Improving Stress Quality for SoC Using Faster-than-At-Speed Execution of Functional Programs

Paolo Bernardi, Alberto Bosio, Giorgio Di Natale, Andrea Guerriero, Ernesto Sanchez, Federico Venini

Improving stress quality for SoC using Faster-than-At-Speed execution of functional programs

Paolo Bernardi (1), Alberto Bosio (2), Giorgio Di Natale (2),
Andrea Guerriero (1), Ernesto Sanchez (1), Federico Venini (1)

(1) Politecnico di Torino, Corso Duca degli Abbruzzi 24, Torino, 10129, Italy
(2) LIRMM, Rue Ada 161, Montpellier, 34095, France

{paolo.bernardi@polito.it, bosio@lirmm.fr, dinatale@lirmm.fr,
andrea_guerriero@polito.it, ernesto.sanchez@polito.it,
federico.venini@polito.it}

Abstract. At the end of the manufacturing cycle of digital circuits, a stress phase is mandatory in order to remove from the final device population the weak devices that may result in early life failures. Devices used in safety critical environments must undergo this phase that is usually accomplished by exploiting the Burn-In (BI) process. Unfortunately, BI has elevated costs for companies and current state of the art techniques are trying to reduce its cost.
In recent days, Faster-than-at-Speed-Test (FAST) has become a useful technique to discover small delay defects. At the same time, overclocking methods to enhance system performances have been studied, which focus on temperature management to preserve system functionalities. In this contribution, a FAST technique is proposed with the aim of intentionally provoking a thermal overheating in the microprocessor by mean of the execution of FAST functional test programs; in other words, functional procedures are executed at higher than nominal frequencies. The goal is to introduce an internal stress stronger than current procedures used during BI in order to speed up early detection of latent faults.
Being the functional stress procedures executed at faster than nominal speed, the original behaviour may not be preserved and therefore non-functional states may be reached. In this contribution, it is illustrated how to avoid blocking configurations due to timing constraints violation and how to obtain a significant increase of the switching activity by carefully increasing the clock frequency. Furthermore, a novel strategy is proposed to generate a suitable set of Faster-than-At-Speed stress programs capable to thoroughly stress processor cores.
Experimental results carried out on a MIPS-like architecture show major achievements of the methodology: the processor may work at frequencies up to about 20 times higher than the nominal one without falling into an unpredictable state and the switching activity is increasing up to 300% per nanoseconds.

Keywords: Faster-than-At-Speed-Test, Functional Test, Stress Test, Switching Activity, Evolutionary Algorithm.

1 Introduction

The reduced scale, predicted years ago by the Moore's law, and the low costs of Integrated Circuits (IC) have been the principal factors of spreading high performance microprocessor devices in mass products and safety critical applications. At the same time, the increasing density of transistors on System-on-Chip (SoC), as a subsequence of their smaller size, and the high frequencies they work, have leaded these products to vulnerability to faults and defects, not only during manufacturing processes but also along with the device lifetime.

To guarantee circuit robustness, these products are subjected to several stress phases aiming at bringing the circuit to stressful conditions oriented to detect and discard weak devices. In particular, the so-called Burn-In (BI) test process aims at producing a degradation similar to that introduced by the normal operation during a long period of time (i.e., aging). It applies an external thermal overheating to the device, that in some cases also includes the application of supply voltages higher than the nominal ones. Thermal-based accelerated aging is produced in order to exacerbate time-related degradation defects such as electro-migration affecting the circuit metal lines, leakage current increasing, and variation of threshold voltage in the gates belonging to the design [1]. BI process is necessary for those devices that have to carry out safety critical applications, and for non-repairable systems. By applying a Burn-In phase at the end of manufacturing, it is expected that fewer defective components are delivered and the consequent reduction in failure rates lowers production costs [2]. However, BI is usually associated with long test time and costs, aspects that make it a bottleneck of the entire IC manufacturing process.

In the last years, some approaches tried to apply stress patterns resorting to functional approaches. Functional test programs verify the system integrity for functional and performance-related specifications. It may test all or most of the system functionalities along with the availability of subsystems. In [3] and [1], for example, the authors demonstrated that it is possible to apply electrical stress on a given device resorting to functional-based stress programs.

It is clear that a relationship exists between the circuit switching activity (SA) and the temperature developed on the surface of the die [1]. As a matter of fact, it is reasonable to assume the possibility to induce a thermal-aware stress by modulating the switching activity of the target device.

In this contribution, we investigate the possibility of running functional programs at a frequency higher than at-speed with the intention of increasing the switching activity of the system. Once the research claims that processor can continue working even at higher frequencies than nominal thanks to system architecture features, a generic methodology to create functional stress programs targeting pipelined processor cores is investigated. The introduced approach presents the rules to estimate a set of frequency ranges in which the processor is not falling into a blocking state and, by means of these rules, it is possible to generate meaningful functional stress programs. Initially programs are generated targeting every relevant module. Then, some positive side effects on the rest of the modules are measured and a new generation campaign is performed on the missing modules. This process iterates until satisfactory results are reached.

In order to guide the generation process, the method computes a set of elements that helps to assess the quality of each stress functional program in terms of spatiality (i.e., how well the switching activity is spread across the die surface), and intensity (i.e., how high the switching activity is among a given number of gates). Interestingly, the proposed approach is also able to automatically setup the processor frequency in order to find the most appropriate faster-than-at-speed stress programs. This generation feature allows the creation of functional stress programs operating at frequency ranges higher than the nominal ones, avoiding to incur in a blocking state of the processor. The chapter finally presents valid ranges of faster-than-at-speed frequencies in terms of switching activity increase for a functional program, and then it provides a comparison between at-speed and faster-than-at-speed functional stress programs automatically generated.

The rest of the chapter is organized as follows: Section 2 provides the framework background, Section 3 describes the proposed approach both for discriminate the processor reaction to FAST, both for describe the automatic stress program generation process, while Section 4 and 5 introduce the case of study and show the experimental results carried out on a MIPS-like processor core synthetized with an industrial library. Finally, Section 6 concludes the chapter drawing some future works.

2 Background

Regarding new stress techniques, recent studies are beginning to consider the execution of patterns with a clock frequency higher than the nominal at-speed. This technique, known as faster-than-at-speed test (FAST), finds satisfying results when applied for detecting small delay defects (SDD) which are not easy to screen out with at-speed test [4], [5].

Several application methods targeting FAST technique have been proposed so far in the literature: external applications [5-6], which are not trivial due to skews and other physical effects that could affect the measurements, therefore, this method requires a more expensive automatic test equipment (ATE). On the other hand, built-in FAST using programmable on-chip clock generation is comfortable with these issues [6-7], as frequency increase is functionally obtained (e.g., executing assembly instructions). Current works concentrate their efforts on classifying faults according to the optimal frequency they can be tested, or on finding an optimized selection of clock frequencies [4], [8]; usually the maximum clock rate value may be up to 3 times higher than the nominal clock rate [4], [9-10].

Considering the generation of functional stress patterns for processor cores, an interesting solution can be based on the so-called Software-Based Self-Test (SBST) techniques. The main idea on SBST is to create a functional program intended to detect the processor faults [11]. Roughly speaking, the processor core executes a program allocated in an available memory zone. The computed results are evaluated in order to determine whether the processor is faulty or not. Interestingly, the processor operates in normal mode, thus, there are no requirements for any hardware modification. Nevertheless, SBST methodologies have a limited application in industry due to the difficulty

to write efficient and effective test programs and to devise suitable methodologies for test application.

To the best of our knowledge, this study is the first approach to Faster-Than-at-Speed Test technique while functional programs are running in a SoC. In this contribution, the objective does not concern the coverage of SDD, in contrast with the works listed before, but rather the switching activity increase, which is useful for stress purposes.

3 Proposed approach

The main objective of this work is to provide detailed understanding about CPU frequency overclocking of a SoC targeting switching activity increase. As already stated in the introduction, increasing the switching activity is beneficial in terms of stress capability and thermal objectives in test. The basic idea of our approach is to analyse the circuit transitions when a functional test program is running, spanning several clock frequencies starting from the nominal one, until reaching very high values. For this reason, in an overclocked situation, it is crucial to understand if:

- The microprocessor can produce *stable* results and keep some functionalities, although some design timing constraints are violated. In this case, we expect a dramatic increase of the internal temperature, thus a further acceleration of aging. Later, this situation is named *functional state.*
- The microprocessor is *stuck* to some unpredictable state, similarly to a "forced functional idle", that leads in a situation in which there will be no gain in terms of temperature exacerbation. Later, this situation is called *unstable state.*

Functional state is highly desirable, while unstable state need to be avoided, in case the induction of a strong stress is the objective [12].

In the following paragraphs, an analysis flow is proposed aiming at determining microprocessor behaviour, functional or unstable, at frequencies higher than the nominal one. Thanks to this methodology, it is possible to identify the causes of the *unstable state*, which will be discussed in details; specific cases and processor configurations need to be avoided in order to maintain a functional state even at very high frequencies.

Once this analysis is complete, purpose becomes to provide an automatic technique able to generate a suitable set of stress programs for processor cores. In fact, the approach creates faster-than-at-speed functional test programs maximizing the switching activity (SA) for the whole processor core. One of the most relevant drawbacks of SBST-based strategies is related to the high amount of resources required to generate test programs. As detailed in [11], most of the generation processes are time consuming and/or require expert engineers able to write the test programs. Thus, the proposed strategy exploits the possibility to generate stress programs for different processor modules in parallel, taking advantages of the positive side-effects that may appear on different processor modules when targeting a particular one. The proposed methodology will be discussed in the following subsections.

3.1 Analysis flow for understanding SoC behaviour

The first objective of our work is to understand when the processor under analysis is falling in an unstable state or not at a certain frequency. To achieve this important understanding, a tool chain setup was devised, enabling the classification of several behaviours.

When overclocking and setting up a clock period tighter than the nominal, unknown (*X*) values can appear in the simulation. These values are caused by the violation of timing constraints of Flip-Flops (FFs) with respect to critical path timing. Figure 1 illustrates an explanatory scenario where the SoC is entering into an unstable state.

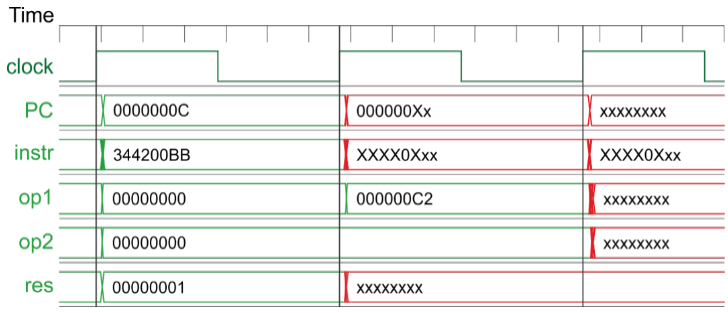


Fig. 1. Some of the CPU registers at the beginning of *unstable state*.

In some cases, the unknown values are bounded in a set of FFs, possibly not compromising the execution of the program. On the contrary, if the unknown values affect a huge number of FFs, CPU reaches an unpredictable state.

The consequences are showed in Figure 1: when timing constraints of FFs are violated, registers will contain unknown values which can quickly propagate through others (the propagation will depend on the hardware connection between modules). In the showed example, Program Counter (PC) has been affected. Therefore, CPU will never move to execute the next instruction, and it will be stuck into an idle-like state. The affection of instruction-to-decode register (*instr*), or others, has similar effects.

Figure 2 visualizes the practical cause of the X values generation. If a faster-than-at-speed clock is used, the propagation along path may not complete even without a delay fault affecting the circuit. In this case, FFs will not sample a stable value, resulting in an unknown output.

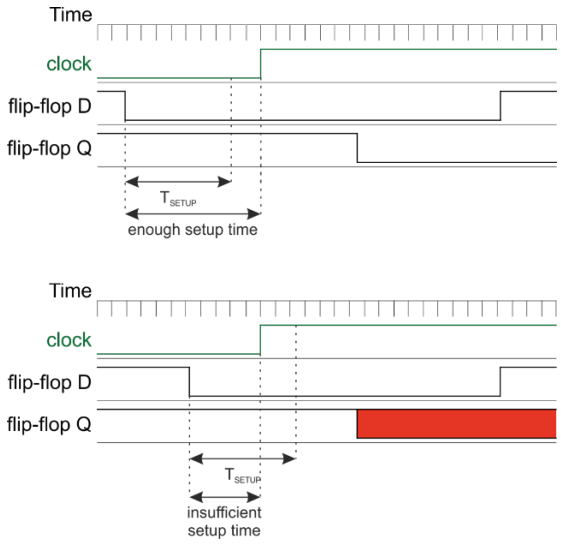


Fig. 2. Violation of setup time of FF (in the lower graph), compared to a good sampling.

Figure 3 depicts the analysis flow to detect unstable behaviour, which was devised and implemented as a tool chain including both commercial EDA tools and ad-hoc tools. The main idea is to look at the simulation and figure out whether the simulation converges on an unstable state. Furthermore, a loop is implemented where the frequency is increased every time the current analysis is concluded. The logic simulator is fed with critical paths that are extracted by a Static Timing Analysis (STA) tool. The simulator is required to dump the state of the critical paths along the simulation time.

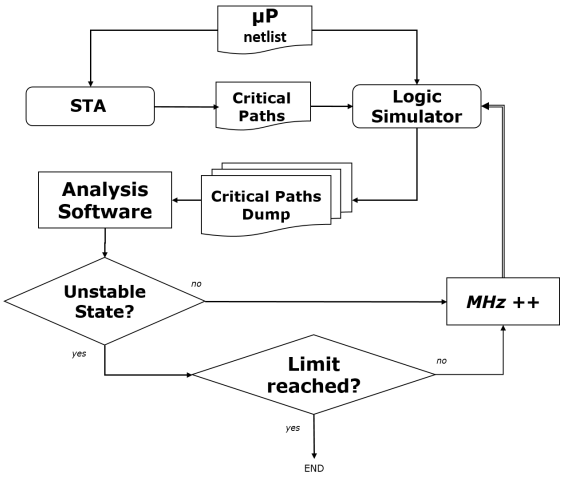


Fig. 3. Proposed analysis workflow for detecting *unstable states*.

This selective simulation dump is finally provided to an ad-hoc tool able to return the following information from the simulation dump:

- The program *execution time*, by simply counting the number of clock cycles resulted in the simulation;
- The *switching activity* computed as the number of transitions resulted in the gates normalized per ns.

Execution time is enough to identify an unstable state of the processor. In the simulation setup, a timeout time is set and several traps are implemented in order to stop the simulation and classify it as functional; these termination conditions are showing specific "footprints" left by the processor when correctly running a program. As far as we saw, when the gates of the processor start to show unknown behaviours it is highly improbable that it can reach one of the termination conditions. In this case, test program will be forced to terminate by the timeout time set by the simulator.
In case the simulation does not fall in an unstable state, the process is repeated at a higher frequency. The implemented toolchain permits to decide which step to use along the loop, and automatically adjust sampling time according to the new frequency. In case the simulation falls in the unstable state, a simple formula, detailed in next subsection, permits to decide if continue in the frequency increase or stop the iterations.

3.2 Processor level analysis and classification of functional states

The common perception is that if a circuit is supplied with a frequency higher than nominal, it will not work. This is an incomplete analysis of the problem as falling in the unstable state may be prevented by some processor architectural features. The first one is the mechanism of pipeline stalls and memory wait-states. They are needed as the memory cores in SoC are usually accessed asynchronously. As a matter of fact, as system stalls because the memory is slow and asks for more time, signals travelling even through the longest paths can reach stable values. Hence, even if another clock period is started, stalled units will not ask for values until the reactivation of pipeline. Figure 4 illustrates the situation: the pipeline is stalled by *stop_all* signal in case the *ram_ready* from the memory goes low too late and it is not sampled. The *ram_ready* signal falling time depends on the latency of the memory. In this example, *ram_ready* is correctly sampled after few clock cycles of pipeline stall as the latency of the memory was set to 5ns; as well this is feasible because the latency is larger than the propagation time through the most critical path, 4.5 ns in the shown example. The result of the previous ALU operation, such as stored in *EX_data* register, will be read only after the reactivation of CPU pipeline, certainly showing correct values.

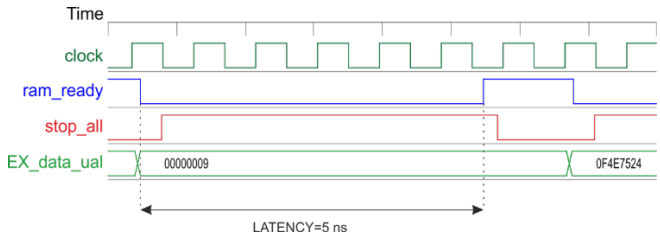


Fig. 4. Typical pipeline stalls during memory latency.

As experimental results proven, this is a lucky configuration that happens only within some ranges of clock frequency. A general very strong rule that can be deduced by the proposed analysis is the following: If the clock period is approximately equal or less than latency, and the latency is larger than the propagation time of functional critical paths, the unstable state depends only on propagation times of asynchronous data-paths. Usually, these data-paths belong to processor boundary (such paths interfacing RAM signals to CPU, such as *ram_ready*, again) or internal CPU units (such as Prediction Unit), which are not synchronized with the pipeline. In most of architectures, propagation times of asynchronous data-paths are usually less than for processor logic synchronously controlled by the clock. As a consequence, asynchronous data-paths may sustain a certain increase of the clock frequency without incurring into unpleasable events. But, as we are pulling up the frequency much higher than nominal, asynchronous data-paths take a significant role in the study. Figure 5 graphically visualizes the concept of asynchronous data-path; two situations can be identified, with incoming and out-coming paths.

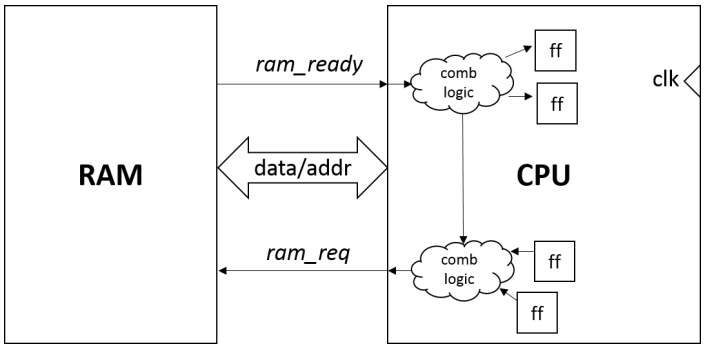


Fig. 5. Example of connections between CPU and companion modules such as memory cores.

Let us consider the *ram_ready* and *ram_req* signals as a reference to clarify the concept. Figure 6 details the acknowledge protocol and relevant times. Once the RAM is rising the *ram_ready* signal, the *ram_req* is falling and again rising after a fixed time T_{REQ} elapsed from the current clock rising edge. Thus, *ram_ready* is falling after a fixed time $T_{\overline{READY}}$, corresponding to the beginning of the next latency time. The time from the falling transition of *ram_ready* to the first clock rising after sampling is called

T_{FIRST_RISE}. As well, the time passing from the last rising of the clock before the *ram_ready* is rising is called T_{LAST_RISE}. Most importantly, the time passing from *ram_ready* rising and clock rising edges, namely T_{RES}, is key to understand whether the processor will fall into an unstable state or not. This time should be large enough both to permit propagation along the asynchronous incoming data paths, namely $T_{ASYNCRONOUS\_PROPAGATION}$, and to afford a signal stability time long enough to respect the setup time T_{SETUP}, as preliminarily described in this section. This scenario is depicted in Figure 7, where T_{DES} is the minimum amount of time that guarantee a correct behavior; in opposition, T_{RES} is the real measurement of time elapsed from the *ram_ready* rising to the following rising edge of the clock signal. T_{DES} can be estimated by simply analyzing the circuital configuration in order to identify propagation cones of asynchronous data paths:

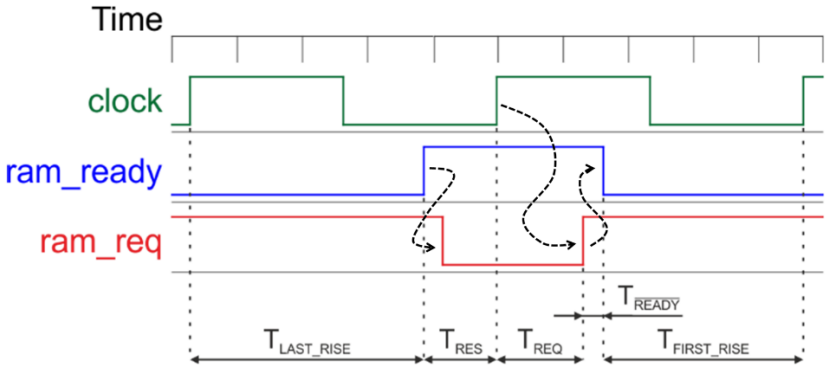


Fig. 6. BUS timing diagram with *ram_ready* and *ram_req*.

$$T_{DES} = T_{ASYNCHRONOUS\_PROPAGATION} + T_{SETUP} \quad (1)$$

where $T_{ASYNCHRONOUS\ PROPAGATION} >> T_{SETUP}$.

T_{RES} can be calculated by adding information related the selected frequency; period in formulas refers to the clock period. To obtain T_{RES}, T_{FIRST_RISE} and T_{LAST_RISE} are needed.

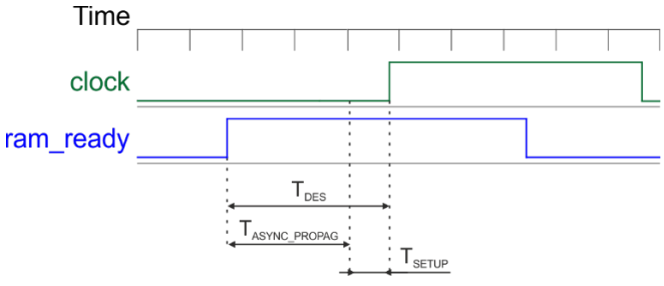


Fig. 7. Timing diagram showing stability requirements for *ram_ready* correct sampling.

$$T_{FIRST\ RISE} = period - \left(T_{REQ} + T_{\overline{READY}}\right) \quad (2)$$

$$T_{LAST\ RISE} = (latency\ -\ T_{FIRST\ RISE})\ \%\ period \quad (3)$$

$$T_{RES} = period - T_{LAST\ RISE} \quad (4)$$

At this point, with T_{RES} calculated, it is possible to compare it with T_{DES} for an estimation of the system behavior under higher than at speed frequencies:

$$if\quad T_{RES} > T_{DES}\ \rightarrow\ FUNCTIONAL\ STATE \quad (5)$$

$$if\quad T_{RES} < T_{DES}$$

$$T_{RES} > T_{ASYNCHRONOUS\ PROPAGATION}\ \rightarrow\ UNSTABLE\ STATE \quad (6)$$

$$if\quad T_{RES} < T_{DES}$$

$$T_{RES} < T_{ASYNCHRONOUS\ PROPAGATION}\ \rightarrow\ FUNCTIONAL\ STATE \quad (7)$$

Let us observe some examples. In the first case, Figure 8.A, the *ram_ready* transition arrives with a sufficient time to permit propagation and setup, and program counter is not affected Program Counter (PC) register. Frequency there is quite high, 156 MHz. On the contrary, according to (6) Figure 8.B shows a wrong sampling scenario, caused by higher frequency, 704 MHz. The rising edge of clock is arriving enough early to have propagation of the asynchronous signals up to FFs, but not enough for setup. The wrong sampling will affect PC register since the next clock cycle.

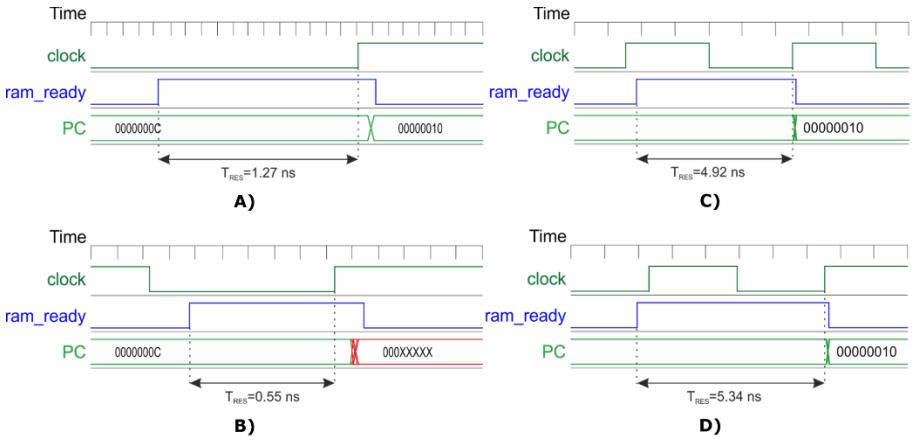


Fig. 8. Dependency on asynchronous path.

Figure 8.C permits a simple understanding of the reason why the processor stops working for a range of addresses and resume for higher frequencies. The clock is quite high in this example, 190 MHz. Even if the clock period is comparable with T_{DES}, equation (5) can be satisfied if the *ram_ready* signal is in a favorable phase with clock, resulting in enough T_{RES}.

Last of the cases, Figure 8.D shows that the period can be even smaller than T_{DES}. In this case, the frequency is 198MHz and the specific case (7) is satisfied because from *ram_ready* to clock rising there is not enough time even to complete the propagation of asynchronous paths to the FFs. The old value is captured and the true value is latched at the next clock rising edge. This is essentially similar to have a latency overhead of 1 clock cycle, but not compromising the system functionalities.
In the analysis script-chain, the limit of clock frequency value is reached when:

$$Period < T_{ASYNCHRONOUS\ PROPAGATION} + T_{SETUP} \quad (8)$$

3.3 Proposed methodology for generating stress programs

Figure 9 sketches the main steps for automatically generate functional stress programs targeting processor cores. The first step of generation consists on the *Processor division*. The idea here is to apply a *divide et impera* strategy that allows the next generation steps to be executed in parallel. Such a division is performed following the internal processor hierarchy. Then, the processor is divided into its more important submodules that would be tackled in the following steps in a separate way.

It is important to consider that, since the main goal during the generation process is to obtain stress programs maximizing the SA on every processor module, the processor division step must produce not only a hierarchical modules division, but also should be able to distribute all the processor gates on every one of the modules belonging to the processor core. Finally, the most relevant instructions belonging to the processor Instruction Set Architecture (ISA) must be correlated with the derived processor submodules.

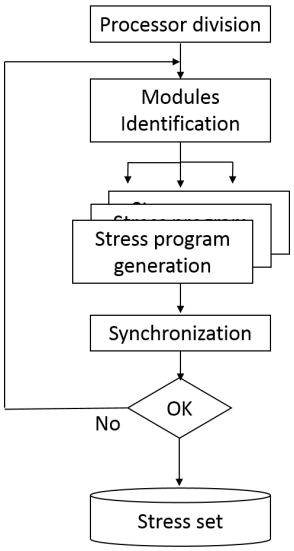


Fig. 9. Stress program generation flow diagram.

The second step is called *Module Identification and setup*. This is the first step composing the methodology iterative loop. Here the processor sub-modules to be targeted during each iteration are selected. The idea is to identify and select at the very beginning the functional units of the processor, then, in a successive step, the control parts. As mentioned before, the previous step provides not only the structural division of the processor core but also the set of possible instructions that are able to excite every one of the processor sub-modules. The idea behind this consideration is that the processor functional units are usually independent on each other. It means that each of them performs a task that does not require the intervention of other functional modules. For example, we can consider the processor arithmetic module and the multiplier. Then, a stress program may try to thoroughly stress one specific module without targeting a different one at the same time. On the contrary, most of the processor modules related to the control are highly correlated to the rest of the processor. Thus, these modules should be tackled in a successive step in order to take advantages of the positive effects produced by the stress programs of the already targeted modules. The *Module Identification and setup* step is also in charge to check the list of gates belonging to the considered module in order to discard from the list the gates that at this point are satisfactory stressed. Clearly, during the first iteration no one gate is discarded.

Once the different modules to be tackled for the current iteration have been identified, the generation process can be parallelized. This step is called *Stress program generation* and can use different strategies, for example deterministic, manual, formal-based, and evolutionary-based strategies [11]. A suitable stress program is characterized by the high switching activity on the targeted module involved gates. Even though the SA is one of the most relevant, some other parameters require to be considered. In the proposed case, we define a parameter related to the stress quality for every gate (G_{SQ}), that is given by:

$$G_{SQ} = SA \cdot \frac{1}{d} \cdot Th \cdot g_{CH} \tag{9}$$

where SA is the actual gate switching activity, d represents the physical distance of the gate to the centre of the module, considering the actual placement of the gate in the final device, Th is the thermal factor related to the chip structure, and g_{ch} is a parameter related only to the gate electrical characteristics (fan out, fan in). Please note that all the parameters except the SA are constants, and are related to the gate type and position in the considered module. Thus, during our experiments the target is to increase the SA on every gate.

During the generation process, it is important to provide the selected tool with all the module information required to carry out the generation process. In particular, the tool receives the module gates, as well as the instructions exciting the module itself. In our proposal, we used an evolutionary optimizer called μGP (MicroGP). The tool is developed by Politecnico di Torino since 2000, and is freely available under the GNU Public License [13]. μGP is able to generate syntactically correct assembly programs by acquiring information about the ISA of the processor, and in particular about every one of the modules under consideration from a user-defined file called *Constraint Library*. It is important to highlight that the evolutionary optimizer defines not only the different instructions composing every program but also the operation frequency. This provides the generation process to create programs that run faster than the nominal frequencies.

The faster-than-at-speed execution of these functional test programs permits to reach a grade of stress quality of the circuit in a shorter time interval with respect to the at-speed execution of same test programs. However, as shown in [11], due to frequencies higher than the nominal ones, the processor core may fall in an unstable state. Then, during the test program evaluation, it is important to assure that regardless the operating frequency, the processor behavior is still stable.

When the generation process starts, the evolutionary tool generates an initial set of random assembly programs (called population), or *individuals*, exploiting the information provided by the library of constraint. Every individual also provides the operation frequency as a parameter defined by the evolutionary core. Then, these stress programs are cultivated in order to be improved following the Darwinian concepts of natural evolution. Every individual is evaluated resorting to external tools that simulate the processor by running the stress program at the given frequency and information about the SA in the consider module is gathered. Additionally, processor stability is also considered. In the performed experiments, we compute the average of the switching activity on all the gates composing the module under evaluation. This value is provided to the evolutionary tool as an indicator of the stress program goodness, this is usually called *fitness value*. Once all the individuals are evaluated, resorting to the fitness values, the individuals are ordered and the best ones are maintained in the population, while the others are discharged from the population. Then, a new evolutionary step starts again trying to improve the remaining individuals in the population.

The last step belonging to the iteration process consists on the *synchronization* of the stress programs obtained until this point. As mentioned before, there are independent generation processes for every processor module, however, until now there are no information regarding the impact of the generated programs in the rest of the processor core. Then, it is important to perform a new simulation that involves the full processor core during the execution of all the programs obtained in the current iteration. At the end of this step, a complete figure about the stress capacity of the temporary set of programs in all the processor modules is obtained. Then, the obtained values are compared to the global satisfactory stress ones, and if the results are not in the satisfactory range, the iteration starts again. Otherwise, the process terminates providing the user with the final set of stress programs.

4 Experimental Results

To demonstrate the feasibility and effectiveness of the proposed methodology, two different experiments have been carried out on a *MIPS-like* core [9] synthesized with ST-corelibrary for 65 nm CPU. After the synthesis, the netlist is composed by 50,000 gates circa, which 3,000 are FFs. The *MIPS-like* processor is based on MIPS architecture [10] and contains a 5-stages pipeline which includes hazard detection and pipeline interlock, branch prediction unit and system co-processor. The processor ISA is composed of 52 instructions, including arithmetic, logic, load and store, and branch related ones. The nominal frequency for normal mode operation is 100 MHz.

This section provides two sets of experimental results: the first one is related to the analysis of the execution of generic FAST functional programs, the second one is related to the automatic generation of FAST functional stress programs.

4.1 Execution of functional programs

By mean of scripts, it has been possible to iterate the execution of a functional program increasing CPU clock frequency by 0,5 MHz for each iteration. *ModelSim* [13] and *Primetime* [14] have been the commercial software used for logic simulation and extraction of critical paths respectively. The delays of single gates are listed in the Standard Delay Format (SDF) file. RAM has been written in System-C language. The software proposed collects data into CSV files for each clock frequency tested.

Several types of functional test programs have been experimented with this framework. Here, we report two cases of programs that are significant in our view:

- Oriented to arithmetic units;
- Stressing branch conditions.

In Figure 10a and 10b, execution time is shown at growing frequencies. Blue points correspond to functional states, red to unstable states. It can be noticed that execution time regrows in some cases, passing from unstable to functional state, as the effect of the additional waiting clock cycles inserted as described in 3.1, Formula (7). Functional states are identified by means of simulation, as in the loop proposed in section 3.1; the processor is expected to store, in a specific RAM address, the signature computed along program execution. The simulation environment is stopped as soon as both these conditions (e.g., correct address and correct signature) are matched, independently on the arrival time that can vary because of the pipeline stalls induced. If the execution time reached a timeout, it will be considered unstable.

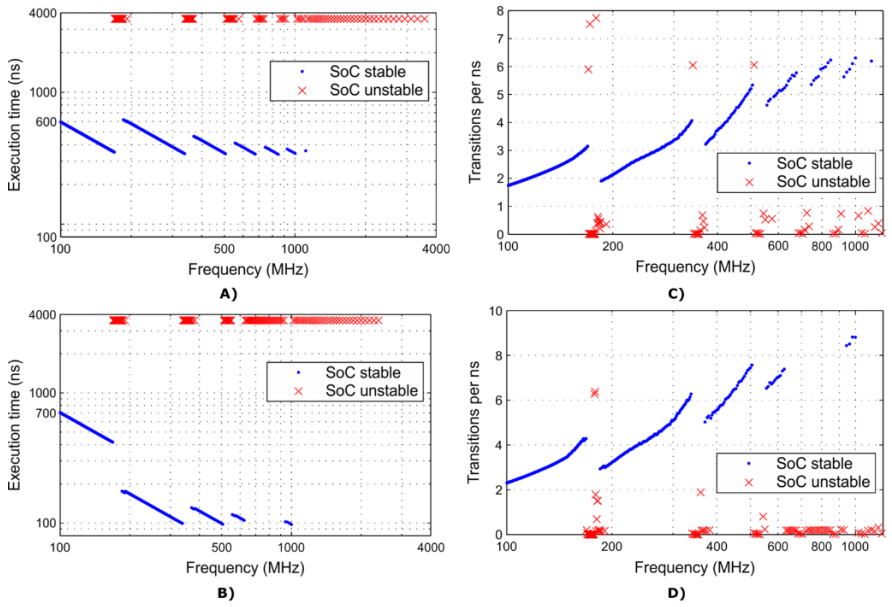


Fig. 10. Execution time of A) arithmetic units and B) branches; switching activity of C) arithmetic units and D) branches.

Figure 10c and 10d shows results in terms of switching activity. In this case, solid transitions are completely considered and counted, transitions from *X* value to 0 or 1, or from 1 or 0 to *X* are only considered for the 50% of their weight. In fact, it is quite usual in unstable states that the *X* values are overwritten with solid ones and vice versa. Switching activity value is normalized to 1 ns, meaning that the reported values correspond to the overall switching activity of the program over its length. For example, at nominal speed (STA reported about 100 MHz) the switching activity is 1.75 per ns, which is average for all observed gates.

Table 1. Analysis of functional programs results.

	Max switching activity [toggle/ns]	*Increase from nominal [%]*	*Frequency [GHz]*
Arithmetic	6.31	361	1.11 GHz
Branch	8.82	383	1.00 GHz

Table 1 shows the best increase in terms of switching activity per time unit. The maximum working frequency is about 1 GHz, which is around 10x of the nominal frequency. In this case the increase in the switching activity value is significant. It is also interesting to note that even at lower frequencies (i.e., within 400 and 500 MHz) the increase in the switching activity is already relevant as up to 6 toggle/ns.

A careful analysis of the switching activity result highlights some frequencies driving to unstable states, but with a large count of transitions per nanoseconds. In this case, the circuit is showing a kind of intermediate behaviour due to some specific timing violations. There are several asynchronous paths that are involved in the management of the memory operations; the longest in our case is relative to data bus management, and shows a propagation time of 0.92 ns, while the longest path involved to manage the code bus is slightly shorter, 0.78 ns. At some frequency, the data bus path timing requirements are violated, but not those related to code bus as it is less critical. In this case, the processor is only partially showing an unstable state, as *X* values are propagated only in a part of the synchronous circuits; switching activity value is mainly conditioned by a larger count of transitions form *X* to solid value and vice versa.

4.2 Automatic generation of functional stress programs

Table 2 summarizes the processor division as required in the first step of the proposed methodology. This division was made by splitting the processor hierarchically. In addition, the table also reports the number of instructions used to stress every processor module.

It is interesting to note that functional modules such as the arithmetic adder, as well as the multipliers, involve a small set of independent instructions; on the contrary, control related parts (e.g., the forwarding unit) require a higher number of instructions that are shared with other modules. The set of instructions derived for every module will be used in the following generation steps in the form of constraint libraries.

Table 2. Processor modules description and correlated instructions counting.

Processor module		# of Gates	# of Instructions
1	Adder	263	2
2	Decoder	2,700	52
3	Bus Control	1,172	30
4	Memory control unit	879	30
5	Forwarding unit	1,355	23
6	Prefetch	691	30
7	Fetch	647	30
8	Sys-coprocessor	1,785	4
9	Register Bank	13,858	32
10	Mult_signed	7,704	2
11	Mult_unsigned	7,652	2
12	Execution stage	7,623	33

The generation flow environment was developed using 4 Python scripts that support the implementation of the iteration loop proposed in Figure 9. The environment allows to set the modules order taken to progressively generate stress programs, the satisfactory stress threshold value, as well as the different parameters required by the generation tool (in this case *μGP*). Taking into account the experimental results obtained in [12], and considering the stress quality equation provided before, we empirically determine for these experiments a Satisfactory Stress Threshold (*SST*) able to evaluate the module stress goodness given by the following factor:

$$SST = \frac{1}{3} \cdot \frac{F_{FAST}}{F_n} \tag{10}$$

where F_{FAST} is the overclocked frequency at which the program is executed, and F_n is the nominal frequency of operation.

Thus, for every targeted module, the ratio SA_{RATIO} between the average SA measured at the FAST execution frequency (SA_{FAST}) and the nominal one (SA_n) should comply with the following condition:

$$SA_{RATIO} = \frac{SA_{FAST}}{SA_n} \geq \frac{1}{3}\frac{F_{FAST}}{F_n} \ . \tag{11}$$

This value is used during the stress program generation process performed for every module, defining in this way a stop condition to be provided to the evolutionary optimizer.

In addition, in order to determine which gates in the module under consideration are successfully stressed, we define a value (called *GSS*) related to the gate SA and the average SA in the module, in the following way:

$$GSS = 0.5\left(SA_{pk} + SA_{avg}\right) \tag{12}$$

where SA_{pk} is the SA peak obtained in the considered module, whereas SA_{avg} is the SA average compute on the gates belonging to the module.

The evolution of stress programs was divided in three different steps to be performed in a row. In the first one the modules labeled in Table 2 from 1 – 8 were tackled, and then in the second run, the rest of them. At the end of the process, a third independent run was performed tackling the processor gates still considered as too low according to the gate successfully stress parameter. In order to automatically generate the stress programs, 10 different constraint libraries were devised to feed μGP with the appropriate information regarding the module under evaluation. Interestingly, only the ones dedicated to the functional modules were independent, while the others were shared and composed of the most instructions in the processor ISA.

The evolution of the generation process is detailed in Table 3, Table 4 and Table 5. The generation process is developed in 3 different levels. The table shows for every generation level, the average SA_{avg} for the modules under consideration, the FAST frequency of the stress program generated, the SST value, SA_{RATIO} and SA_{peak} measured during the generation process, the *GSS* value for the gates in the module, and in addition, the number of gates successfully stressed (**ssg**) according to the previous parameters.

Table 3. Stress programs evolution (Level 1).

.	**L1**							**SYNC. 1**
Processor module	SA_{avg}	**freq [MHz]**	**SST**	SA_{ratio}	SA_{peak}	**GSS**	**ssg**	**Δssg**
Adder	18.65	652.74	2.15	2.07	24.70	21.675	0	233
Decoder	88.17	612.37	2.02	3.16	92.40	90.285	1001	322
Bus Ctrl	128.28	684.46	2.26	4.75	136.2	132.25	695	416
Mem	110.12	676.59	2.23	4.53	119.1	114.64	756	64
Forward	53.34	651.04	2.14	2.29	74.22	63.780	650	386
Prefetch	34.08	512.30	1.69	6.31	49.83	41.955	410	82
Fetch	79.55	595.59	1.96	4.48	83.84	81.695	421	108
Sys-cop	24.46	623.44	2.05	1.75	32.51	28.485	0	570
Reg. Bank	-	-	-	-	-	-	-	-
Mult_sign.	-	-	-	-	-	-	-	-
Mult_unsig.	-	-	-	-	-	-	-	-
Execution stage	-	-	-	-	-	-	-	-

Furthermore, the table shows for each synchronization step (1 to 3), the additional gates (**Δssg**) successfully stressed by exploiting the positive effects of different programs in the targeted module. Finally, the last column of each table (3, 4 and 5) shows the final number of gates successfully stressed for every module considering all the previous contributions.

During the first generation level, the initial set of modules (the first 8 in the Table 2) is targeted. Then, the second set of modules. After every generation level, a synchronization step is performed. Then, the third synchronization step evaluates the different programs on all the other modules. The level 3 generation intends to stress the weak gates still present in the processor, and shows again the **Δssg** obtained in this phase. The reader may notice that there is a positive effect obtaining during the synchronization phases on the different modules. This positive effect reduces the generation time, allowing the test engineer to discard some gates for free.

Table 4. Stress programs evolution (Level 2).

	L2							**SYNC. 2**
Processor module	SA_{avg}	**freq [MHz]**	**SST**	SA_{ratio}	SA_{peak}	**GSS**	**ssg**	**Δssg**
Adder	-	-	-	-	-	-	-	-
Decoder	-	-	-	-	-	-	-	-
Bus Ctrl	-	-	-	-	-	-	-	-
Mem	-	-	-	-	-	-	-	-
Forward	-	-	-	-	-	-	-	-
Prefetch	-	-	-	-	-	-	-	-
Fetch	-	-	-	-	-	-	-	-
Sys-cop	-	-	-	-	-	-	-	-
Reg. Bank	25.46	669.79	2.23	2.20	36.11	30.785	0	2,103
Mult_sign.	83.41	674.31	2.25	2.60	90.81	87.110	2,314	320
Mult_unsig.	85.08	677.97	2.26	2.60	91.03	88.055	2,433	287
Execution stage	99.55	677.97	2.26	2.51	107.05	103.300	1,987	1,208

It is interesting to note that in some cases, for example during the adder stress process, the SA_{RATIO} is lower than the *SST* parameter, then, since the condition is not satisfied no gates can be considered as thoroughly stressed. Then, in these cases, it is necessary to consider not only the stress programs developed targeting the module, but all

the others. This is the reason why the gates successfully stressed in the adder appear only during the first synchronization step.

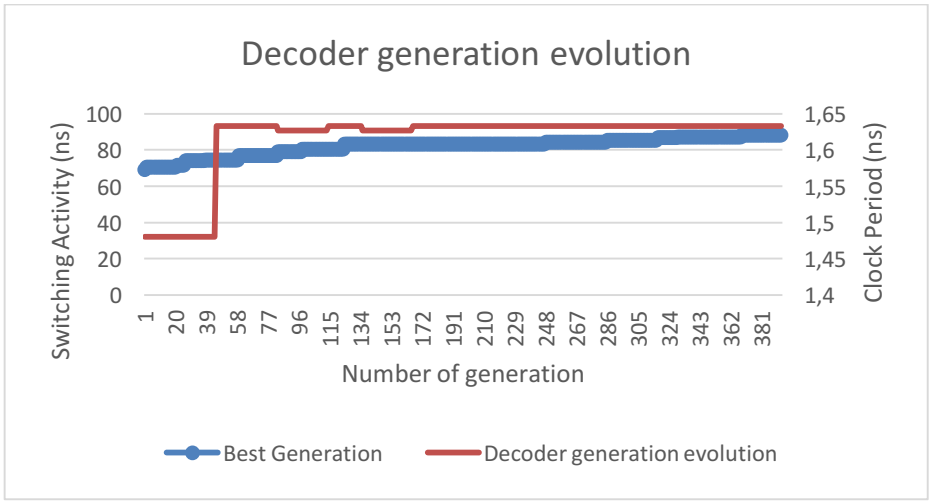
For the purpose of comparison, typical scan values may be considered. From our experience, we state that this structural solution for stress is strongly limited by the tester environment (such as along Burn-In testing). Additionally, scan procedures for stress need an accurate evaluation in terms of power consumption, as they are not functional. Thus, it is quite usual that a low power procedure is finally adopted, which are showing very low switching activity values.

Table 5. Stress programs evolution (Level 3 and total amount).

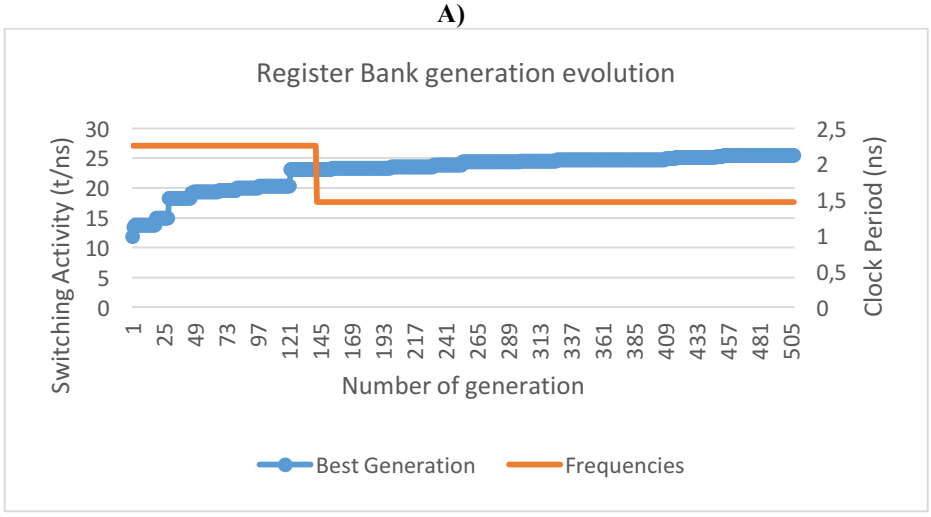
	SYNC. 3	**L3**	**TOT**
Processor module	**Δssg**	**Δssg**	**ssg**
Adder	0	12	**245**
Decoder	785	250	**2,358**
Bus Ctrl	20	3	**1,134**
Mem	3	0	**823**
Forward	12	53	**1,101**
Prefetch	0	9	**501**
Fetch	4	21	**554**
Sys-cop	0	2	**572**
Reg. Bank	5,601	1,007	**8,711**
Mult_sign.	302	208	**3,144**
Mult_unsig.	298	306	**3,324**
Execution stage	2,110	809	**6,114**

The experiments described before were run in a workstation based on 2 Intel Xeon E5450 CPUs running at 3.00 GHz, equipped with 64 GB of RAM. Every stress generation process requires about 12h. However, the process is highly parallelizable and it globally requires about 3 working days.

Checking some graphs gathered from the evolution of the stress programs, in particular, Figure 11a depicts the evolution of the best stress programs for the decoder unit. The figure shows the evolution of the best individuals considering the maximum average SA (transition/ns) and the FAST execution frequency. It is interesting to see how at the very beginning of the evolution process there are some programs that run at high frequencies (about 1.475 ns = 677.9 MHz) and very low SA average (70 transitions/ns), but later, the evolution improves the SA even reducing the FAST frequency.



A)



B)

Fig. 11. Stress programs evolution for decoder unit (A) and register banks (B).

On the contrary, Figure 11b shows a different shape evolution where the frequency starts at a very low value (about 2.25 ns = 444.4 MHz), and later the frequency as well as the SA is increased.

5 Conclusions

In this contribution, we proposed a functional approach that can allow the target processor to work at faster-than-at-speed frequencies along with a method to identify which are the frequency ranges in which this technique is applicable. Furthermore, we proposed a novel technique able to automatically generate stress programs for pipelined processors at faster-than-at-speed frequencies exploiting the previous analysis.

Pushing the clock frequency to values higher the nominal is feasible thanks to processor architectural features such as pipeline stalls and memory wait states. The cause of blocking states is mainly attributable to the propagation of signals that violates the design timing constraints resulting in unknown values sampled by FFs. The enhanced switching activity achievable with this technique can be exploited to develop a much more reliable and effective electrical activity on a given device reaching high thermal stress.

For this reason, the analysis is followed by a possible exploitation of a program that can be executed at frequencies higher than the nominal one guaranteeing a high stress capacity. In fact, an automatic generation process for FAST functional programs is developed and provided, in order to reach a high grade of stress for pipelined processor cores. The proposed method splits the processor hierarchically and generates stress programs for every one of the processor modules. Then, the positive side effects of these programs are considered on the rest of the processor, reducing in this way the generation times. The method suitability was experimentally evaluated in a MIPS-like processor core obtaining a set of stress programs containing 13 programs running at frequencies about 6.4 higher than the nominal ones. In addition, from the performed experiments, it is possible to see that more than the 60% of the processor gates were labelled as successfully stressed gates.

The proposed method is a preliminary study carried out on a *MIPS-like* processor as reference architecture in order to verify its feasibility. The next step is to migrate this approach on an industrial device, identifying non-blocking regions in which it is possible to manage the activity of the processor and create a frequency-targeted stress. Moreover, the intrinsic dependency of internal delays to the real die temperature will be further and better investigated on silicon.

6 References

1. S. Rosinger, M. Metzdorf, D. Helms, W. Nebel, "Behavioral-level thermal- and aging-estimation flow," in *2011 12th Latin American Test Workshop (LATW)*, Porto de Galinhas, 2011, pp. 1-6.
2. Way Kuo and Yue Kuo, "Facing the headaches of early failures: A state-of-the-art review of burn-in decisions," in *Proceedings of the IEEE*, vol. 71, no. 11, pp. 1257-1266, Nov. 1983.
3. D. Appello, P. Bernardi, R. Cagliesi, M. Giancarlini, M. Grosso, E. Sanchez, M. Sonza Reorda, "Automatic Functional Stress Pattern Generation for SoC Reliability Characterization," in *14th IEEE European Test Symposium (ETS)*, Sevilla, 2009, pp. 93-98.

4. S. Hellebrand, T. Indlekofer, M. Kampmann, M. A. Kochte, C. Liu and H. J. Wunderlich, "FAST-BIST: Faster-than-at-Speed BIST targeting hidden delay defects," in *2014 International Test Conference (ITC)*, Seattle, WA, 2014, pp. 1-8.
5. H. Yan, A. D. Singh, "Experiments in detecting delay faults using multiple higher frequency clocks and results from neighboring die," in *2003 International Test Conference (ITC)*, 2003, pp. 105-111.
6. R. Tayade and J. A. Abraham, "On-chip Programmable Capture for Accurate Path Delay Test and Characterization," in *2008 IEEE International Test Conference (ITC)*, Santa Clara, CA, 2008, pp. 1-10.
7. S. Pei, H. Li and X. Li, "An on-chip clock generation scheme for faster-than-at-speed delay testing," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, Dresden, 2010, pp. 1353-1356.
8. M. Kampmann, M. A. Kochte, E. Schneider, T. Indlekofer, S. Hellebrand and H. J. Wunderlich, "Optimized Selection of Frequencies for Faster-than-at-Speed Test", 2015 24th IEEE Asian Test Symposium, pp. 109-114.
9. N. Ahmed, M. Tehranipoor and V. Jayaram, "A Novel Framework for Faster-than-at-Speed Delay Test Considering IR-drop Effects," in *2006 IEEE/ACM International Conference on Computer Aided Design*, San Jose, CA, 2006, pp. 198-203.
10. M. Amodeo and B. Cory, "Defining faster-than-at-speed delay tests," in *Cadence Nanometer Test Quarterly*, 2(2), May 2005.
11. M. Psarakis, D. Gizopoulos, E. Sanchez, M. Sonza Reorda, "Microprocessor Software-Based Self-Testing," in *IEEE Design & Test of Computers*, vol.27, no.3, pp.4-19, May-June 2010.
12. P. Bernardi, A. Bosio, G. Di Natale, A. Guerriero, F. Venini, "Faster-than-at-speed execution of functional programs: an experimental analysis," in *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Tallinn, 2016.
13. miniMIPS, *Opencores.org*, 2016. Available: http://opencores.org/project,minimips .
14. Primetime, *synopsys.com*, 2016. Available: https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html .