a trace instruction; (3) executing the detected trace instruction in parallel, synchronized and conditioned to the complete execution of the preceding non trace instruction. In this way, it allows the tracing process to be non-intrusive in terms of execution time inside the pipeline, since the sequence and the execution time of the program under analysis are not modified by the introduction of the traces, as they are executed in a parallel path, as long as there are not extra accesses to memory.

The design above is based on a previous contribution of our research group. Specifically it is based on the registered patent with number ES2697548A1 [16]. This paper presents the implementation of this design on a RISC-V processor architecture [17,18], which enables a way to instrumentate the code without being intrusive in terms of execution time. The solution avoids the undesired discrepancy between the execution time of the system with and without instrumentation, adding more flexibility in the process than other solutions such as the patent proposal US 2017147472A1 [19], which autonomously traces every branch or jump instruction.

The interest in having a mechanism to be able to implement the code and obtain trace information without adding any overhead to the execution time is of particular relevance in real-time systems with a high degree of criticality. Examples of these systems include those used in aerospace, automotive, medicine or plant control, including nuclear power plants since they all require characterization of their temporal response. The overhead introduced by the instrumentation causes that the response time of the system being characterized does not fully fit the system that is finally deployed. In general, situations in which the temporal response of the systems differs from the estimated one, should be avoided. A clear reason for this is that a catastrophic failure could happen if the execution time exceeds the estimate of the WCET. However, more in line with the problem addressed by this paper, an overly pessimistic approach may lead to an under-utilization of computing resources resulting in an additional unit cost of the final system [20].

The objective of this article is twofold, on the one hand, the work is a demonstration that the concepts presented in the patent ES2697548A1 can be implemented on a real processor providing also a method of verification that such implementation is correct, based on the instrumentation

examples generated by a commercial tool. On the other hand, the paper evaluates the cost of this implementation in terms of the FPGA resources used. The work is of special interest for the development of on-board satellite embedded systems for two reasons: Firstly, the RISC-V processor is being considered by the aerospace industry as a candidate to be part of the On Board Data Handling (OBDH) of spacecrafts [21–23]. Secondly, the use of FPGAs to support the processing capabilities in space missions is a successful trend [24,25], and because of the flexibility it brings, future prospects is that this trend will continue to be consolidated [26].

The remainder of this paper follows the ensuing structure: first, in Section 2, a brief overview over the state of the art is exhibited. Next, in Section 3, the design of a classical RISC pipelined architecture is presented, along with the design modifications adopted to accomplish the goal of this work. Then, in Section 4, the specific implementation details of the RISC-V core with the corresponding modifications are described. Afterwards, in Section 5, the different programs and algorithms used to test this implementation are highlighted, meanwhile in Section 6 the results obtained are discussed. Finally, in Section 7, the conclusions are recapitulated.

2. Related Work

The characterisation of software behaviour under any circumstances is one of the challenges of software engineering. The key role that software plays in critical embedded systems have led to a significant amount of research aimed at trying to estimate the software behaviour from different perspectives, such as its energy consumption [27], reliability [28] or timing response [29]. Concerning this last aspect, obtaining the exact WCET in a completely automated manner for every input of a program is not feasible, as it would mean finding a solution to the halting problem [30]. However, in recent decades, the estimation of the WCET of real-time software has been one of the most relevant topics of interest.

One of the basic techniques that was used to calculate WCET is the end-to-end testing based on the obtaining of the denominated High-Water Mark Time (HWMT) [31]. This method involves executing from start to end a block of critical software under exhaustive test conditions. Finally, when it is deemed satisfactory, the process is finished and the longest execution time of that piece of code, plus some given margin, is considered to be a reference during the rest of the software design. However, this value is not guaranteed to be bound to the actual WCET as there is no knowledge of the full execution path inside the block and hence there could have been some initial conditions that made the program last longer.

The dynamic approach of the end-to-end testing can be improved with the addition of instrumentation that enables the measurements of the WCET [32]. This method inserts specific instructions that allow, with a certain degree of hardware support, to know the time when the software takes the different paths of execution inside the analysed block of code. Hence, by analysing the obtained measurements, it is possible to bound the WCET to some value with some uncertainty. Nevertheless, this kind of approach introduces the "probe effect" problem, previously mentioned, as the instrumentation instructions have to be executed and stored in memory, which constitutes a runtime overhead that modifies the behaviour of the software system.

In parallel to dynamic analysis, a different strategy was explored called static analysis [33]. This method consists of estimating the WCET of a program without the need for executing its code, only with the binary code and a mathematical model of the processor's behaviour. The advantage of this kind of analysis is that it completely eliminates the "probe effect" problem as there is no code execution. The approach, however, does not take in account that building a time-predictable system [34], concerns not only hardware, but also software [35], so the mathematical model of the processor's behaviour may not be accurate enough. Although there have been proposals of time-predictable processors [36,37] that could support this type of mathematical models, they require that software would be developed under specific constraints and using complex compilers. Because of that, the use of these time-predictable processors in real applications is not extended,

while the modern processors that usually are used to develop the embedded system are so complex that, to bound the WCET, the estimations of static analysis are generally too pessimistic [38].

As it was mentioned in the introduction, modern approaches try to tackle these problems using as an alternative a combination of both static and dynamic analysis for obtaining the WCET of critical software [39], making what is nowadays called hybrid analysis.

However, inside hybrid analysis, there are many different strategies as this is a current field of research. For example, such analysis can be end to end, meaning the full program is executed under different conditions to obtain the measures of the different paths. Meanwhile, other more advanced approaches divide the program into small segments executing and measuring each of them separated to compute the estimation of the WCET, alleviating the need to find a single test vector to inspect the slowest execution path [9].

One aspect to be determined in the hybrid analysis, and a field of research, is whether the instrumentation occurs at the source level or the object level. Programmers are accustomed to working at the source level. Hence, if the instrumentation occurs here, making sense of what is happening inside the program afterwards, during execution, is relatively easy as it is evident where the trace instruction is inserted in the source code. However, tracing at the source level generates the "probe effect" as instrumentation is always necessary. On the other hand, object-level tracing can be accomplished without instrumentation (and therefore without overhead) with techniques such as branch tracing, as used in [40,41]. This approach, however, clouds the understanding of what is being executed. For example, from the perspective of a running CPU, it is complicated to map a jump instruction in the object code to the corresponding statement of the source code function that is being analysed. Even though this lack of understanding can be mitigated as is explained in [10,41], the other main disadvantage of object-level instrumentation is the complex hardware support it needs. There has to be a hardware module, usually called Embedded Trace Unit (ETU), in charge of tracking events, tracing and storing or writing them to an external device, such as for example the IEEE-ISTO 5001 (Nexus) [42] standard hardware debugging interface or ARM's CoreSight technology.

An additional attribute of hybrid analysis is the moment when the main inspection of the traces gathered with the instrumentation is carried out. If the examination occurs after the execution of the program, this is considered offline analysis. In contrast, when the study coincides with the execution of the program, it is labelled online analysis.

The main advantage of offline analysis is the simplicity of its realisation, as it requires very little or no hardware support. On the other hand, it presents the downside of creating a lot of data, needing substantial bandwidth and buffers to store it, causing it to be hard to obtain meaningful insights. In juxtaposition, online analysis can decide which trace data is important enough to be stored for later and what can be overwritten during program execution. In this way, the problem displayed by offline analysis is solved but with the compromise of needing a lot of complex hardware support and additional external devices to perform the analysis in real time. For example, in [8,9] there are two instances of developments of this latter kind. In them, an online hybrid analysis workflow was developed where, after a preprocessing stage of the code, the traces of the execution provided by the architecture ETU are redirected to an FPGA where they are analysed in real time by a statistics module. Finally, in an offline postprocessing stage, with the data provided by the statistics module, the WCET can be obtained.

In contrast to the methods mentioned above, the solution presented in this paper searches for an intermediate solution. Instead of building such a complex hardware support system, the approach proposes an affordable increase of the hardware support for current offline hybrid analysis tools such as RapiTime [43] nullifying the execution time overhead caused by the instrumentation.

Besides the mentioned differences, the cited non-intrusive trace support alternatives are focused on execution path tracing, aimed mainly to estimate the WCET of code, but also to calculate other metrics such as the coverage obtained by testing [44]. These non-intrusive alternatives, however, are not suitable to support the selective observation of other context execution information that is part

of the software behaviour characterization, as the processor structure presented in this paper does. Specifically, a non-intrusive stack usage monitoring based on instrumentation, as proposed in [45], can be supported by the dedicated parallel pipeline that is implemented in this work. It is, therefore, a more general solution that provides code execution observability based on a form of instruction-level parallelism [46]. The parallelism, as in superscalar processors [47], uses a dedicated execution unit for a specific type of instruction, the trace instruction, but in this case the instruction is executed synchronously with the preceding instruction, and only with the objective of tracing its execution. As a result, the overhead that a sequential execution of both instructions would entail, is eliminated. The synchronization between both pipelines, also, avoids the out of order execution of superscalar processors that produce so-called time anomalies, and cause serious problems for existing WCET analysis methods [48].

3. Processor Design

The proposed solution begins with the description of a standard RISC pipelined processor design on which a set of modifications affecting the pipeline design was implemented. This solution can be applied in many different RISC architectures such as ARM, SPARC, or RISC-V, as the modifications are architecture-agnostic. They only transform the underlying structure to support non-intrusive tracing. First, the baseline pipelined processor will be briefly described after which the modifications will be detailed including a basic example of operation.

The overall scope of the modifications to be carried out consists of the introduction of a specific instruction for code instrumentation, together with the modifications of the processor pipeline described below. This new instruction was defined and implemented to serve as the trace trigger of the preceding instruction. RISC-V ISA specification gives great importance to the flexibility to implement specialized processors. For this, the specification leaves free operation codes, not usable in standard extensions, to be defined in a customized way for a specialized design. One of those specified free opcodes was used to implement the proposed instrumentation operation.

3.1. Baseline Pipeline Structure

The base structure of a RISC-V processor, based on a 5-stage pipeline, was designed. This design will serve as a foundation for making the proposed modifications and also as a benchmark processor for comparative data.

As shown in Figure 1, between each two of the processing stages there is always a register. These registers are in charge of propagating the result between stages, allowing the synchronization of the pipeline. In addition, there is a data forwarding unit (FU) in charge of overcoming data hazards which can occur during the execution of a typical program.

The 5 processing stages of the pipeline are the following:

1. Instruction Fetch stage (IF): This stage commands the insertion of the instructions inside the processor's pipeline with the purpose of executing them.
2. Instruction Decodification stage (ID): This stage is in charge of extracting the meaning of the instruction previously supplied by the IF stage. Once the meaning of the instructions is decodified, it creates the control signals for each of the remaining stages allowing them to perform its defined labour.
3. Execution stage (EX): This stage works according to the control signals provided by the ID stage, governing the operation of each instruction. For instance, in a jump instruction, it decides whether or not to take a branch and determines its respective jump address.
4. Memory stage (MEM): This stage performs the process of storing and loading data to and from data memory and general purpose registers.

5. Write-Back stage (WB): This stage chooses which data is stored back in the general purpose registers, if the corresponding instruction needs it, whether it is the result of the operation from EX stage, or the value obtained inside the MEM stage (in case of a load instruction).

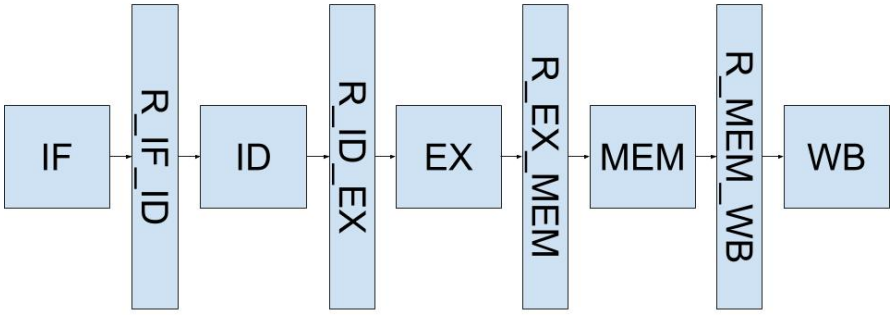

Figure 1. RV32XTrace baseline pipeline structure.

3.2. Processor Structure Modifications

The proposed structure detects trace instructions and executes them synchronized in parallel with regular processor instructions but conditioned to the complete execution of the preceding instruction. In this way, it allows the tracing process to become less-intrusive since the sequence and execution time of instructions at pipeline level becomes transparent as trace instructions are executed in a parallel path.

The main elements which were modified of the baseline structure, shown in Figure 2, are: (a) the instruction fetch stage, which now has a double reading port; (b) a new trace pipeline path was added; and (c) the decoding stage, which now is in charge of decoding both instructions supplied by the fetch stage and providing the control values to the trace pipeline path for routing the trace along the instruction to trace.

The first modification is the dual-port fetch stage depicted in Figure 2. It allows two instructions to be loaded simultaneously to the decoding stage so that they can be decoded in parallel. The stop signal of this stage is used to model possible waiting states and can be activated, for example, after a processor reset or as a consequence of a jump becoming effective, which causes the injection of bubbles in the pipeline. The stop signal will be deactivated in order to notify the IF stage that the instructions are available for loading. The instructions in the decoding stage always, noticeably, correspond to instructions stored in consecutive words in memory.

The second modification is the addition of a second pipeline, as shown in Figure 2, that propagates the trace instructions at the same time as the regular ones are executing. This is necessary to guarantee the parallel execution of both instructions.

Finally, the third modification is in the Instruction Decodification (ID) stage, displayed in Figure 2, to determine if the decoded instructions are of trace type or belong to the regular instruction set. Three scenarios may apply: (a) having one trace and a non-trace type instruction, (b) two non-trace type instructions or (c) two trace type instructions. In scenario (a), when there are a couple of instructions of a different type, the trace type one is routed to the trace pipeline while the non-trace type instruction is processed by the Control Unit (CU) and transmitted to the following stage. In scenario (b), when both instructions are of regular type, the first of the two instructions, taking into account its position in memory, is processed during the actual cycle, while the other is waiting to be processed until the next cycle. Lastly on scenario (c), having a pair of trace type instructions does not make any sense and is discouraged, as trace instructions, by definition, have to be preceded by a regular instruction. In case this scenario occurred, the second of the trace instructions would be discarded, producing an idle cycle over the version of the code without instrumentation.

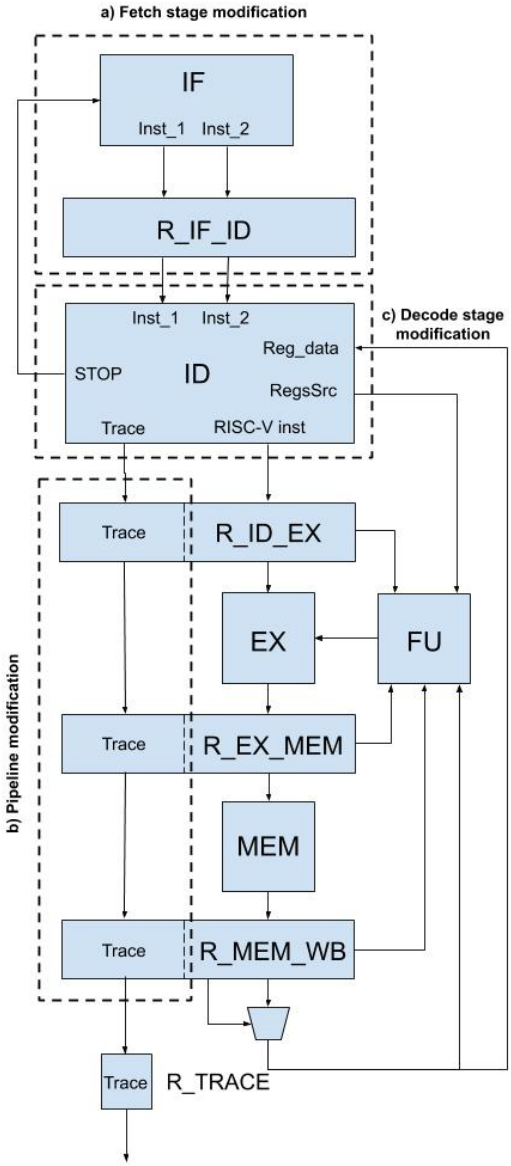

Figure 2. RV32XTrace top-level design with the following modifications: (**a**) Modified IF component to support a double instruction port and enlargement of the pipeline register between IF and ID stages for holding both instructions, result of the IF stage; (**b**) Enlargement of the pipeline registers between ID and WB stages featuring the trace instructions pipeline; (**c**) Modified ID stage to be able to dispatch both input instructions to each corresponding pipeline and to control trace instructions' pipeline.

3.3. Comparative Walkthrough

The following walkthrough is presented to clarify the functionality of each of the previously mentioned modifications. This example is a simple C language program which adds the elements of one array named "operands", storing the final value into a variable called "result". The code is the following:

RVS_I() is a function-like C macro with a single argument that is later expanded by the preprocessor into the corresponding trace instructions. The references to this macro are inserted by the RapiTime tool in specific locations of the code. The argument passed to the macro is the identifier that works as the data field of the trace instruction. Identifiers are assigned by the RapiTime tool during the instrumentation process. Each identifier is unique and directly linked to the code's

location. After the execution of a trace instruction, the assigned identifier can be captured and time tagged by a logic analyser and later used to estimate the WCET.

Code 1: Walkthrough example C code.

```
void main(void){
int operands[4];
int result = 0;

int length = sizeof(operands)/sizeof(operands[0]);
RVS_I(10);
operands[0]=1;
operands[1]=2;
operands[2]=3;
operands[3]=4;
RVS_I(11);

for(int i=0; i<length; i++){
RVS_I(12);
result += operands[i];
RVS_I(13);
}
}
```

The walkthrough is specifically focused on the following 2 statements:

Code 2: Statements to be analyzed.

```
operands[3]=4;
RVS_I(11);
```

In addition, the result of compiling these statements are the following assembly instructions:

Code 3: Resulting instructions to be analyzed.

```
30:    fcf42e23            sw    a5,-36(s0)
34:    00400793            li    a5,4
38:    fef42023            sw    a5,-32(s0)
3c:    0000058b            0x58b
```

It should be noted that just the last three assembly code instructions correspond to the previous two C code statements translation. The first instruction is included because, as it is mentioned before, instructions are fetched in pairs. As a result, there are two pairs of aligned instructions: the first composed by instructions located at 0x30 and 0x34 addresses and the second pair at 0x38 and 0x3c addresses.

An aspect to take into account of this code example is that the trace instruction located at 0x3c is the second instruction of the pair. As it was mentioned before, in the case of having a heterogeneous pair of instructions, one regular instruction and one trace instruction, the order inside the pair causes a variation in the functionality of the pipeline. Because the nature of the trace instructions is to trace their preceding instruction, if the trace instruction occupies the first position in the instruction pair, this means that the instruction to be traced is the second instruction of the previous instruction pair. An example of this situation can be seen in Code 4 with the instruction pair composed of instructions at 0x18 and 0x1c addresses. In this example, the trace instruction, located at 0x18 address, should be synchronised with the instruction to be traced, instruction at address 0x14 in the code. A multiplexer that selects the trace instruction between stages ID or EX will allow this synchronization. As this is not the case in Code 3 example, there will be no need to perform any synchronization.

Code 4: Example code of a situation in which a trace instruction occupies the first position of the instruction pair.

```
10:    00400793            li    a5,4
14:    fef42223            sw    a5,-28(s0)
18:    0000050b            0x50b
1c:    00100793            li    a5,1
```

The walkthrough will cover and follow the execution of the last pair of instructions through all the stages of the proposed solution's pipeline explaining its corresponding functionality.

At first, the instructions have to be inserted into the pipeline through the IF stage. As can be seen in Figure 3, the new instructions are fetched in parallel from the instruction memory and supplied to the first intra-pipeline register, placed between the IF and the ID stage. Meanwhile, the intra-pipeline register is feeding the ID stage with the previous pair of instructions.

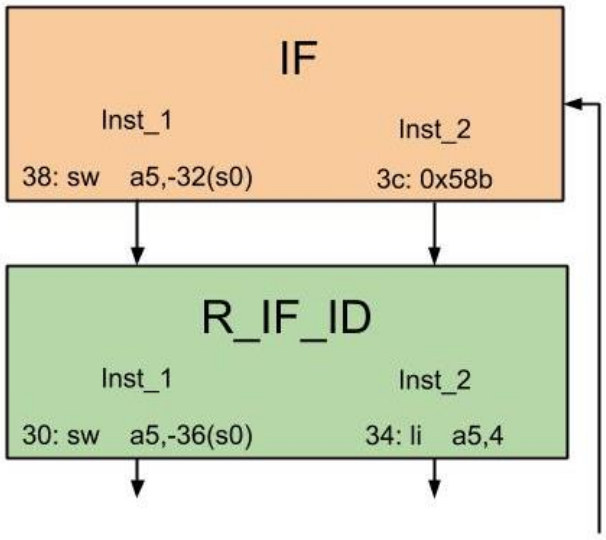

Figure 3. Trace's instruction pair fetch stage.

The instructions from each pair are organized as "Inst_1" and "Inst_2". The first one will be placed and propagated by the "Inst_1" port, while the second one will do it by the port "Inst_2" port.

Once the new cycle begins, the R_IF_ID intra-pipeline register will update the instructions previously supplied by the IF stage and feed them to the next stage, i.e., ID.

The ID stage will arrange into which pipeline the instructions will be inserted, filtering the trace instructions from the regular ones. Figure 4 represents how both instructions are routed to their corresponding pipeline: the identifier of the 0x3c trace instruction on one side and the data and control values of the 0x38 regular instruction on the other. There is also a "STOP" port which is in charge of stopping the insertion of new instructions into the pipeline by the IF stage if needed. Lastly, the "Sel_tr" port manages the multiplexer in charge of the overtaking between traces in those cases where the trace instructions are placed first in the instruction pair.

On the next cycle, the intra-pipeline register between the ID and the EX stage (R_ID_EX), will update the processed regular instruction, shown on the left side of Figure 5, and the trace ID, on the right side, provided by the ID stage. This process will be analogous for each of the remaining intra-pipeline registers.

Figure 5 represents both pipelines, the left one for traces and the one on the right for regular instructions. The most relevant feature of this image is the presence of the aforementioned multiplexer in charge of the overtakings between traces. In this example, there is no overtaking, so the result value of the trace ID extracted by the multiplexer will be the one fed by the R_ID_EX and not the one supplied by the ID stage. In parallel, the regular instruction is being executed by the

EX stage components, and its result propagated forward to the next intra-pipeline register. In the case of Code 4 an overtaking will be effective by trace instruction, from 0x18 address, which will be in the ID stage, in order to be processed in parallel with its corresponding instruction to trace, instruction from 0x14 address, currently at EX stage.

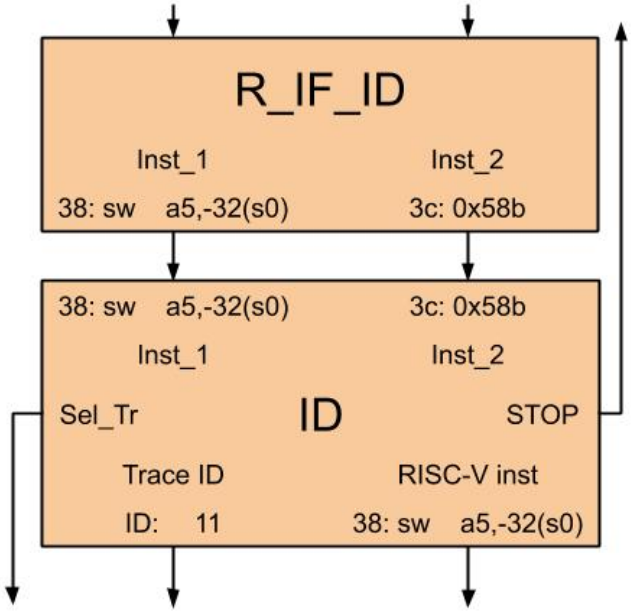

Figure 4. Trace's instruction pair decodification stage.

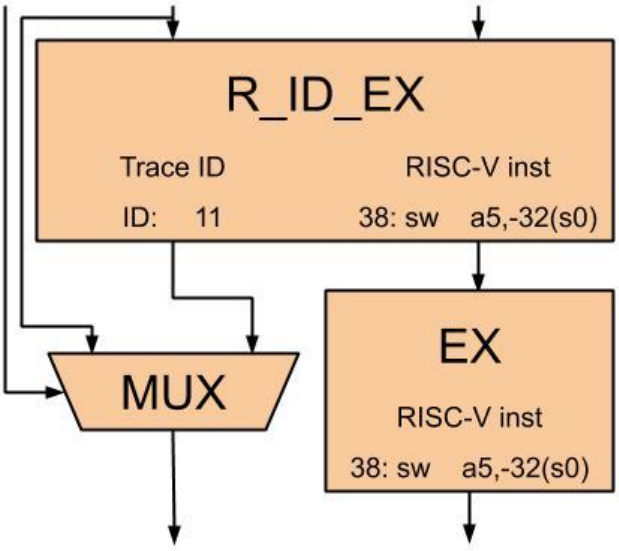

Figure 5. Trace's instruction pair execution stage.

Figure 6 represents the functionality of the next cycle of the stage. On this cycle there is no treatment of any kind related to the trace ID: it is only propagated to the next intra-pipeline register. Meanwhile, in a similar way as in Figure 5, the regular instruction is being processed by the MEM stage components and its result also propagated to R_MEM_WB.

Finally, Figure 7 shows the last step of both instructions. At this time, the trace ID is supplied to R_TRACE, a register that only changes its value when the value of the trace ID is different from 0. This register is used to trace the execution of the program by propagating its value to external monitoring hardware.

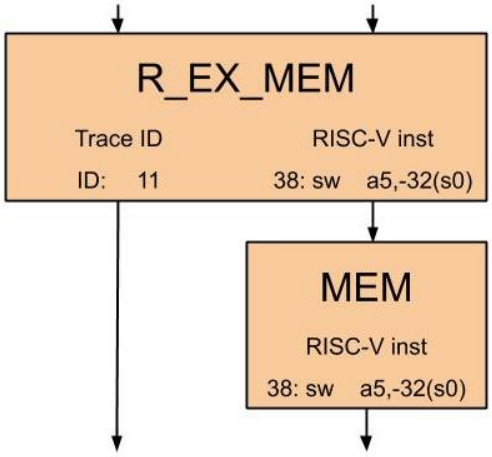

Figure 6. Trace's instruction pair memory stage.

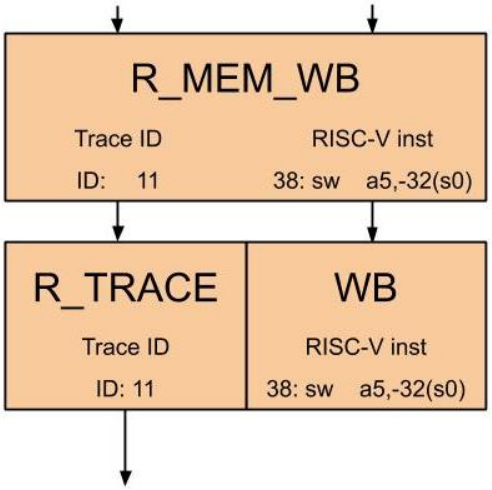

Figure 7. Trace's instruction pair write-back stage.

Meanwhile, the regular instructions come to their end, being processed by the components of the WB stage, writing when necessary the result of the instructions back into the general-purpose registers.

Taking a time-waveform perspective, Figure 8 shows the execution of the instructions which compose this walkthrough example between the yellow and blue markers. It is noteworthy that the four instructions consume only three clock cycles, as the second instruction pair is heterogeneous, taking only one clock cycle to execute both instructions. On the other hand, a standard RISC-V processor spends four cycles, as shown in Figure 9.

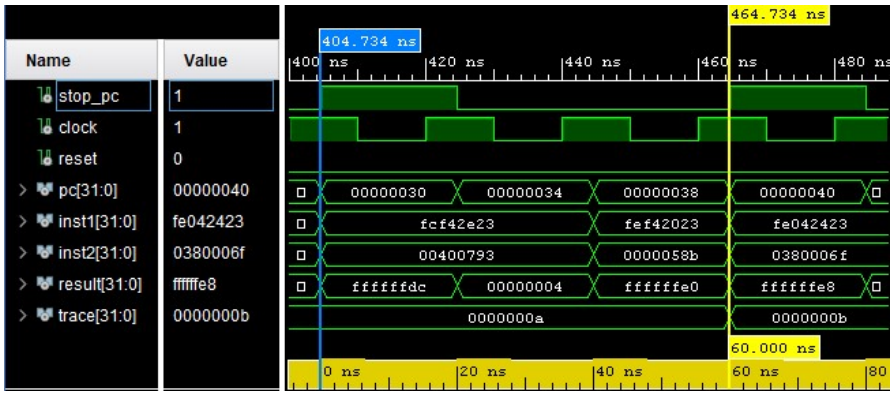

Figure 8. Waveform of the example's result execution on a modified processor.

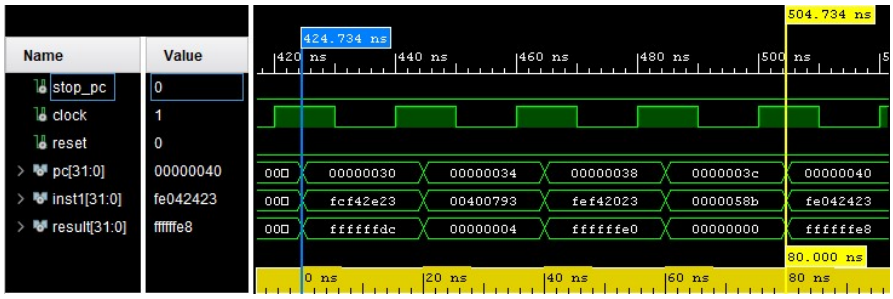

Figure 9. Waveform of the example's result on a non-modified processor.

3.4. Singular Trace Execution Cases

As it was mentioned before, the order inside an instruction pair is important to determine the instruction to be traced. Nevertheless, as the implementation is sufficient enough to solve this discrepancy, during the static analysis provided by other tools, this kind of alignment information is not taken into account. On the other hand, looking a bit further, in order to obtain meaningful results from the instrumentation, the traces have to be well placed in the code.

An example of this situation is the placement of a trace instruction right after a jump instruction, as shown in Code 5. By definition, the trace instruction will never trace the jump instruction as it will always jump to another point of the code, invalidating the execution of this instrumentation point. Conversely, the trace execution will be made effective if another jump has its destination to that point. Figure 10 shows the execution of an unconditional jump and how the trace associated with this instruction is not executed, as the trace value keeps unchanged. In contrast, Figure 11 shows the execution of the previous trace as a destination of another jump.

Code 5: Example of a trace instruction right after an unconditional jump.

```
10:    020000ef            jal    ra,30 <self_test>
14:    0000050b            0x50b
```

Code 6: Example of a trace instruction right after a conditional jump.

```
d8:    fce7d0e3            bge    a5,a4,98 <self_test+0x68>
dc:    0000058b            0x58b
```

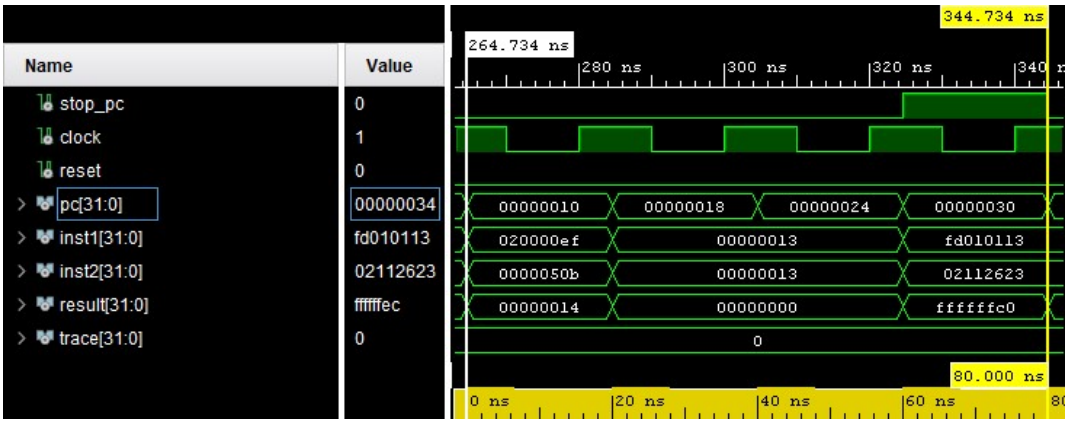

Figure 10. Execution results of a pair composed of an unconditional jump (0x10) and a trace instruction (0x14). In the first cycle, the jump is executed, and the trace instruction is invalidated. At the following two cycles, NOP instructions are executed due to the unconditional jump. In addition, in the last cycle, the jump's target instruction (0x30) is executed.

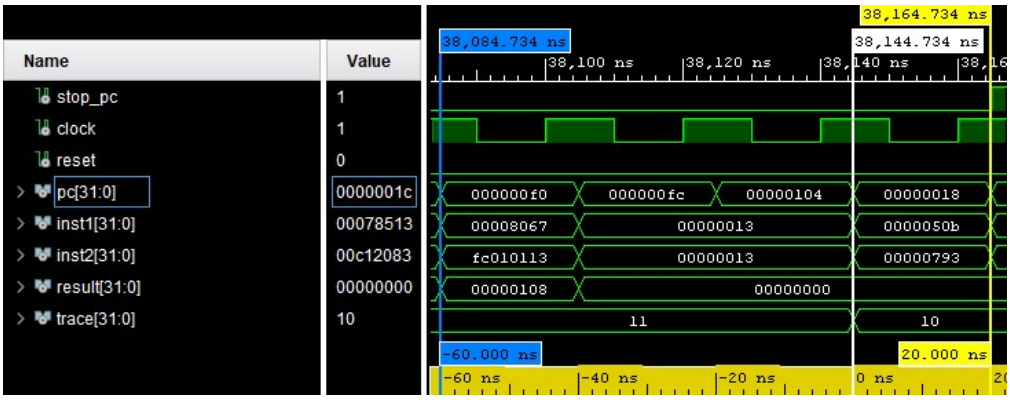

Figure 11. Execution results of an unconditional jump (0xf0). In the first cycle, the jump is executed. At the next two cycles, NOP instructions are executed due to the unconditional jump. At the last cycle, the jump's target instruction (0x14) is executed as can be seen on the variation from 11 to 10 of the trace value.

In the case of a conditional jump, as in the example Code 6, there are two different behaviours which could occur, i.e., the jump could be effective or not. In the first situation, the behaviour of the processor would be similar to the one shown in Figures 10 and 11. If the conditional jump is not effective, the trace associated with this instruction will execute successfully, as shown in Figure 12.

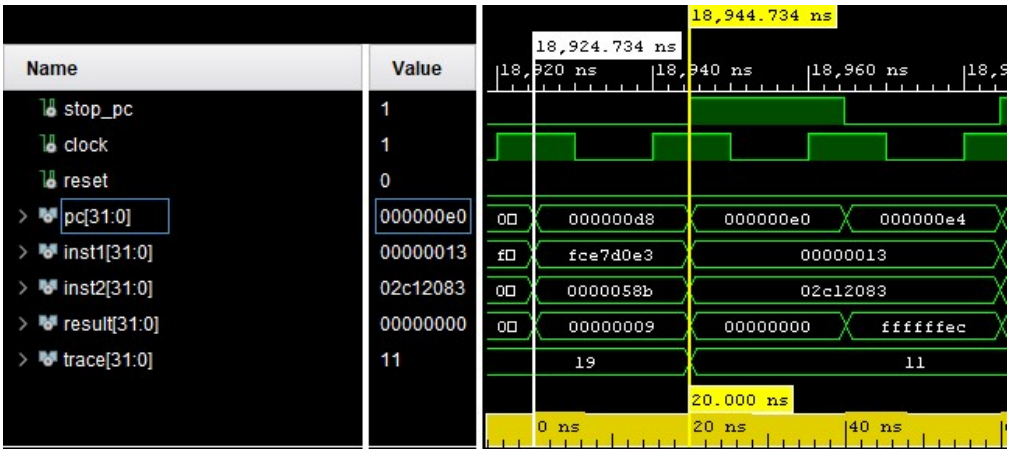

Figure 12. Execution results of a pair composed of a conditional jump (0xd8) and a trace instruction (0xdc). In the first cycle, the jump is not effective, and the trace instruction is executed, changing its value from 19 to 11. The remaining cycles execute the corresponding instructions after the conditional jump.

In summary, in regards to jump instructions and traces, a trace instruction is executed only when either its preceding conditional jump is not effective, or after an effective jump (both conditional and unconditional) that points towards the trace, preserving in these cases the integrity of the program execution.

4. Implementation

This section describes the hardware implementation of the modifications mentioned above. This implementation is based on a description model of a standard RISC processor architecture, in particular on the RISC-V ISA [17,18]. The advantages this architecture presents, namely, its open-source nature which has made it an attractive option for academic purposes, along with its promising performance, makes RISC-V a viable alternative for different applications, not only for those previously mentioned in the domain of aerospace industry, but also for IoT [49], or AI [50].

These architecture description models allow the generation of the manufacturing details of the device, which can be materialized on a programmable FPGA device (Field Programmable Gate Array) or an Application Specific Integrated Circuit (ASIC). To obtain the measurements closest to a real case scenario and to have the ability to test different solutions, the modifications were performed on an FPGA. Specifically, the Digilent Nexys 4 DDR board was used, which accommodates an XC7A100T-1CGS324C FPGA from Xilinx [51]. The characteristics of this Xilinx's FPGA can be consulted at [52] and in Section 6, an analysis of results with specific information about the resources used can be found. On the other hand, in industrial production or space project missions, it would be quite beneficial to have two alternative versions of the processor: one with the modification proposed and another one without it. The one with the modification would be useful in the on-board computer engineering model that is used to characterize the WCET of the code that is required for the software schedulability analysis. The version without the modifications would be used in on-board computer flying models to reduce the Rad-Hard FPGA or ASIC resource consumption.

This implementation was developed from scratch following both volumes from the RISC-V ISA [17,18]. Specifically, it was designed to fulfil the requirements to demonstrate the performance improvement of a system with the proposed modification over an equivalent non-modified system. Having this fact in consideration, two equivalent IP-cores were implemented, a vanilla one and another modified with the proposed solution. The main differences between each of them are described earlier, but as these adaptations were made from scratch, some changes were performed aiming to increase the performance and reduce the area used by circuits. The philosophy of the design was to maintain the stages computationally balanced.

Figure 13 shows a functional diagram considering the changes previously mentioned at the processor structure modifications Section 3.2. First, the instructions are decoded. Then, they are propagated to the HU (Hazard Unit), based on [53] implementation, to discover any possible data hazard and notify their existence to the TCU (Trace Control Unit). The TCU is in charge of placing each instruction provided by the decoders respectively in its dedicated pipeline path, becoming the most complex component of the proposed modification. In case of having two regular instructions, this component also solves the imminent structural hazard by paralysing the flux of instructions introduced by the fetch stage until the hazard is solved. In Figure 13, the framed (a) section represents the regular pipeline path, while the framed (b) section represents the pipeline path of the trace. It also be seen in the figure that the treatment of the regular instruction is performed by the CU (Control Unit), which is in charge of obtaining the control values for the subsequent stages. The TCU was placed before the CU to anticipate the decision of which pipeline path each instruction has to be directed.

To compare the improvement in performance resulting from the modifications in the design explained in the previous section, the following two equivalent implementations were developed.

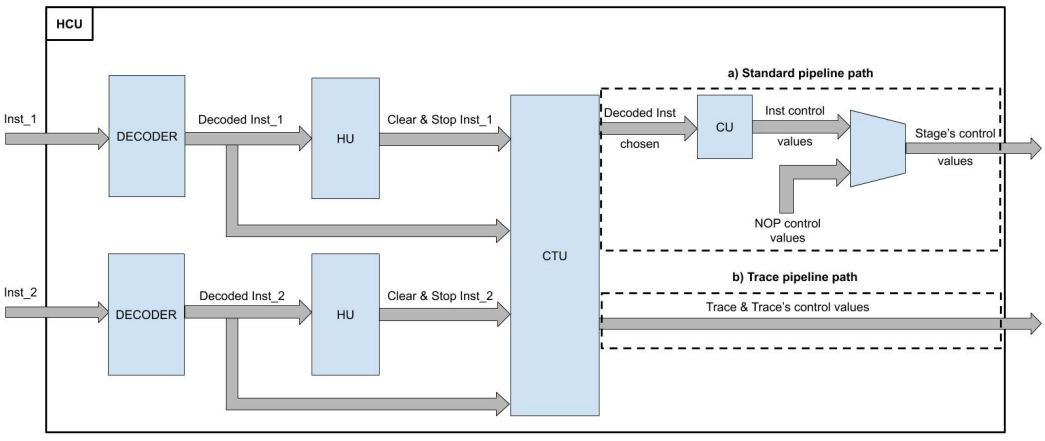

Figure 13. Resulting modifications applied on the RV32Xtrace Hazard Control Unit (HCU).

4.1. Development of a Vanilla IP-Core

First, there was the need to have a vanilla RISC-V segmented processor on which to apply and compare the proposed modifications. As such, as a base, an earlier own design was taken. This implementation lacked a segmented design, and hence, the first few steps were rearranging its functionality and transforming the single-stage processor to a multiple-stage segmented one. Once this task was completed the structure of this vanilla processor was the one aforementioned on Section 3.1, with five stages (IF, ID, EX, MEM, and WB) and a forwarding unit to overcome data hazards that can occur during program execution.

The correctness of the implementation was validated using the designed tests explained on Section 5, allowing the future comparison of the results between the vanilla and the modified versions, as it will be studied in Section 6.

4.2. Modification of the Vanilla IP-Core

With an already functional version of a RISC-V processor, the next step consists of applying the mentioned modifications according to the design concepts proposed in the patent with number ES2697548A1 [16]. Those modifications imply having to read two instructions at the same time, decode them in parallel and dispatch each of them to their appropriate pipeline. Hence, to fulfil the modifications requirements, there is the need for (a) a dual-port instruction memory, (b) a mechanism or logic in charge of the process of the instructions dispatching and (c) an additional pipeline for the trace instructions as well as a (d) duplicated decoder previously modified for decoding trace instructions.

As all the core components were implemented on the FPGA, changing a single port instruction memory for a double port one was straightforward. Due to the replacement of the instruction memory, it was also necessary to duplicate the size of the bank of registers placed between the fetch and the decode stages. In the case of the insertion of the additional pipeline, enlarging the banks of the pipeline registers from the EX to the WB stage would be enough to be able to propagate the trace in parallel with the instruction to trace execution.

On the other hand, the most important part of the mechanism for dispatching the instructions is represented by the TCU component. This component is the one that channels each instruction to its belonging pipeline. Furthermore, it also is in charge of monitoring and controlling the evolution of the traces going through their pipeline. The trace's pipeline implements multiplexers before the banks of registers. Those multiplexers have the function of suppressing the trace they propagate in case there is the presence of a bubble due to a jump or a hazard, as the trace value in these cases is no longer meaningful.

The TCU component has two differentiated tasks: to route each instruction to its corresponding pipeline and to control the execution of the traces along the rest of the remaining stages of the processor. The most significant and complex feature of this component is the determination of the pipeline into which each instruction should be transmitted.

As it is mentioned before, there is a case, the most common, in which two regular instructions were fetched. In this case, two instructions have to be dispatched to the same pipeline path, and as a consequence, a structural hazard emerges. To treat this structural hazard, the TCU has to release the first instruction into the regular pipeline while it is retaining the second instruction for release in the next cycle. To accomplish this, the TCU has to stop the normal flow of the program by paralysing the Program Counter register (PC); otherwise, the normal flow of the program will be affected by losing instructions in that wait cycle, needed to route those two preceding regular instructions correctly.

To make this possible, the TCU has to know if it has to hold back one of the instructions, propagate both of them or propagate the remaining one. This decision-making mechanism was implemented as a Mealy state machine. Figure 14 and Table 1 represent the Mealy machine used and its defined behaviour. During program execution, the TCU is always working in the INIT state unless a pair of regular instructions are fetched. If the component works in this state, both instructions are propagated, and there is no alteration nor stoppage in the pipeline's instruction flow unless there is a hazard. If a

hazard emerges and cannot be solved by the intervention of the FU, the instruction flow must be stopped to maintain data and program integrity. In another vein, if a pair of regular instructions were fetched, the TCU will dispatch the first one into the regular pipeline, sheltering the second, and will change its state to PENDING. The function of the transition to this state is to stop the pipeline's instruction flow and to guarantee the program's integrity. As soon as the sheltered instruction is inserted into the regular pipe, and as long as there is no hazard, the TCU will return to the INIT state. If there is a hazard, the flow will be stopped one or more cycles until it disappears.

There are other cases in which the program flow is altered, namely a jump, a branch instruction being effective or a hardware reset. In these cases the TCU will change to or remain in, the PENDING state reducing the penalty to only one cycle instead of two, as one of the two instructions is discarded.

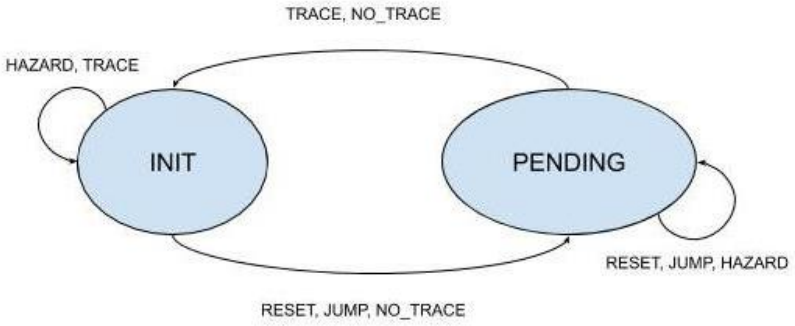

Figure 14. RV32XTrace Trace Control Unit Mealy machine.

Table 1. RV32XTrace Trace Control Unit Mealy machine.

Input	Meaning	Values
0	RESET	1XXXX
1	JUMP	01XXX
2	HAZARD	001XX
3	NO_TRACE	00000
4	TRACE	00001, 00010, 00011

Finally, having applied all the modifications mentioned above, the adaptation can execute computational programs along with their traces without introducing any intrusiveness related to the execution.

5. Comparative Tests

The application was validated by a series of tests that prove its effectiveness. Two different types of tests were used: the first subset tests the functionality of each stage of the implementation; the other subset tests that the execution of the trace instructions attached to the program does not introduce a time overhead. For the first subset of tests, a classical approach was adopted, using test-benches for each stage of the implementation that validate, under predefined inputs, that the outputs obtained are those expected in value and time.

This section focuses on the explanation of the second type of tests used to prove that the solution eliminates the execution time overhead of the trace instructions added as part of the instrumentation process. The tests were organized in pairs: one that executes a representative algorithm in the original processor, and another that executes a version of the algorithm that was instrumented with the RapiTime tool [43] in the processor with the modified structure. The results obtained are compared, using the cycles of execution of both versions, to determine that the proposed solution avoids the time overhead of the execution of the trace instructions.

The first test pair is based on the quicksort algorithm, which can be considered an illustrative and well-known sorting algorithm. It is not a trivial algorithm to compute, since it has such a high complexity level to obscure the functionality of the proposed solution and it is also considered an efficient sorting algorithm with a divide and conquer design. These characteristics make it an appropriate candidate for demonstrating how the resulting implementation eliminates the intrusiveness introduced by trace instructions in terms of execution time overhead.

The second pair is an example provided by RAPITA RVS tutorials, which is focused on making some basic mathematics operations to see how RapiTime instruments the different statements of the code. The purpose of this example is to show that, with the source code provided and implemented by RapiTime, the same execution time results are obtained.

To produce the second test set, one needs to write its corresponding program and build it with a RISC-V processor as a target. To do this, the compiler that was used in the first instance is GNU RISC-V Embedded GCC of the GNU MCU Eclipse project. This chain of Eclipse tools, now part of the xPack project [54], is mainly focused on C/C++ languages and bare-metal embedded systems. Therefore, the language used to create the test programs has to be C or C++. As RapiTime supports both languages, the tests were written in C due to its widespread use in the space industry. Finally, the RapiTime tool from RAPITA Systems [55] is used to track and analyse the code related to the critical systems, as well as the familiarity of the research group with its tools for the validation of the software of the Instrument Control Unit (ICU) of the Energetic Particle Detector (EPD) [56] on-board Solar Orbiter Mission, launched on 10 February 2020.

Once the environment is set up, the next step is to obtain a traced version of the tests using RapiTime. This version can be obtained using a configuration file which allows tracing the code with the variety of options that the tool provides. The quicksort test was traced defining a custom tracing profile in which the traces were placed at the entrance and exit of functions. In RapiTime's tutorial case, the already provided premade tracing configuration profile COV_178_DAL_C was used. This configuration profile is focused on tracing conditional statements and function exits, and it is used for software characterization under the DO-178C software aviation standard. Having both pairs traced, the last step to take is to associate the traces, inserted by RapiTime, with the trace instructions supported by the implementation.

At this point, the test set is complete and ready to be built up and recorded on the FPGA to obtain results.

6. Result Analysis

In this section, the results of the tests described in Section 5, obtained from the execution on the vanilla IP-core and on the modified implementation, from now on RV32Xnotrace and RV32Xtrace respectively, are presented.

Table 2 results were obtained from the execution of the first pair of tests, based on the quicksort algorithm, with lists of elements of variable length.

Table 2. Number of cycles of execution of both instrumented and non-instrumented quicksort algorithm and with lists of different lengths on both RV32Xnotrace and RV32Xtrace.

	List Length	RV32Xnotrace (Cycles)	RV32Xtrace (Cycles)
Instrumented quicksort	4	755	731
Non-instrumented quicksort	4	731	731
Instrumented quicksort	8	1952	1894
Non-instrumented quicksort	8	1894	1894
Instrumented quicksort	16	4497	4373
Non-instrumented quicksort	16	4373	4373

As it can be seen, the amount of cycles required to complete the version of the algorithm that is not instrumented is the same in both implementations. This result is a good indicator as if the values were not the same it would mean that the different implementations were not equivalent and that the modifications introduced would have been intrusive. The same does not happen in the instrumented version of the algorithm, as would be expected due to the insertion of trace instructions caused by the RapiTime tool. This instrumented version has an increment of 3.94% more instructions respect to the not instrumented one. Respect to the increase in the number of cycles needed to execute these tests, there are 3.28%, 3.06% and 2.84% more cycles in the execution of the instrumented quicksort's code with 4, 8 and 16 element list length respectively in the RV32Xnotrace. The subtraction between each execution value shows the exact number of trace instructions that were processed in parallel in the proposed solution, as they take one cycle to execute in the vanilla processor.

It also can be seen that the number of cycles required in the implementation with the proposed solution is equal to cycles required in the standard implementation. This is another good indicator since it means that the trace instructions have not introduced any overhead to the default treatment of the algorithm. It is necessary to remark that the RV32Xtrace has the same critical path that RV32Xnotrace, so both processors work at the same clock frequency, and the comparisons based on execution cycles are, in fact, based on measures of execution time.

In case of RVS tutorial, the insertion of trace instructions supposes an increment of a 9.39% on the number of instructions on which the test consists of. With respect to the increase in the number of cycles needed to execute these tests, there are 4.30% more cycles in the execution of the instrumented RVS tutorial's code in the RV32Xnotrace. From the results shown in Table 3, and in an analogous way to the previous table, it can be corroborated that the code instrumented by RapiTime is executed in the modified processor without the typical interference caused by the use of the code instrumentation in a standard processor.

Table 3. Number of execution cycles of both instrumented and non-instrumented RVS tutorial test on both RV32Xnotrace and RV32Xtrace.

	RV32Xnotrace (Cycles)	**RV32Xtrace (Cycles)**
Instrumented	994	953
Non-instrumented	953	953

To determine how significant these tests are, a comparative of the "tracing patterns" of a real critical software instrumented with Rapita, and the patterns inside the quicksort and mathematica examples is provided. The term "tracing patterns" refers to the different combinations of synchronous executions of an instruction to be traced with its trace instruction, each one in its respective pipeline, according to the type of the traced instruction. These patterns were obtained as a consequence of how the instrumentation is generated with the objective of estimating the WCET by means of hybrid analysis, and they constitute a realistic scenario of operation of the RV32Xtrace.

Specifically, the software whose "tracing patterns" was compared is an instrumented version of the Boot Software (BSW) of the Instrument Control Unit (ICU) of the Energetic Particle Detector (EPD) on-board Solar Orbiter. The BSW is a criticality category B software, which cannot be replaced during mission, with minimal functionality comprising only the system start-up and the safe mode. This software was developed by our research group, and more details of its design can be found in [57]. Although the BSW is a much more significant example of a real embedded system than quicksort or RVS tutorial example, its size, which exceeds 32 KiB in its instrumented version, is too long to be executed in the current version of RV32Xtrace, which has no more than 4KiB memory. In addition, the board where RV32Xtrace was developed lacks the specific hardware that is controlled by the Board Support Package integrated in the BSW, so its execution on RV32Xtrace would have an unpredictable behaviour that could not help to complete the analysis. Because of that, the approach followed was to statically analyse the binary of the instrumented version of the BSW, the quicksort,

and the RVS tutorial, in order to find the "tracing patterns", so they can be compared. The objective is to determine if the tests carried out have covered the possible situations that the structure of RV32Xtrace would face on the BSW execution.

The results shown in Table 4 show the instructions selected by the Rapita instrumentation method to be traced on programs. In the first place, the most recurrent is the store type followed by conditional and unconditional jump. Finally, arithmetic instructions are also traced but with a very low incidence. As can be inferred, these types of instruction define the relevant "tracing patterns". It is also noted that as the quicksort test matches in every pattern, in the RVS tutorial test there are no arithmetic instructions traced, but provides a more incidence of the most frequent patterns. This means that both quicksort and RVS tutorial tests complement each other and cover the patterns shown in a real embedded critical software as it is the BSW of the EPD's ICU. It can be considered that the exhaustive analysis based on those tests is significant enough to validate the behaviour of the structure in the parallel and synchronous execution of the trace instructions without time penalty. Hereafter an update of both RV32Xtrace and RV32Xnotrace implementations is in progress in order to work in a board that can support the hardware resources, as watchdog, memory fault tolerance devices or telemetry and telecommand interfaces, that are demanded by a satellite critical software as the BSW is.

Table 4. "Tracing patterns" obtained from the static analysis of BSW, Quicksort and RVS tutorial tests.

Type of Traced Instruction	BSW	Quicksort	RVS Tutorial
Conditional jump	446	1	15
Unconditional jump	497	2	6
Store	744	4	22
Arithmetic	4	1	0

The results presented up until this point are focused only on the time or cycles a program takes to execute. To complete this analysis, the rest of this section is focused on results related to space consumption and which resources are being used the most on the exhibited implementation, as well as power consumption.

In absolute terms, as it can be seen in Table 5, a modified version of the processor takes 435, approximately a 0.68%, more slice LUTs and 185, a 0.15%, more slice register resources of the FPGA compared to the original one.

On the other hand, the relative increase in the used resources on the modified version is approximately a 36% $(((1651/1216 + 681/496)/2 - 1) \times 100)$ higher compared to the vanilla one. This difference, however, is not very significant taking into account that both versions implement a significantly reduced portion of the whole RISC-V ISA. Hence, as can be checked in Table 6 based on the works of [58–60], the more ISA extensions implemented, the more modest this relative value will become, making the relative size of the modifications presented in this paper completely affordable.

Table 5. Resources used on the implementation of the CPU with and without the presented modifications.

	RV32Xnotrace	RV32Xtrace
Slice LUTs	1216 (1.92%)	1651 (2.60%)
Slice Registers	496 (0.39%)	681 (0.54%)

The same way there is an increase in the used resources, there is also an increase in power consumption. As it would be expected, the RV32Xtrace implementation takes more power than the RV32Xnotrace one. Specifically, it takes a 0.001 W more, resulting in a 0.86% (0.118W in RV32Xtrace and 0.117 W in the RV32Xnotrace) more power in the FPGA. All these measures were extracted from Vivado power analysis.

Table 6. Resources used on the synthesis of different CPUs implementing multiple ISA's extensions.

Processor Core	ISA	Frequency (MHz)	LUT	LUTsinc%	FF	FFinc%
SiFive E31	RV32IMAC	100	3614	12.04%	1642	11.27%
Pulpino RI5CY	RV32IMC	50	6748	6.45%	2577	7.18%
ORCA	RV32I	185.2	1354	32.13%	746	24.80%
freedom	RV32IMAC	32.5	2692	16.16%	1311	14.11%

7. Conclusions

Embedded software in real-time systems must meet some critical requirements that demand the use of appropriate tools and techniques. One of the most widely used techniques is real-time tracing that combined with code instrumentation provides a simple, but effective, way to enable the observability of software systems during its execution. These methods have the drawback of its intrusiveness, as they introduce overhead in the execution time of the code under test. This paper presents the detailed design, implementation and validation of a pipelined processor, based on RISC-V architecture, that enables the use of code instrumentation without being intrusive in terms of execution time.

The design was approached as a modification of a baseline vanilla RISC-V processor developed from scratch. The modification extends the RISC-V Instruction Set with a specific trace instruction, and it also adds a second pipeline to the internal structure to execute in parallel the trace instructions added during code instrumentation. Each trace instruction is executed in parallel and synchronously to the preceding instruction, which is the one selected to be traced, so the execution time overhead of code instrumentation is eliminated.

The solution was validated using a broadly extended use case of code instrumentation as it is the Worst-Case Execution Time estimation based on hybrid analysis. Specifically, a set of examples of code instrumentation for WCET estimation automatically generated by a commercial tool were used. The results show that the execution time inside the proposed adaptation with and without traces does not differ, and this time also coincides with the one obtained within the baseline vanilla processor without traces. The analysis was also extended by comparing the "tracing patterns" (pairs of type of instruction to be traced and trace instruction) found in these examples with those obtained by instrumenting the Boot Software of the control unit of the EPD instrument on-board the satellite Solar Orbiter. The comparison shows that the exhaustive analysis based on those tests is significant enough to validate the behaviour of the structure in the parallel and synchronous execution of the trace instructions without time penalty. The results means that our solution enables code tracing, which is transparent, deterministic, and precise with respect to the original code. Finally, remark that the work presented in the paper is also a demonstration of the viability and correctness of the concepts of the patent ES2697548A1, registered by our research group.

Author Contributions: Conceptualization, A.M.H. and Ó.R.P.; methodology, A.M.H. and I.G.d.R.; software, I.G.d.R. and P.P.; validation, I.G.d.R., J.S. and M.J.A.; formal analysis, I.G.d.R.; investigation, I.G.d.R.; resources, S.S.; data curation, I.G.d.R.; writing—original draft preparation, I.G.d.R., M.J.A. and Ó.R.P.; writing—review and editing, A.M.H., A.d.S., I.G.d.R., J.S., M.J.A. and P.P.; visualization, I.G.d.R.; supervision, A.M and Ó.R.P.; pr administration, S.S.; funding acquisition, A.M.H. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by predoctoral aids "Design and Implementation Implementation of RISC-V Architectures Improvements" and "LEON Processor Improvements" of the Youth Employment Initiative (YEI) of the European Social Fund (ESF), under the Operational Program of Youth Employment (POEJ), grants PEJD-2018-PRE/TIC-9016 and PEJD-2019-PRE/TIC-16525.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Henzinger, T.A. Two challenges in embedded systems design: Predictability and robustness. *Philos. Trans. R. Soc. A Math. Phys. Eng. Sci.* **2008**, *366*, 3727–3736. [CrossRef] [PubMed]
2. Vermeulen, B. Functional debug techniques for embedded systems. *IEEE Des. Test Comput.* **2008**, *25*, 208–215. [CrossRef]
3. Berger, A. *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*; CRC Press: Boca Raton, FL, USA, 2001.
4. Lojewski, E.; Walcott, K.R. LwProf: Lightweight Profiling and Coverage Tool for Embedded Software. In Proceedings of the International Conference on Embedded Systems, Cyber-physical Systems, and Applications (ESCS), Las Vegas, Nevada, USA, 30 July–2 August 2018; pp. 10–16.
5. Bhatti, N.A.; Mottola, L. HarvOS: Efficient code instrumentation for transiently-powered embedded sensing. In Proceedings of the 2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), Pittsburgh, PA, USA, 18–21 April 2017; pp. 209–220.
6. McGrath, W.; Drew, D.; Warner, J.; Kazemitabaar, M.; Karchemsky, M.; Mellis, D.; Hartmann, B. Bifröst: Visualizing and checking behavior of embedded systems across hardware and software. In Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology, Québec City, QC, Canada, 22–25 October 2017; pp. 299–310.
7. Dias, D.; Lima, G.; Barros, E. A Cache Design Assessment Approach for Embedded Real-time Systems Based on Execution Time Measurement. In Proceedings of the 2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC), Joao Pessoa, Brazil, 1–4 November 2016; pp. 168–173.
8. Decker, N.; Dreyer, B.; Gottschling, P.; Hochberger, C.; Lange, A.; Leucker, M.; Scheffel, T.; Wegener, S.; Weiss, A. Online analysis of debug trace data for embedded systems. In Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 19–23 March 2018; pp. 851–856.
9. Dreyer, B.; Hochberger, C.; Lange, A.; Wegener, S.; Weiss, A. Continuous non-intrusive hybrid WCET estimation using waypoint graphs. In Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016), Toulouse, France, 5 July 2016.
10. Betts, A.; Merriam, N.; Bernat, G. Hybrid measurement-based WCET analysis at the source level using object-level traces. In Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010), Brussels, Belgium, 6 July 2010.
11. Wilhelm, R.; Engblom, J.; Ermedahl, A.; Holsti, N.; Thesing, S.; Whalley, D.; Bernat, G.; Ferdinand, C.; Heckmann, R.; Mitra, T.; et al. The worst-case execution-time problem—Overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* **2008**, *7*, 1–53. [CrossRef]
12. Liu, J.W.S. *Real-Time Systems*; Prentice Hall: Upper Saddle River, NJ, USA, 2000.
13. Sha, L.; årzén, K.-E.; Abdelzaher, T.; Cervin, A.; Baker, T.; Burns, A.; Buttazzo, G.; Caccamo, M.; Lehoczky, J.; Mok, A.K. Real time scheduling theory: A historical perspective. *Real-Time Syst.* **2004**, *28*, 101–155. [CrossRef]
14. Betts, A. Hybrid Measurement-Based WCET Analysis Using Instrumentation Point Graphs. Ph.D. Thesis, University of York, York, UK, February 2010.
15. Gait, J. A probe effect in concurrent programs. *Softw. Pract. Exp.* **1986**, *16*, 225–233. [CrossRef]
16. Rodriguez Polo, O.; Martinez Hellin, A.; Parra Espada, P.; Sanchez Prieto, S.; Da Silva Fariña, A. A Method and a Parallel Processing Device of Program and Trace Instructions. ES 2 697 548 A1, January 2019. Available online: https://patents.google.com/patent/ES2697548A1/en (accessed on 6 November 2020).
17. RISC-V International. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*; Document Version 20191213; CS Division, EECS Department, University of California: Berkeley, CA, USA, 2019.
18. RISC-V International. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*; Document Version 20190608-Priv-MSU-Ratified; CS Division, EECS Department, University of California: Berkeley, CA, USA, 2019.
19. Eckert, R.J.; Peters, J.; Tam, H.Y.; Solorzano, A. Systems and Methods for a Real Time Embedded Trace. U.S. Patent Application 14/945,815, 25 May 2017.
20. Vahid, F.; Givargis, T. *Embedded System Design: A Unified Hardware/Software Introduction*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2001.

21. Di Mascio, S.; Menicucci, A.; Gill, E.; Furano, G.; Monteleone, C. Leveraging the Openness and Modularity of RISC-V in Space. *J. Aerosp. Inf. Syst.* **2019**, *16*, 454–472. [CrossRef]
22. Di Mascio, S.; Menicucci, A.; Furano, G.; Monteleone, C.; Ottavi, M. The Case for RISC-V in Space. In *Applications in Electronics Pervading Industry, Environment and Society*; Springer International Publishing: Cham, Switzerland, 2019; pp. 319–325, [CrossRef]
23. Aranda, L.A.; Wessman, N.J.; Santos, L.; Sánchez-Macián, A.; Andersson, J.; Weigand, R.; Maestro, J.A. Analysis of the Critical Bits of a RISC-V Processor Implemented in an SRAM-Based FPGA for Space Applications. *Electronics* **2020**, *9*, 175, [CrossRef]
24. Reorda, M.S.; Violante, M.; Meinhardt, C.; Reis, R. An on-board data-handling computer for deep-space exploration built using commercial-off-the-shelf SRAM-based FPGAs. In Proceedings of the 2009 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Chicago, IL, USA, 7–9 October 2009; pp. 254–262.
25. Siegle, F.; Vladimirova, T.; Ilstad, J.; Emam, O. Availability analysis for satellite data processing systems based on SRAM FPGAs. *IEEE Trans. Aerosp. Electron. Syst.* **2016**, *52*, 977–989. [CrossRef]
26. Furano, G.; Menicucci, A. Roadmap for On-Board Processing and Data Handling Systems in Space. In *Dependable Multicore Architectures at Nanoscale*; Springer International Publishing: Cham, Switzerland, 2018; pp. 253–281, [CrossRef]
27. Brandolese, C.; Corbetta, S.; Fornaciari, W. Software energy estimation based on statistical characterization of intermediate compilation code. In Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design, Fukuoka, Japan, 1–3 August 2011; pp. 333–338.
28. Von Mayrhauser, A.; Malaiya, Y.K.; Srimani, P.K.; Keables, J. On the need for simulation for better characterization of software reliability. In Proceedings of the 1993 IEEE International Symposium on Software Reliability Engineering, Denver, CO, USA, 3–6 November 1993; pp. 264–273.
29. Rodd, M.; Motus, L. *Timing Analysis of Real-Time Software*; Elsevier: Amsterdam, The Netherlands, 1994.
30. Puschner, P.; Koza, C. Calculating the maximum execution time of real-time programs. *Real-Time Syst.* **1989**, *1*, 159–176. [CrossRef]
31. Tracey, N. Engineering real-time behavior. *IEEE Instrum. Meas. Mag.* **2002**, *5*, 29–38. [CrossRef]
32. Petters, S.M.; Farber, G. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications, RTCSA'99 (Cat. No. PR00306), Hong Kong, China, 13–15 December 1999; pp. 442–449.
33. Ferdinand, C.; Heckmann, R. Ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 377–383.
34. Axer, P.; Ernst, R.; Falk, H.; Girault, A.; Grund, D.; Guan, N.; Jonsson, B.; Marwedel, P.; Reineke, J.; Rochange, C.; et al. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.* **2014**, *13*, 1–37. [CrossRef]
35. Thiele, L.; Wilhelm, R. Design for timing predictability. *Real-Time Syst.* **2004**, *28*, 157–177. [CrossRef]
36. Yan, J.; Zhang, W. A time-predictable VLIW processor and its compiler support. *Real-Time Syst.* **2008**, *38*, 67–84. [CrossRef]
37. Schoeberl, M.; Puffitsch, W.; Hepp, S.; Huber, B.; Prokesch, D. Patmos: A time-predictable microprocessor. *Real-Time Syst.* **2018**, *54*, 389–423. [CrossRef]
38. Colin, A.; Puaut, I. Worst case execution time analysis for a processor with branch prediction. *Real-Time Syst.* **2000**, *18*, 249–274. [CrossRef]
39. Kirner, R.; Wenzel, I.; Rieder, B.; Puschner, P. Using measurements as a complement to static worst-case execution time analysis. *Intell. Syst. Serv. Mank.* **2005**, *2*, 8.
40. Wright, D.R.S. WCET Analysis of Object Code with Zero Instrumentation. 2017. Available online: https://www.rapitasystems.com/blog/wcet-analysis-object-code-zero-instrumentation (accessed on 6 November 2020).
41. Kästner, D.; Pister, M.; Wegener, S.; Ferdinand, C. TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis. In Proceedings of the 19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019), Stuttgart, Germany, 9 July 2019.
42. IEEE-ISTO. *IEEE-ISTO 5001-2012, The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface*; 445 Hoes Lan; IEEE- Industry Standards and Technology Organization: Piscataway, NJ, USA, 2012.

43. Rapita Ltd, R.S. RapiTime. 2020. https://www.rapitasystems.com/products/rapitime (accessed on 6 November 2020).
44. Whalen, M.W.; Rajan, A.; Heimdahl, M.P.; Miller, S.P. Coverage metrics for requirements-based testing. In Proceedings of the 2006 International Symposium on Software Testing and Analysis, Portland, ME, USA, 17–20 July 2006; pp. 25–36.
45. Carrillo-Mondéjar, J.; Castelo-Gómez, J.; Roldán-Gómez, J.; Martínez, J. An instrumentation based algorithm for stack overflow detection. *J. Comput. Virol. Hacking Tech.* **2020**, *16*, 245–256. [CrossRef]
46. Jouppi, N.P.; Wall, D.W. Available instruction-level parallelism for superscalar and superpipelined machines. *ACM SIGARCH Comput. Archit. News* **1989**, *17*, 272–282. [CrossRef]
47. Johnson, W.M. *Super-Scalar Processor Design*; Stanford University: Stanford, CA, USA, 1989.
48. Wenzel, I.; Kirner, R.; Puschner, P.; Rieder, B. Principles of timing anomalies in superscalar processors. In Proceedings of the Fifth International Conference on Quality Software (QSIC'05), Melbourne, Australia, 19–20 September 2005; pp. 295–303, [CrossRef]
49. Taştan, I.; Karaca, M.; Yurdakul, A. Approximate CPU Design for IoT End-Devices with Learning Capabilities. *Electronics* **2020**, *9*, 125, [CrossRef]
50. Wu, N.; Jiang, T.; Zhang, L.; Zhou, F.; Ge, F. A Reconfigurable Convolutional Neural Network-Accelerated Coprocessor Based on RISC-V Instruction Set. *Electronics* **2020**, *9*, 1005, [CrossRef]
51. Xilinx. Available online: https://www.xilinx.com/ (accessed on 27 July 2020).
52. Xilinx. 7 Series FPGAs Data Sheet: Overview. 2018. Available online: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (accessed on 6 November 2020).
53. Patterson, D.A. *Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition/David A. Patterson, John L. Hennessy*; Computer Systems Design; Computer Hardware, Morgan Kaufmann: Cambridge, MA, USA, 2018.
54. Ionescu, L. The xPack GNU RISC-V Embedded GCC. Available online: https://xpack.github.io/riscv-none-embed-gcc/ (accessed on 27 July 2020).
55. Rapita Systems Ltd. Available online: https://www.rapitasystems.com/ (accessed on 28 July 2020).
56. Rodríguez-Pacheco, J.; Wimmer-Schweingruber, R.F.; Mason, G.M.; Ho, G.C.; Sánchez-Prieto, S.; Prieto, M.; Martín, C.; Seifert, H.; Andrews, G.B.; Kulkarni, S.R.; et al. The Energetic Particle Detector—Energetic particle instrument suite for the Solar Orbiter mission. *Astron. Astrophys.* **2020**, *642*, A7. [CrossRef]
57. Sánchez, S.; Prieto, M.; Polo, Ó.R.; Parra, P.; da Silva, A.; Gutiérrez, Ó.; Castillo, R.; Fernández, J.; Rodríguez-Pacheco, J. HW/SW co-design of the instrument control unit for the energetic particle detector on-board solar orbiter. *Adv. Space Res.* **2013**, *52*, 989–1007. [CrossRef]
58. Heinz, C.; Lavan, Y.; Hofmann, J.; Koch, A. A Catalog and In-Hardware Evaluation of Open-Source Drop-In Compatible RISC-V Softcore Processors. In Proceedings of the 2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Cancun, Mexico, 9–11 December 2019; pp. 1–8.
59. Gookyi, D.A.N.; Ryoo, K. Selecting a Synthesizable RISC-V Processor Core for Low-cost Hardware Devices. *J. Inf. Process. Syst.* **2019**, *15*, 1406–1421.
60. Höller, R.; Haselberger, D.; Ballek, D.; Rössler, P.; Krapfenbauer, M.; Linauer, M. Open-Source RISC-V Processor IP Cores for FPGAs—Overview and Evaluation. In Proceedings of the 2019 8th Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 10–14 June 2019; pp. 1–6.