Operation-Dependent Frequency Scaling Using Desynchronization

Nitish Srivastava, *Student Member, IEEE*, and Rajit Manohar, *Senior Member, IEEE*

***Abstract*—Asynchronous circuits are inherently more robust than their synchronous counterparts. Desynchronization is a way to obtain asynchronous circuits from a synchronous specification using standard design tools while improving circuit for variation tolerance, electromagnetic interference, and resulting in similar area, delay, and energy as the synchronous baseline. This paper proposes a novel operation-dependent desynchronization technique, which desynchronizes the circuit and improves performance beyond the limits of synchronous design. We perform a case study of our proposed technique on RISC-V rocket core and show significant improvement in performance with minimal power and area overheads.**

***Index Terms*—Asynchronous circuits, circuit optimizations, desynchronization, frequency scaling, synchronous designs.**

I. Introduction

ASYNCHRONOUS circuits have several potential benefits over synchronous circuits. They are less prone to failure due to process variations, voltage, temperature, etc., which can reduce the growing timing complexity in digital design. In spite of these benefits, industry has been reluctant to fully migrate to asynchronous methodology due to the costs and risks of leaving synchronous domain, which has a long legacy of success and sophisticated computer-aided design (CAD) infrastructure. One of the drawbacks of asynchronous design is that standard CAD tools for logic synthesis cannot be used for control signals, since those signals must be hazard-free. Asynchronous designs such as quasi-delay-insensitive (QDI) circuits which do not make assumptions about the timing of different components have immature CAD flow. The designing of these circuits is also very different from synchronous circuits and has a steep learning curve for designers. Bundled data are another approach of designing asynchronous circuits where similar to synchronous design combinational logic blocks are implemented using standard CAD tool flow. It uses asynchronous handshake controllers to control the data transfer between multiple combinational blocks. These designs are not self-timed such as QDI circuits as they require the knowledge of delays of the combinational blocks. For QDI circuits, the design itself is delay insensitive and hence the correctness of the circuits does not depend on supply voltage, operating temperature, and process variations. Bundled data circuits on the other hand have local handshake controllers which are placed close to the combinational blocks and hence undergo similar voltage, temperature, and process variations as the combinational block. The closed-loop property of the asynchronous circuits distinguishes them from the synchronous circuits. Asynchronous circuits work on the principle of handshakes. If one component takes more time to perform the computation, the rest of the components wait for this computation to finish. Unlike synchronous design where any change in timing of a component can result in circuit failure. Techniques for automated conversion of synchronous to asynchronous designs are appealing, because they can address the challenges of asynchronous design yet provide the benefits of asynchronous designs.

Desynchronization [1], [2] and phased logic [3] are two techniques that convert synchronous designs to asynchronous ones. Linder and Harden introduced the notion of phased logic, in which each synchronous combinational logic gate is replaced with a small sequential handshaking asynchronous circuit. Cortadella *et al.* [1] and Andrikos *et al.* [2] use a fully automated synthesis flow which do not change the overall structure of the synchronous design and show that the resultant asynchronous circuit has similar delay, and energy compared to the original synchronous design, while improving the timing variations and electromagnetic interference (EMI) at the same time. Cortadella *et al.* [1] showed a 22% area overhead in the desynchronized circuit, however, a most of that overhead comes from using latches instead of registers. Bertrand *et al.* [4] proposed a desynchronization technique which uses registers instead of latches and has small controllers to decrease the area overhead of desynchronization. Their proposed technique showed significant power and energy reduction while similar performance compared to standard desynchronization using latches as in [1]. To improve the performance of a desynchronized circuit, Weaver [5] and Proteus [6] and [7] have explored the possibility of converting synchronous designs into an aggressively pipelined asynchronous implementations that can exceed 1 GHz in frequency. While these approaches can improve performance in some cases, the cost in area and power can be significant [8]. Semeraro *et al.* [9] designed a multiple clock domain microarchitecture using a globally asynchronous, locally synchronous clocking style to solve the challenges of globally clocked

Manuscript received May 19, 2018; revised August 24, 2018 and November 5, 2018; accepted November 16, 2018. Date of publication December 25, 2018; date of current version March 20, 2019. This work was supported by the NSF Award 1065307. *(Corresponding author: Nitish Srivastava.)*

N. Srivastava is with the Department of Electrical and Computer Engineering, Cornell University, Ithaca, NY 14850 USA (e-mail: nks45@cornell.edu).

R. Manohar is with the Department of Electrical Engineering and the Department of Computer Science, Yale University, New Haven, CT 06520-8267 USA (e-mail: rajit.manohar@yale.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2018.2885335

synchronous systems and showed a performance degradation of less than 4%.

A traditional asynchronous design methodology has the potential to achieve better performance than their synchronous counterparts, because it is data driven and activate only those paths in the design which are needed for certain computation. It is challenging to exploit this property when starting from a detailed synchronous design, because the design is created with a global clock in mind and it becomes difficult to determine what operations are in fact data dependent after the design has been transformed by logic synthesis tools. Proteus [6] provides support for conditional send and receive primitives as a way to reduce power overhead of the baseline flow with user intervention as well as some automation support. Mendes *et al*. [10] proposed a tool for synthesizing asynchronous circuits using extended signal transition graph specifications. Telescopic units [11] breaks down the critical paths to operate in two clock cycles instead of one and uses remaining paths to determine clock frequency of the entire design. Toosizadeh *et al*. [12] recognized the fact that all critical paths in a synchronous design are not active at the same time and designed a control circuitry which changes the clock period every cycle based on the largest critical path used by any of the operations in the pipeline. However, it uses a central clock pulse generator so the clock tree still needs to be synthesized and clock issues related to spatial–temporal variations, and EMI are not resolved. Xu and Wang proposed a variable delay line and use Mousetrap controllers [14] to desynchronize synchronous circuits. However, the study was done on a simple linear pipeline of floating point adder which does not consist of any forks or joins. Saifhashemi *et al*. [15] proposed a resynthesis technique to improve the performance of desynchronized bundled data circuits. A performance improvement of 25% was shown over the synchronous baselines. However, the circuits used were instruction decode and execution units which are much simpler than entire processor pipelines.

This paper builds on the desynchronization approach for synchronous-to-asynchronous conversion, which is known to produce asynchronous designs that have about the same area, delay, and energy of their synchronous baseline, but are inherently closed loop and hence, more robust against supply voltage, operating temperature, and large-scale process variations [1]. We introduce operation-dependent desynchronization, a technique that requires minimal designer effort but results in improved average-case performance while preserving the core benefits of desynchronization. Our key contributions are as follows.

1) We identify the performance limitations in existing desynchronization approaches, discussing why similar techniques do not provide better performance than the corresponding synchronous design.
2) We propose a novel desynchronization technique which does not modify the datapath of the synchronous design similar to the previous approaches, but can perform better than the baseline synchronous design.

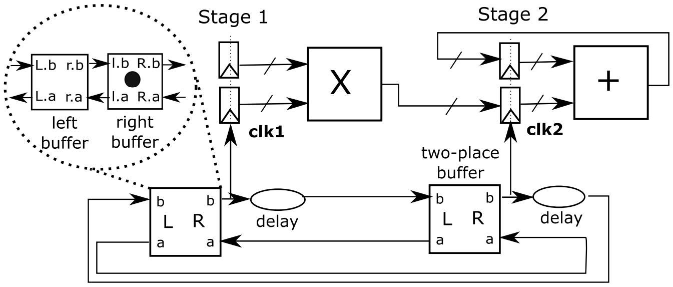


Fig. 1. Two-stage pipeline for multiply accumulate and the asynchronous control to provide the clock.

3) We perform a case study of the proposed technique on the RISC-V processor pipeline from the University of California Berkeley and show a significant improvement in performance and energy efficiency for many benchmarks.
4) We use standard CAD flow for simulation and synthesis in our proposed approach and hence our approach does not required a different CAD tool flow.

The rest of this paper is organized as follows. Section II discusses the traditional desynchronization technique on a toy example and its limitations. Section III describes the proposed desynchronization technique. Section IV discusses the desynchronization of RISC-V processor pipeline along with experimental results, followed by conclusions in Section V.

II. Standard Desynchronization

Desynchronization techniques that use standard design tools and replace clock network with an asynchronous circuit which is responsible for generating the clock signals. At one extreme, desynchronization results in a design where each one-bit register has its own clock generated by the asynchronous control, which requires as many asynchronous controls as the number of registers and can result in high area and power overhead. At the other extreme, the entire system uses a single asynchronous controller that generates one clock for all the registers—which is the same as the synchronous baseline. To reach a balance between the two extremes, researchers often use a single clock for all registers in a pipeline stage but different clocks for different pipeline stages as in [1] or use some clustering and separation analysis methods to reduce the controller overhead [16]. These techniques reduce area and power overheads and at the same time desynchronize the circuit with low design effort.

A. Desynchronization Example

We first explain desynchronization based on pipeline stages on a very simple example of a synchronous multiply-accumulate pipeline which multiplies two inputs and accumulates the result, as shown in Fig. 1. We use asynchronous buffers to design the asynchronous circuit for generating clock for each stage. There have been many studies on different asynchronous handshake controllers that can be used for desynchronization with different signal transition graph specifications [1]; we simply use one design throughout this paper for simplicity, and the technique can be incorporated into other designs without difficulty.

The desynchronization controller that we use is a two-place buffer with input channel L and output channel R and an initial token (one bit), as shown in Fig. 1. These buffers have two ports on the left channel (L), `req` port (L.b) to receive a communication token, and `ack` port (L.a) to acknowledge the sender about the received token. On the right channel (R), the `req` port (R.b) sends the token to the buffer connected to the right and the `ack` port (R.a) receives the acknowledgment. To desynchronize the two-stage multiply-accumulate example, we design an asynchronous control having two two-place buffers for the two stages connected in a ring, as shown in Fig. 1. Each two-place buffer is made up of two one-place buffers where one buffer starts by waiting for a token on the `L` channel and then sends the token on the `r` channel, which we denote as left buffer. The other buffer starts the communication by sending a token on the `R` channel and then waits for a token on the `l` channel, which we denote as right buffer, as shown in Fig. 1. The following equations show the implementation of both left and right buffers in communicating hardware processes (CHP) language [17]:

$$*[[L.b];\ L.a+;\ [\sim L.b];\ L.a-;\ r.b+;\ [r.a];\ r.b-;\ [\sim r.a]]$$

$$*[R.b+;\ [R.a];\ R.b-;\ [\sim R.a];\ [l.b];\ l.a+;\ [\sim l.b];\ l.a-].$$

The first equation implements a left buffer and the second implements a right buffer, + denotes a rising transition, - denotes a falling transition, `[]` denotes a `wait` till the signal becomes true, `;` denotes the sequential composition and `*` at the beginning means that the same process keeps repeating forever.

In Fig. 1, whenever a right buffer makes a positive transition on the right request wire (`R.b`), it also provides a positive edge of the clock to the registers in that stage. A delay line on the right request wire is used to ensure that it takes at least the maximum logic delay of the sender stage to complete a send to the left buffer of the receiver stage. Once the send is complete left buffer in receiver stage can pass that token to its right buffer which then initiates a send on its R channel and also provides a positive clock edge to the registers of that stage. The delay lines ensure that not only the output of the combinational logic between the two stages is stable but also that any combinational logic path internal to a pipeline stage is stable.

In general, for correct functionality, any two connected stages in synchronous design will require a delay line between their asynchronous buffers. The delay for these delay line is set by the maximum of the reg-to-reg delay from source to the destination stages and intrastage reg-to-reg delay of the source. The delay of the gates in the asynchronous controller can also be taken into account when determining the delay of the explicit delay line. In the case of nonlinear pipelines, where the output of one synchronous stage can go to multiple stages, different delay lines are placed between any set of source destination pair based on the appropriate path delays. The acknowledgment wire ensures that the data have been correctly read by the destination registers, and hence, a Muller C-element is used to ensure that the source buffer receives the acknowledgment only after all the destination buffers have acknowledged. When a destination stage receives data from two or more stages, a Muller C-element on req wire is used to ensure that the input is read only after the data from all the sources are stable. This construction is also the basis for desynchronization [1].

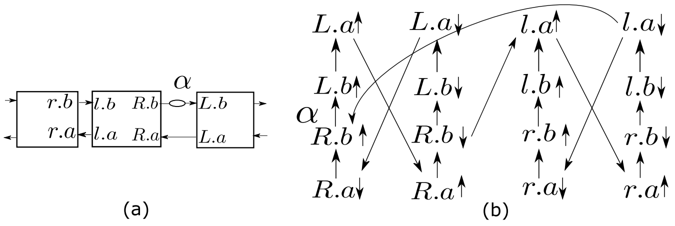


Fig. 2. (a) Asynchronous control for a single pipeline stage. (b) Timed graph showing the largest algorithmic cycle.

The delay lines are sufficient to ensure that the setup times of flip-flops are satisfied. To ensure that the hold times are also satisfied, one can make a buffer as slow as its neighbor or can use a handshake controller which accounts for hold time [1]. We use the first approach in our design throughout this paper.

B. Observations on Performance

After desynchronizing the multiply-accumulate pipeline, we do not observe any improvement in performance. The reason is even though the delay of the accumulate stage is small, the overall cycle time (time for a complete handshake in the asynchronous ring) is still determined by the longest delay, i.e., delay through the multiplier. Thus, the desynchronized pipeline achieves the same performance as the synchronous counterpart. We realize that any desynchronization technique, whether based on pipeline stages as in [1] or clustering and separation analysis as in [16], which does not change the synchronous logic itself and just tries to alter the clock network cannot perform better than the original synchronous circuit. The worst case timing path in the clocked implementation is a reg-to-reg path that still exists in the design and must appear as part of some delay line in the system. It is well-known that the throughput of an asynchronous system where the system has and causality (as is the case for the asynchronous controllers) is determined by the maximum cycle mean of the timed event graph [18]. Fig. 2 shows an asynchronous buffer with a token connected via wires to an empty buffer on its left and using a delay to an empty buffer on its right and part of the corresponding timed event graph. There is a simple cycle with a cycle mean corresponding to the delay line—and hence, the maximum reg-to-reg delay between the two desynchronization regions. Therefore, for any synchronous design, the largest logic delay will show up in some cycle in the event rule graph and hence the throughput will always be limited by the maximum of all the reg-to-reg delays in the circuit—which is the clock period of the original synchronous circuit.

III. Proposed Approach

The clock period of a synchronous circuit is set by the longest path between two registers known as the critical path. When there are a small number of critical paths that are primarily limiting performance, synchronous designers use pipelining, retiming, clock domain crossing (CDC), dynamic time borrowing, and various other techniques to improve throughput. These techniques, however, come with the cost of significant design effort, area, and energy and also may not always be feasible.

A. Design Modularity

Most complex designs consist of a large collection of modules. Even though timing characterization requires global analysis, a chip is broken down into modules for ease of understanding and design. Many times most of the critical paths lie in a subset of modules which may not be needed all the time and the active state of those modules can also be determined based on the operations being performed in the current cycle. Our proposed operation-dependent desynchronization technique focuses on exploiting this observation.

When confronted with a slow module, a designer could make major architectural changes to ensure that the slow module is no longer on the critical path. Examples of this include repipelining the design, or operating the module in a separate clock domain and using clock domain conversion logic, etc. However, these are very intrusive changes and require significant effort. Also, given two modules A and B, with A running at a frequency slightly higher than B, while having many critical paths in B, a designer might have to pipeline all of those paths or compromise the frequency of the overall system. Neither of these are good options if the design effort, area, energy, and power overheads of pipelining cannot be justified—especially, if module B is infrequently used. The situation is even worse when the design has a large number of modules.

In synchronous designs, satisfying timing constraints statically for all paths is a hard constraint which is required to run the whole design on a single clock. Current synchronous design tools have support for clock gating registers in individual modules to save power and energy when they are not needed. Designers use valid bits or enable bits to indicate when a section of the logic in their design can be clock gated. While this is highly effective at reducing power consumption, it does not help with performance. If we can exploit the information about the active state of the modules, one can imagine designing a clock that can vary its frequency from one cycle to the next based on the dynamic information about the active components in the design. This kind of dynamic frequency scaling can improve the performance of synchronous circuits which are designed using single clock. However, designing such a centralized clock is not an easy task since the frequency for each clock cycle is a function of the global state of all the modules on that cycle. The set of possible configurations that must be considered grows quickly with the number of modules in the system.

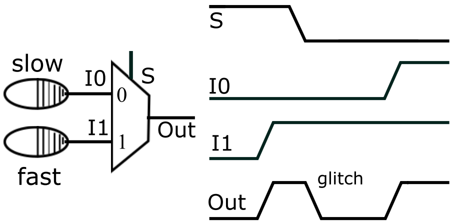


Fig. 3. Slow multiplexer select signal can result in a glitch causing spurious handshakes.

B. Operation-Dependent Desynchronization

Instead, we propose a novel distributed technique that accomplishes the same goal that we call operation-dependent desynchronization. Using only information about a module and the operations for which it is active, we create a simple handshake controller that permits the global system throughput to dynamically increase when modules that contain the critical path are not active. The technique is scalable, and circuit overhead grows linearly with number of regions where the technique is applied.

First of all, the synchronous design is partitioned into desynchronization regions and a two-place buffer is used to provide clock to each region instead of pipeline stages as in standard desynchronization (Section II). A desynchronization region consists of a single module or a group of modules in the synchronous design. Fig. 4 shows a simple example with five regions each of which has some registers and some combinational logic. These regions can consist of any synchronous design which involves single clock with positive edge triggered logic. We will not discuss generated clocks, multiple clocks, and latch-based designs in this paper, but the proposed techniques are generic enough to be adapted for these designs with minor modifications. The regions can, however, consist of any number of pipeline stages connected in any form. All registers in a region are provided the same clock; however, registers in different regions are provided different clocks. The clocks are generated from the asynchronous control shown in Fig. 4(b). The buffers of different regions communicate with each other via handshakes based on the topology of the synchronous design. If the modules in a region send data to some other module in a different region then there exists a communication channel between the asynchronous buffers for the two regions. The request wire on the R channel of the buffer of any region is used as clock for all the modules in that region. To ensure correct data delivery, a delay line is placed on the request wire between the buffers of the sender and receiver stages so that the communication token is received by the receiver stage only after the delay requirement corresponding to largest reg-to-reg path in the sender or between the sender and receiver stages is satisfied. Designing a delay line with a predictable delay is a thoroughly studied problem. A recent reference that describes how to synthesize delay lines that track the delay of the combinational logic they are supposed to match is [19]. A lower power and mixed-signal solution for long delay lines is [20].

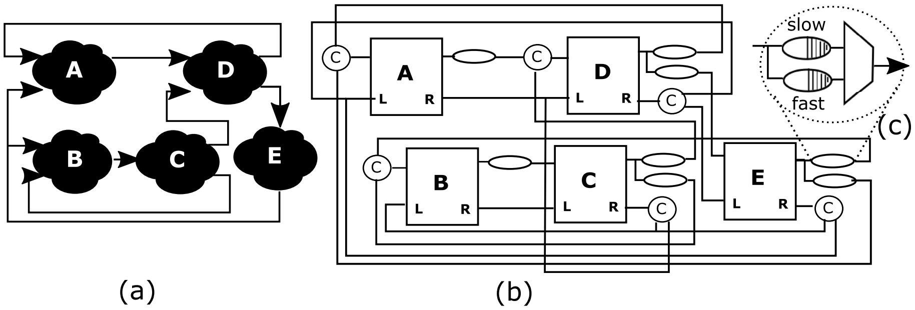


Fig. 4. (a) Synchronous designs with five modules. (b) Asynchronous controller for clock generation. (c) MUX to skip the delays between the modules.

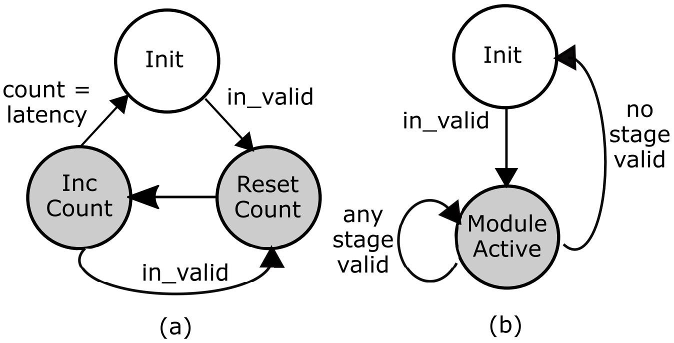


Fig. 5. FSM for modules with (a) fixed latency and (b) valid signals. Shaded region: states in which the module is active and in_valid: signal which tells whether an input will be fed to the module in the next cycle or not.

The performance of this desynchronized design is still limited by the worst case reg-to-reg delay as the largest cycle in the timing graph is the one that corresponds to the longest critical path. To go beyond this performance limit, we make use of the active state of different desynchronization regions. For each region, we assume that we can use the operations being performed in the synchronous circuit to identify whether or not the modules in that region are active in the current clock cycle or not. When a module is inactive, its state does not change and hence any path that originates from the registers in the module can be ignored when setting the delay of that region. We replace single delay lines with a set of delay lines and multiplexers which pick the right delay line depending upon whether a module is idle or not, as shown in Fig. 4. This permits the design to operate at a higher frequency, in situations when the part of the system that contains the critical path is idle.

Even when all the modules are active, it is possible that certain critical paths in a module are required for a particular operation, but can be ignored most of the time. For example, a critical path through multiplier in an in-order pipeline can be ignored when there are no multiply instructions in the pipeline. In these scenarios, frequency can still be increased even if all the modules are active and one can imagine having extra delay lines corresponding to these fast frequencies and logic to determine when such critical paths are inactive.

Finally, we note that in many synchronous designs, determining the active state of a module is not a challenging task. This information is often present in the design in the form of valid signals which are used by design tools for power optimizations such as clock gating or can be easily determined by identifying the operations, for example, by matching opcodes in processor pipelines, current cycle count, and pipeline initiation interval in the case of CGRAs, etc. Many of the modules have pipelined datapaths with valid signals or have fixed latency which can also be used to determine the active state of the modules as shown in Fig. 5. The finite-state machine (FSM) on the left corresponds to a module which has a fixed latency and hence by using an incrementing counter it can be easily determined when the module is active or idle. The FSM on the right corresponds to the module which has valid signals in each pipeline stage. Here, the idle state of a module can be determined by combining the valid signals from different stages using OR logic. The only design overhead that the synchronous designers have to pay is to create a small as shown in Fig. 5 which generates information about the active state of a module based on operations in the pipeline or various valid signals from different modules. Furthermore, the output of the FSM responsible for selecting the delay line using multiplexers should be faster than the fastest of all the delay lines that are input to this multiplexer. As shown in Fig. 3, if the output of the FSM going into the multiplexer select signal is slower, then a glitch can appear on the multiplexer output resulting into undesired handshakes between the asynchronous controllers. However, this is not a problem most of the time as the logic delay of the combinational logic for select signals based on the current FSM state is quite small compared to the delay lines determining the clock period.

IV. Case Study: RISC-V Rocket Core

To demonstrate the proposed operation-dependent desynchronization technique, we did not want to create our own synchronous design and then use it to show improvements, and hence, we are demonstrating this idea on the open-source implementation of RISC-V rocket core developed at the University of California Berkeley [21]. We synthesize the RISC-V core using Synopsys Design Compiler (compile_ultra) and generate standard cell netlist and timing reports using a low-power 28-nm library. The timing reports show that with a 28-nm technology node, the design is able to achieve a clock period of 1.8 ns, i.e., 555 MHz and most of the critical paths are in the floating point unit (FPU), some in the mul–div unit of the core and few in the control unit corresponding

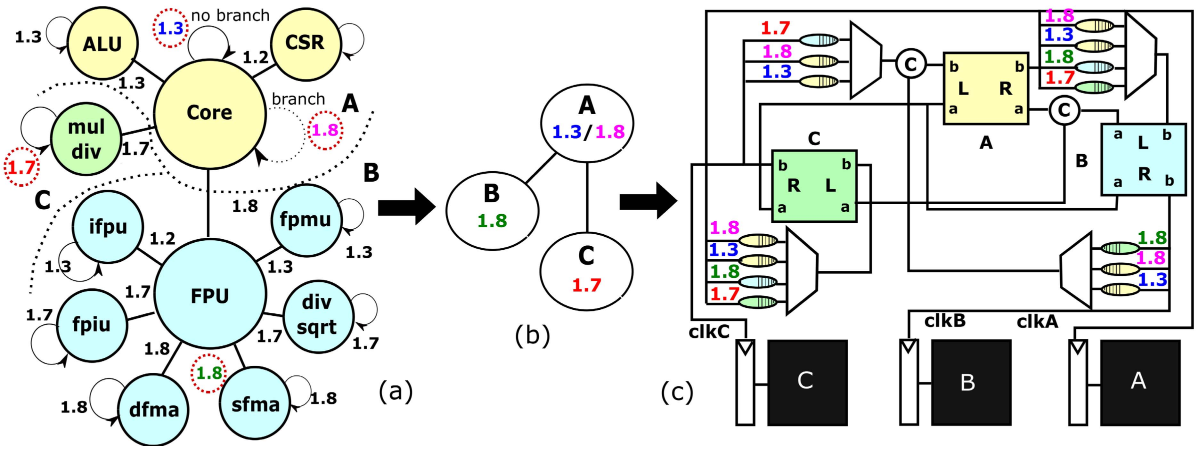


Fig. 6. Desynchronization of RISC-V core. (a) Module graph. (b) Desynchronization regions and clock period required for each of them. (c) Asynchronous control for clock generation.

to branch instructions. Ignoring these units/paths helps us to achieve a clock period of around 1.3 ns, i.e., a frequency of 770 MHz. As all the instructions in any application need not be floating point, multiply/divide or branch, ignoring the slower modules/paths when they are not used can potentially improve the average-case performance.

To determine the desynchronization regions, we first extract a graph whose vertices represent different modules in the design, edges represented the existence of a reg-reg path between the two modules, and weight on the edges represent the longest reg-reg path delay between the two modules, as shown in Fig. 6(a). The self-edges represent the longest reg-reg delay within a module. If the self-edge contains a path which can be easily detected at run time, and the rest of the paths have delay significantly less than the self-edge, the self-edge is deprecated to the next largest critical path and two delay lines are assigned for these two different critical paths. For example, in Fig. 6(a), the core module had critical paths corresponding to branch instructions which were significantly larger than other paths and hence the self-edge (shown as dotted arrow) is deprecated to the second largest delay of 1.3 ns (shown as solid arrow) and an extra delay line of 1.8 ns is added in the asynchronous control. To construct the module graph, we wrote a python script which parses the netlist obtained after the synthesis using Synopsys Design Compiler. It creates a list of different modules in the design using the module definitions in the Verilog netlist and then determines all reg-to-reg paths between two modules. It then creates the module graph by assigning a node to each module and connecting the nodes using edges when there is reg-to-reg path between two modules. Then, weights on the edges are assigned using the delay of the longest reg-to-reg path between two modules from the timing reports.

Next, this graph is split into three different regions such that each region gets its own local asynchronous clock generator. Splitting of the graph into different regions are not automated at this point and has to be done manually. One could use sophisticated separation analysis and clustering techniques as in [16] for finding the desynchronization regions that would minimize the overall controller area. However, for RISC-V rocket core, we assign the modules which are conceptually closer, i.e., share the same opcode or similar functionality to a single region. For example, different units in the FPU are assigned to a single region as it becomes easier to detect the active state of the region just based on the opcode, resulting in small detection circuit. Since the operations being performed inside the arithmetic logic unit (ALU) such as add, bitwise-AND/OR/XOR, comparisons all resulted in a critical path less than 1.3 ns, all of them are combined together with the core and assigned to a single desynchronization region. In case, the critical paths in the ALU had been significantly different for different operations, one could split different operations inside the ALU into different regions as well. In our case, since multiply and divide were the only operations that had significantly larger critical paths, they are assigned to a different desynchronization region.

The next challenge was to determine when these regions are active. After studying the RISC-V rocket core design, we realized that the active state of the FPU can be easily determined by inspecting the valid bits of the pipeline stages in the FPU, as shown in Fig. 5(b). For multiply/divide instructions, we used the valid bit in the val/rdy interface of the multiply/divide module and the pipeline latency to create an FSM as shown in Fig. 5(a) and used opcode matching for branch instructions. As the cycle period needs to be scaled up before the instruction enters these modules/paths, a small opcode matching logic was placed in the fetch stage, which can predict ahead of time whether the instruction is floating point, multiply/divide, or branch and the frequency can be scaled before the instruction actually gets executed. This technique, however, has its own drawback—for example, even when the floating point/multiply/divide/branch instruction is stalled due to some dependence, the processor would still run at a lower frequency even though these units/paths are not being used. For simplicity, we use this approach; however, one can use more complex logic that only indicates whether the module/path is going to be active in the very next clock cycle or not. Next, we design the asynchronous handshake controllers for the three regions. Fig. 6(c) shows the delay lines for region A corresponding to the maximum reg-to-reg

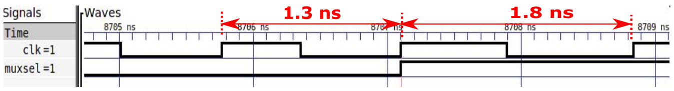
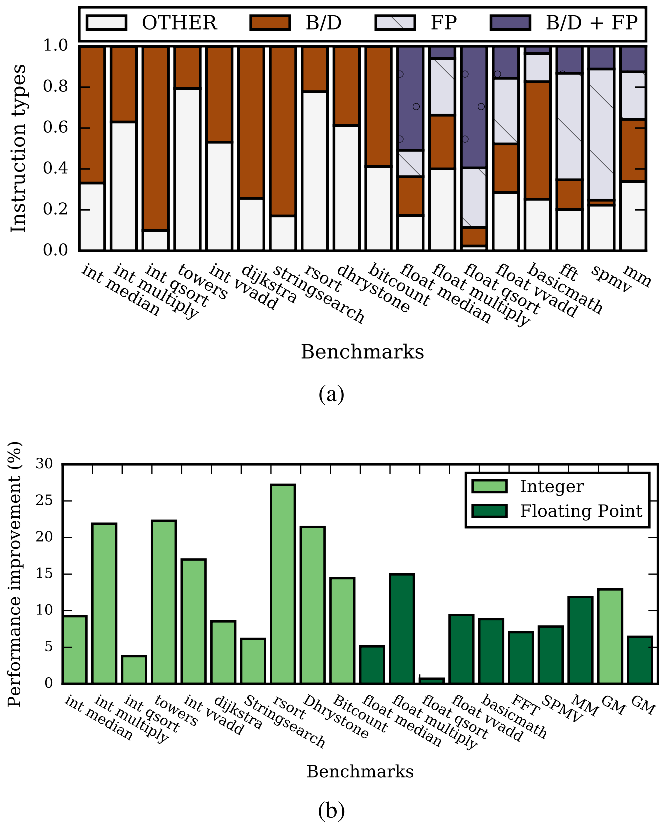


Fig. 7. Waveform showing the change in clock frequency when the muxsel signal goes high.

delay inside the core when branch logic is active, when it is inactive, and the maximum reg-to-reg delay of its neighbors B and C. Similarly, delay lines for region B correspond to the maximum reg-to-reg delay within region B and of the neighbor A and for region C correspond to maximum reg-to-reg delay within C and of neighbor A. To desynchronize the rocket core, we implement the three asynchronous controllers, all the delay lines, C-elements, and multiplexers shown in Fig. 6(c). The active bit from the designed FSMs are used as the select signals for the multiplexers. This results in dynamic scaling up of frequency when instructions being executed on the RISC-V core are integer arithmetic/load-store instructions which do not involve floating point, multiply, divide, or branch calculation hardware, as shown in the waveform in Fig. 7.

To create the asynchronous controller, we design the buffers, delay lines, and multiplexer in CHP language and translate them to the production rules manually as described in [17]. After thorough testing using our in-house production rule simulator, we combine them together and modify the RISC-V rocket core to use the asynchronous control to provide clock for different regions. The signals from the designed FSMs are then connected to the asynchronous control. This joint timed simulation of asynchronous circuits with synthesized register-transfer level (RTL) is facilitated by an integrated simulation tool developed internally that combines a custom asynchronous logic simulator with Synopsys VCS. The Standard Delay Format file produced by the design compiler was used to model the actual gate delays in the design. To make sure that one does not have to rely on our in-house production rule simulator or integrated simulation tool, we redesigned all the controllers by rewriting the production rules in Verilog and used Synopsys VCS to test the entire design.

For all our experiments, we used the timing reports generated after the synthesis using Design Compiler. However, as the placement and routing can change the delays of various paths, in a complete flow the proposed desynchronization technique should be applied using post place and route timing reports. Maintaining the delay values is important for the proposed desynchronization technique to work and hence to achieve that a three-step placement approach can be adopted. In the first step, the design should go through place and route and the timing reports should be obtained. These timing reports should be used to determine the desynchronization regions. In the second step, the synchronous design should go through the place and route again, but this time asynchronous controllers with empty cells for delay lines should be placed next to each desynchronization region. Local clock tree networks should also be generated to distribute the clock within different desynchronization regions. In the last step, delay values should be obtained based on the timing reports from the second step and delay lines should be placed in the empty cells. Since the entire circuit except the delay lines is placed and routed in the second step and then delay lines are placed accordingly in the third step, this three-step placement mitigates convergence issues.



(a)

(b)

Fig. 8. Instruction breakdown and performance improvement for various benchmarks in RISC-V and MiBench suites. (a) Instruction breakdown: B/D and FP are the fraction of times only branch/mul–divide instructions and only floating point instruction are in the pipeline, respectively. B/D + FP is the fraction when both branch/mul–divide and floating point instructions are in the pipeline, OTHERS represent the remaining instructions. (b) Percentage improvement in performance for various benchmarks.

We test and evaluate our design on all the benchmarks provided in the RISC-V test suite and some from the MiBench embedded benchmark suite [22]. Fig. 8(a) shows as a fraction of number of cycles how many times a branch or mul/divide or a floating point instruction is being executed in any of the processor pipeline stages and Fig. 8(b) shows the performance improvement for various benchmarks. It can be seen from Fig. 8(b) that the performance improvement of int-multiply, towers, radix sort, and dhrystone are all >20%. This is because these benchmarks use integer data types and hence do not use floating point instructions and also have fewer branches and divide instructions. Integer vector-vector add, bitcount, float multiply, and matrix multiplication achieve a performance improvement of 12%–17%. This is because the first two benchmarks use integer data types and are not very heavy in branches, the last two on the other hand use floating point data types but still have fair fraction of instructions which are neither floating point nor a branch or divide. Int median, int quicksort, dijkstra, and stringsearch,

TABLE I

DATA SET SIZES FOR DIFFERENT BENCHMARKS

Benchmark	Suite	Data-type	Dataset size
Median	RISC-V	float	400
		int	400
Multiply	RISC-V	float	100
		int	100
Qsort	RISC-V	float	2048
		int	2048
Towers	RISC-V	int	7 discs
Vvadd	RISC-V	float	300
		int	300
Dijkstra	MiBench	int	100 nodes
Stringsearch	MiBench	int	60
Rsort	RISC-V	int	2048
Dhrystone	RISC-V	int	size 50, runs 500
Bitcount	MiBench	int	7500 iters, 7 funcs
Basicmath	MiBench	float	–
FFT	MiBench	float	32
Spmv	RISC-V	float	1000 × 1000, density 0.01
Matrix-mult	RISC-V	float	6 × 6

TABLE II

GATE COUNT FOR THE RISC-V PROCESSOR PIPELINE, DESIGNED FSM, AND ASYNCHRONOUS CIRCUIT

Logic	Number of Gates
FSM	103
Async Control	150
RISC-V Core + FPU	79,116

all achieve a performance improvement less than 10% as all of these had lots of forward and backward branches. Float median, float quick sort, float vector-vector add, float basicmath, fast Fourier transform (FFT), and sparse matrix-vector multiplication, all had a lot more floating point instructions as compared to other floating point benchmarks which resulted in a performance improvement of less than 10%. Overall, we achieve a geometric mean improvement of 12.9% for integer benchmarks and 6.5% for floating point benchmarks. From Fig. 8(a) and (b), it can be seen that the benchmarks which have less number of floating point, branch, and multiply/divide instructions achieve more performance improvement. Table I shows the size of data sets for different benchmarks.

In terms of area, gate counts for both synchronous processor and asynchronous control are provided in Table II. The table shows that additional number of gates required to attain this performance improvement is just 0.3% and is very small compared to the original RISC-V design.

We performed a thorough power estimation of both synchronous and desynchronized designs using the value change dump files produced by our simulations and the time-based power analysis in Synopsys Prime Time. Fig. 9 shows the power consumption of synchronous and desynchronized pipelines. Since increasing frequency at the same voltage to get higher performance also increases the power consumption, the desynchronized version is consuming more power than synchronous design. From Fig. 9, it can be seen that the increase in power is as high as 20%–30% for int-multiply, int-vvadd, towers, rsort, and dhrystone benchmarks, for which our technique provides the best performance results as well. For the benchmarks with

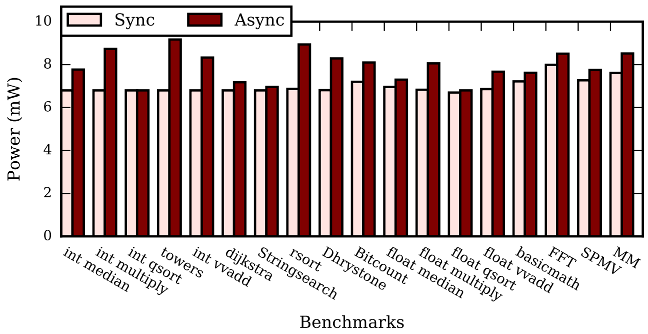


Fig. 9. Power consumption of synchronous pipeline and desynchronized pipeline.

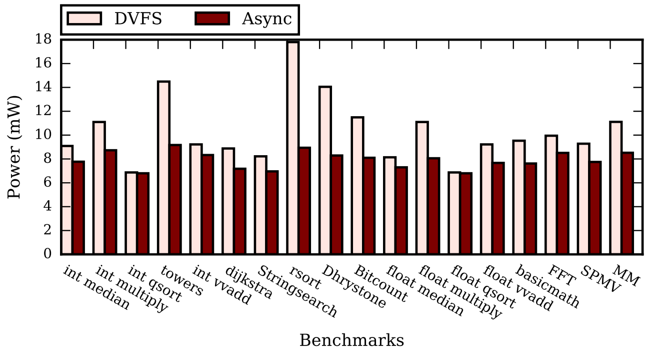


Fig. 10. Power consumption of synchronous pipeline and desynchronized pipeline, where the synchronous pipeline is running at a higher voltage and frequency (DVFS state) to match the performance of desynchronized pipeline for each benchmark.

low-performance improvement, the power consumption also does not increase much.

To perform a fair comparison between the synchronous and desynchronized version, for each benchmark, we used the dynamic voltage and frequency scaling (DVFS) voltage–frequency curve as in [23] to scale up the voltage and frequency of synchronous design and estimate the power consumption when both synchronous and desynchronized designs achieve the same performance, as shown in Fig. 10. Here, the desynchronized circuit is operating at 1-V supply voltage and clock frequencies of 555, 588, and 770 MHz (corresponding to 1.8-, 1.7-, and 1.3-ns timing paths). For synchronous design, as different benchmarks required different voltages and frequencies in order to match the performance of the desynchronized design, voltage, and frequency numbers for all the benchmarks were not provided. The original supply voltage and clock frequency over which DVFS scaling was performed are 1 V and 555 MHz. To compare the synchronous and desynchronized designs for energy efficiency which is defined as the number of instructions executed per unit μJ, we used the power consumption from Fig. 10 and execution time to determine the energy consumption for desynchronized design and the synchronous design running at higher frequency and voltage. We used the dynamic instruction count and the energy consumption to determine the energy efficiency for the two designs as shown in Fig. 11. As one can see, desynchronized design is more energy efficient than the synchronous design

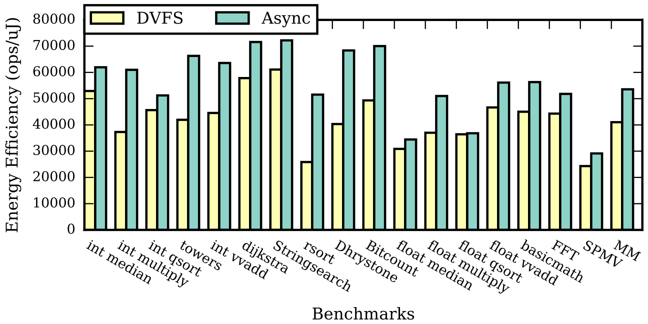
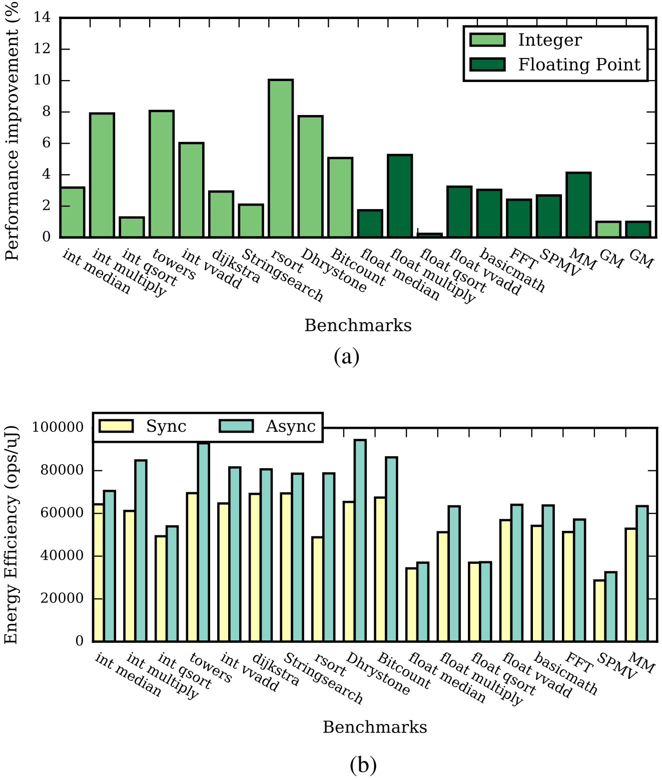


Fig. 11. Energy efficiency (#dynamic instructions/energy) of synchronous pipeline and desynchronized pipeline, where the synchronous pipeline is running at a higher voltage and frequency (DVFS state) to match the performance of desynchronized pipeline for each benchmark.



(a)

(b)

Fig. 12. Performance and energy efficiency comparison of synchronous pipeline and desynchronized pipeline where desynchronized pipeline is running at a lower voltage and frequency to have the same power consumption as the synchronous pipeline. (a) Performance improvement of the desynchronized design running at lower voltage and frequency to achieve the same power as synchronous design. (b) Energy efficiency (#dynamic instructions/energy) comparison of synchronous design and desynchronized design running at lower voltage and frequency to achieve the same power consumption as synchronous design.

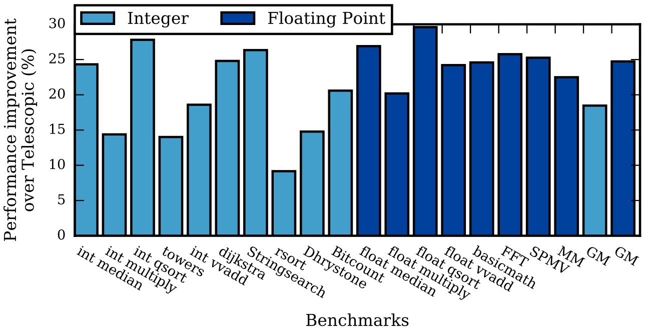
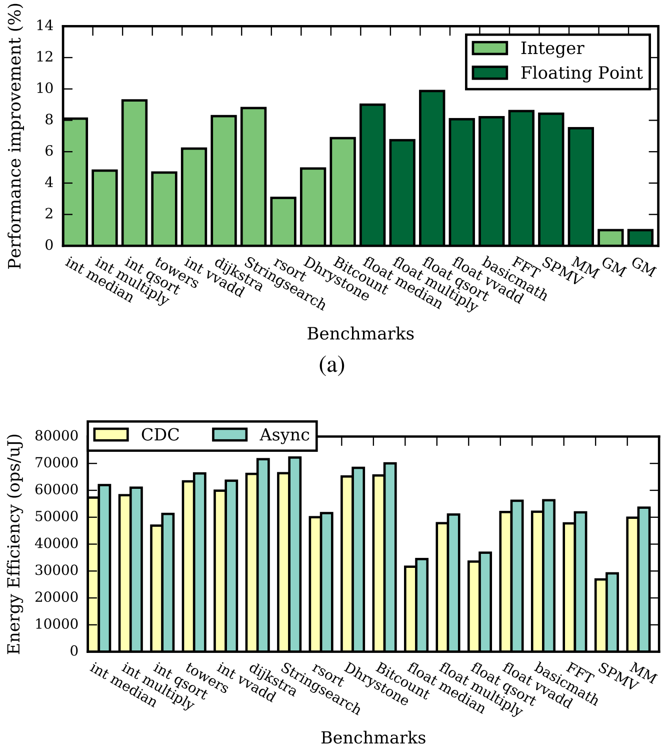


Fig. 13. Performance improvement over Telescopic Units.

running at higher voltage and frequency to achieve the same performance.

In order to compare the synchronous and desynchronized designs under the same power consumption, we scaled down the voltage and frequency of the desynchronized design using [23] and estimated the performance and energy efficiency of the two designs. Fig. 12(a) shows the performance improvement of desynchronized design running at lower voltage and frequency over the synchronous design with the same power consumption. Here, synchronous design is operating at 1-V supply voltage and 555-MHz clock frequency. Fig. 12(b) shows the energy efficiency for both synchronous design and the desynchronized design running at lower voltage and frequency. It can be seen that under the same power consumption, desynchronized design has a higher performance and energy efficiency than the synchronous design. For all our experiments, power consumption does not include clock tree network. As clock is one of the major components of power consumption in synchronous designs (around 40%) [24], we expect that the desynchronized design will have lower power consumption than the synchronous design.

To compare our proposed approach against the previous work of telescopic units [11], we evaluated the performance improvement of all of the benchmarks using telescopic units and compared it with the performance improvement of our proposed approach. From Fig. 13, it can be seen that our proposed approach is able to outperform telescopic units in all the cases. The reason behind this is quite simple-since telescopic units runs the design at a high frequency and uses two cycles instead of one for slow operations, it can increase the time taken for a slow operation and if the application is full of operations that require the slow clock, it can cause a negative impact on the performance. Telescopic units is known to improve overall throughput only when the long latency paths have low occurrence probability [11] which is not true for these benchmarks. The proposed approach, on the other hand, uses the exact amount of delay that would be required for any operation and hence does not over or under estimate the delay requirements. To compare our design against Varipipe [12], we realize that Varipipe has the potential to achieve the same performance as our proposed design. However, it uses a centralized clock generator, and delay lines from all units have to be combined centrally, whereas our scheme is distributed and only requires the interaction between delay lines of communicating blocks. Furthermore, a centralized clock generator also does not solve the issues of spatial–temporal variations and EMI in the clock and lacks the key benefits of desynchronization, as shown in [1]. To compare our proposed methodology with multiple clock domains or CDC, we realize that CDC assigns different clocks to different regions of the circuit and use synchronizers to communicate signals between different clock domains.



(a)

(b)

Fig. 14. Performance and energy efficiency comparison of desynchronized design over clock domain crossing where regions A, B, and C in Fig. 6 are put in different clock domains and are connected using synchronizers. (a) Performance improvement of the desynchronized design over clock domain crossing. (b) Energy efficiency (#dynamic instructions/energy) of synchronous design with three clock domains and desynchronized pipeline.

CDC is good for the designs where the communication between different clock domains is less. However, for the processor design such as RISC-V core, keeping FPU and main core in different clock domains is not beneficial as the FPU can be frequently used by many benchmarks and adding synchronizer increases latency of FPU which in turn decreases the throughput. For example, if an instruction is dependent on a floating point instruction whose latency increases due to the overhead of synchronizers, the performance of the system will suffer. However, for an approach such as ours, there is no overhead of synchronizers. Furthermore, CDC requires intrusive design changes, for example, if there is some combinational logic on the path from one register in one clock domain to another register in another clock domain, a register has to be placed after the combinational logic to avoid the probability of failure due to glitches [25]. Due to the potential problems of metastability, the integration of different clock domains is considered much more difficult compared to the integration of various asynchronous modules [24]. Fig. 14(a) and 14(b) show the performance improvement of the desynchronized RISC-V core over the design with multiple clock domains. The design with multiple clock domains consists of three clock domains for each of the regions A, B, and C in Fig. 6. A two-cycle synchronizer delay is modeled between any two clock domains. As we can see the desynchronized design outperforms design with multiple clock domains and also has a better energy efficiency. Dynamic time borrowing is another technique where registers are split into latches and these latches are moved around to balance the combinational path delay in each stage, so that the circuit achieves the average-case performance instead of worst case performance. However, dynamic time borrowing adds a lot many timing constraints which sometimes makes the synthesis process more difficult [26], while our proposed approach does not add to the complexity of the synthesis process. In dynamic time borrowing, for any latch in the system, the data must arrive in time to be properly captured and as time for data arrival is a function of all the previous pipeline stages, a worst case data arrival analysis is done which requires calculating the timing of all the stages that share a path. The complexity grows for the circuits that have loops [26]. To compare our proposed approach against [4], we realize that this paper focuses on desynchronizing a synchronous circuit with negligible area overhead and potential power consumption reduction. However, it achieves the same performance as the synchronous counterpart. This paper is orthogonal to our work since it does not focus of dynamically skipping the critical paths which are not active, however, the technique in [4] combined with our technique can provide potential savings in power and high performance at the same time. TonyChopper [27] provided a desynchronization package that uses the standard cell libraries and reduces the leakage power. However, it results into a desynchronized circuit which could be 2.4× slower than synchronous baseline and can have an area overhead as large as 2.4× which is very high compared to 0.3% area overhead in our proposed approach.

Although we have shown the application of the proposed desynchronization technique in the context of microprocessors, it can also be applied to complex synchronous circuits, especially those generated through high-level synthesis (HLS). HLS takes source program in languages such as C, C++, and OpenCL and generates hardware circuits in the form of HDLs for both FPGAs and ASICs. This high-level specification many times consists of if-else branches and HLS generates circuit for both true and false branches. The circuit for these branches becomes active based on the dynamic branch condition. If both the branches have different critical paths, one can think of using the branch condition to determine the active state of the two branches and put the branches in different desynchronization regions. For circuits designed with RTL, if certain hardware block is conditionally used, then the designer can provide a signal indicating the active state of the block and the proposed technique can be used for dynamically scaling the frequency to achieve better performance.

V. Conclusion

In this paper, we discussed the reasons that restrict traditional desynchronization techniques from improving the performance of synchronous designs. We proposed a novel operation-dependent desynchronizationtechnique, which uses the information about active modules in a synchronous design and dynamically scales frequency to achieve a higher throughput. As achieving a higher frequency for every part of the design is essential to improve overall throughput, the proposed technique provides a way to combine various designs built under different constraints without sacrificing performance.

REFERENCES

[1] J. Cortadella, A. Kondratyev, L. Lavagno, and C. P. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 25, no. 10, pp. 1904–1921, Oct. 2006.

[2] N. Andrikos, L. Lavagno, D. Pandini, and C. P. Sotiriou, "A fully-automated desynchronization flow for synchronous circuits," in *Proc. 44th ACM/IEEE Des. Automat. Conf.*, Jun. 2007, pp. 982–985.

[3] D. H. Linder and J. H. Harden, "Phased logic: Supporting the synchronous design paradigm with delay-insensitive circuitry," *IEEE Trans. Comput.*, vol. 45, no. 9, pp. 1031–1044, Sep. 1996.

[4] F. Bertrand, A. Cherkaoui, J. Simatic, A. Maure, and L. Fesquet, "CAR: On the highway towards de-synchronization," in *Proc. 24th IEEE Int. Conf. Electron. Circuits Syst. (ICECS)*, Dec. 2017, pp. 339–343.

[5] A. Smirnov, A. Taubin, and M. Karpovsky, "Automated pipelining in ASIC synthesis methodology: Gate transfer level," in *Proc. 13th Int. Workshop Logic Synth. (IWLS)*, 2004.

[6] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An ASIC flow for GHz asynchronous designs," *IEEE Des. Comput.*, vol. 28, no. 5, pp. 36–51, Sep./Oct. 2011.

[7] R. Manohar, "Systems and methods for performing automated conversion of representations of synchronous circuit designs to and from representations of asynchronous circuit designs," U.S. Patent 7 610 567 B2 Oct. 27 2009.

[8] F. Akopyan, C. Otero, and R. Manohar, "Hybrid synchronous-asynchronous tool flow for emerging VLSI design," in *Proc. Int. Workshop Logic Synth.*, 2016.

[9] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," in *Proc. 8th Int. Symp. High Perform. Comput. Archit.*, Feb. 2002, pp. 29–40.

[10] F. Mendes, T. Curtinhas, D. L. Oliveira, H. A. Delsoto, and L. A. Faria, "A novel tool for synthesis by direct mapping of asynchronous circuits from extended STG specifications," in *Proc. 17th Int. Conf Embedded Syst. (VLSID)*, Jan. 2018, pp. 451–452.

[11] L. Benini, E. Macii, M. Poncino, and G. De Micheli, "Telescopic units: A new paradigm for performance optimization of VLSI designs," *IEEE Trans Comput. Aided Des. Integr. Circuits Syst.*, vol. 17, no. 3, pp. 220–232, Mar. 1998.

[12] N. Toosizadeh, S. G. Zaky, and J. Zhu, "VariPipe: Low-overhead variable-clock synchronous pipelines," in *Proc. IEEE Int. Conf. Comput. Des.*, Oct. 2009, pp. 117–124.

[13] J. Xu and H. Wang, "Desynchronize a legacy floating-point adder with operand-dependant delay elements," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2011, pp. 1427–1430.

[14] M. Singh and S. M. Nowick, "MOUSETRAP: High-speed transition-signaling asynchronous pipelines," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 6, pp. 684–698, Jun. 2007.

[15] A. Saifhashemi, D. Hand, P. A. Beerel, W. Koven, and H. Wang, "Performance and area optimization of a bundled-data intel processor through resynthesis," in *Proc. 20th IEEE Int. Symp. Asynchronous Circuits Syst.*, May 2014, pp. 110–111.

[16] A. Davare, K. Lwin, A. Kondratyev, and A. Sangiovanni-Vincentelli, "The best of both worlds: The efficient asynchronous implementation of synchronous specifications," in *Proc. 41st Des. Automat. Conf.*, Jul. 2004, pp. 588–591.

[17] A. J. Martin, "Synthesis of asynchronous vlsi circuits," Dept. Comput. Sci., California Inst. Technol., Pasadena, CA, USA, Tech. Rep. Caltech-CS-TR-93-28, 2000.

[18] S. M. Burns, "Performance analysis and optimization of asynchronous circuits," California Inst. Technol., Pasadena, CA, USA, Tech. Rep. Caltech-CS-TR-91-01, 1991.

[19] A. Moreno and J. Cortadella, "Synthesis of all-digital delay lines," in *Proc. 23rd IEEE Int. Symp. Asynchronous Circuits Syst. (ASYNC)*, May 2017, pp. 75–82.

[20] Y. Chen, R. Manohar, and Y. Tsividis, "Design of tunable digital delay cells," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, May 2017, pp. 1–4.

[21] K. Asanović *et al.*, "The rocket chip generator," Dept. Elect. Eng. Comput. Sci, UC Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-17, 2016.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. 4th Annu. IEEE Int. Workshop Workload Characterization*, Dec. 2001, pp. 3–14.

[23] *TI-DVFS*. Accessed: May 17, 2018. [Online]. Available: http://www.ti.com/lit/an/slva646/slva646.pdf

[24] M. Krstic, E. Grass, and X. Fan, "Asynchronous and gals design-overview and perspectives," in *Proc. New Gener. CAS (NGCAS)*, Sep. 2017, pp. 85–88.

[25] S. Churiwala and S. Garg, *Clock Domain Crossing (CDC)*. New York, NY, USA: Springer, 2011, pp. 73–89.

[26] V. G. Oklobdzija, V. M. Stojanovic, D. M. Markovic, and N. M. Nedovic, *Digital System Clocking: High-Performance and Low-Power Aspects*. Piscataway, NJ, USA: IEEE Press, 2003.

[27] Z. Wang, X. He, and C. M. Sechen, "TonyChopper: A desynchronization package," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des. (ICCAD)*. Piscataway, NJ, USA, IEEE Press, Nov. 2014, pp. 446–453.

Nitish Srivastava (S'18) received the B.Tech. degree from the Department of Electrical Engineering, IIT Kanpur, Kanpur, India, in 2014. He is currently working toward the Ph.D. degree at the Department of Electrical and Computer Engineering, Cornell University, Ithaca, NY, USA.

His current research interests include spatial hardware for high-performance computing, FPGAs, and circuit design.

Rajit Manohar (M'98–SM'10) received the B.S., M.S., and Ph.D. degrees from the California Institute of Technology, Pasadena, CA, USA, in 1994, 1995, and 1998, respectively.

From 1998 to 2016, he was a Faculty Member at the Cornell University, Ithaca, NY, USA, where he was a Stephen H. Weiss Presidential Fellow. Since 2017, he has been a Faculty Member at Yale University, New Haven, CT USA, where his group conducts research on the design, analysis, and implementation of self-timed systems and he is currently the John C. Malone Professor of Electrical Engineering and a Professor of Computer Science. He founded the Computer Systems Laboratory at both Cornell University and Yale University. His current research interests include the design and implementation of a number of self-timed VLSI chips including the first high-performance asynchronous microprocessor, the first microprocessor for sensor networks, the first asynchronous dataflow FPGA, the first radiation hardened SRAM-based FPGA, and the first deterministic large-scale neuromorphic architecture.

Dr. Manohar was a recipient of the NSF CAREER Award, the nine best paper awards, and the nine teaching awards. He was named to MIT Technology Review's top 35 young innovators under 35 for contributions to low-power microprocessor design.