University of Rhode Island
Dept. of Electrical and Computer Engineering
Kelley Hall
4 East Alumni Ave.
Kingston, RI 02881-0805, USA

Technical Report No. 032000-0100

Achieving Typical Delays in Synchronous Systems via Timing Error Toleration

Augustus K. Uht
Department of Electrical and Computer Engineering
University of Rhode Island

Email: uht@ele.uri.edu
Web: www.ele.uri.edu/~uht

March 10, 2000

This work has been submitted for publication.

Abstract

This paper introduces a hardware method of improving the performance of any synchronous digital system. We exploit the well-known observation that typical delays in synchronous systems are much less then the worst-case delays usually designed to, typically by factors of two or three or more. Our proposed family of hardware solutions employs timing error toleration (TIMERRTOL) to take advantage of this characteristic. Briefly, TIMERRTOL works by operating the system at speeds corresponding to typical delays, detecting when timing errors occur, and then allocating more time for the signals to settle to their correct values. The reference paths in the circuitry operate at lower speeds so as to always exhibit correct values (worst-case delays). The nominal speedups of the solutions are the same as the ratio of worst-case to typical delays for the application system. The increases in cost and power dissipation are reasonable. We present the basic designs for a family of three solutions, and examine and test one solution in detail; it has been realized in hardware. It works, and exhibits substantially improved performance.

This work was partially supported by the National Science Foundation through grants: MIP-9708183, DUE-9751215, EIA-9729839, by an equipment grant from the Champlin Foundations, by software donations from Mentor Graphics Corporation, Xilinx Corporation, Virtual Computer Corporation, and equipment donations from Virtual Computer Corporation. Patent applied for.

1 Introduction and Background

Ever since synchronous digital systems were first proposed, it has been necessary to make the operating frequency of a system much less than necessary in typical situations to ensure that the system operates correctly assuming worst case conditions, both operating and manufacturing. The basic clock period of the system is padded with a guard band of extra time to cover extreme conditions. There are three sources of time variation requiring the guard band. First, the manufacturing process has variations which can lead to devices having greater delay than the norm. Second, adverse operating conditions such as temperature and humidity extremes can lead to greater device delays. Lastly, one must allow for the data applied to the system to take the worst delay path through the logic.

However, none of these extremes is likely to be present in typical operating conditions. The only known method to still obtain typical delays in all cases is to change the basic model to an *asynchronous* model of operation[4]. But this is undesirable: asynchronous systems are notoriously hard to design, and there are few automated design aids available for asynchronous systems.

This paper proposes a family of *TIMing ERRor TOLeration* synchronous digital systems, or TIMERRTOL, to realize typical delays using standard synchronous design methodologies. Our methods of doing this will increase the performance of any synchronous digital system commonly by a factor of two or more, assuming the system runs under typical operating conditions (e.g., temperature, altitude) and is a typical product of the manufacturing process. Of course, our solutions function correctly even if the typical constraints are not met. The implementations dynamically adapt to achieve the best performance possible under the actual operating or (prior) manufacturing conditions. The cost varies from an increase of greater than the performance factor increase to significantly less than the performance factor. Cycle time need not be impacted. Power dissipation increases by about the same as the performance factor up to the square of the performance factor increase, across the implementation family. In the case of our physical example, the power dissipation is much less than the latter pessimistic limit.

This means that virtually every digital device design today could be operated twice as fast as it is now. In general, devices would have to be redesigned, but the process is conceptually straightforward.

We have designed an example of one of the implementations and realized it in a Xilinx FPGA (Field Programmable Gate Array). Although it is desirable to perform chip fabrication as well, FPGA realization gave us great flexibility in experimentation, being able to rapidly change the design and quickly evaluate it. FPGAs are also becoming mainline realization platforms, given such features, as well as easy upgrade, etc.

The realized adder is a 32-bit adder operating at a frequency about twice that of a baseline FPGA adder. It is likely that this could be improved upon. Although the nominal cost and power increases can be quite high in the style of implementation employed, the adder application lent itself to much less additional hardware and power dissipation. It remains to be seen if this will be a common phenomenon.

The paper is organized as follows. A review of synchronous system timing is given in Section 2. In Section 3 the basic ideas of timing error toleration are presented, including our family of three solutions or implementations. Section 4 describes our realization of a high-performance 32-bit adder for an FPGA using the third solution. Our experimental methodology is described in Section 5, with the experimental results presented in Section 6. Other related work is discussed in Section 7. We conclude in Section 8.

2 Timing Background

Digital circuits that compute a result based solely on the state of the circuits' current inputs are said to be constructed of *combinational* logic. Combinational systems can be used in many applications, but for any interesting digital system to be realized the system must base its output on both current inputs and the system's prior outputs or state.

There are two types of digital systems with state. The first type, *asynchronous* digital systems, change state as soon as an input changes its value. Modeling, designing and verifying asynchronous systems has in practice been found to be extremely difficult, even with modern asynchronous techniques. Further, there is substantial cost and performance overhead with asynchronous systems[3]. Hence, asynchronous digital systems are rarely used. This is unfortunate, because asynchronous systems operate as fast as the logic delays will allow.

Virtually all digital systems today are *synchronous* systems. In these systems, the state only changes at times determined by a global system clock (that is, in *synchronism* with the clock). For example, if we consider a 500 MHz Intel Pentium III processor, its basic on-chip (CPU) clock oscillates 500 million times a second; the processor will only change its state at the start of one or more of those oscillations. Since a designer (and the machine) is thus only concerned with the state at instants of time, rather than over a continuous period of time, as in the asynchronous approach, the synchronous approach makes the design, construction and use of digital systems highly straightforward and reliable (at least as far as the hardware is concerned).

All synchronous digital systems can be represented by the model shown in Figure 1. The two components to the system are the *Combinational Logic (CL)* and the *Flip-Flops or latches (FF)*.

The latches hold the current or *Present State (PS)* of the system. Each latch typically stores one bit of information, having a value of 0 or 1. A flip-flop only changes its contents or state when a clock signal makes a transition (say 0-to-1). The same clock goes to all latches; clock signals typically oscillate at Megahertz frequencies.

The logic has no clock input or feedback loops: a change in one of its inputs propagates to one or more outputs with a delay due only to electrical circuit and speed-of-light constraints. A latch also has a propagation delay, but from the clock transition to a change in its output.

The system operates by using the logic to compute the Next State (NS) of the system from its present state and the current values of the inputs to the system. The next state is then stored in the latches when the clock rises, and the process repeats. In order for the system to function properly, the computation must propagate through the logic and appear at the inputs to the latches before the relevant transition of the clock occurs at the latches.

So far so good: if one knew the exact delays through the logic and latches, the clock frequency could be set to the inverse of the sum of the delays, and the system would operate at peak performance (as measured by computations per second). However, the delays are not constant, but vary with differences in the manufacturing process, variations in the power

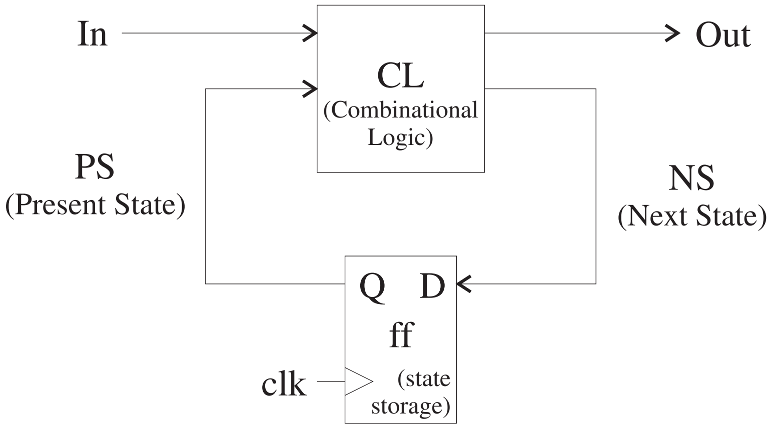


Figure 1: *Standard digital system.* All synchronous digital systems can be modeled by this diagram.

supply voltage, variations in the operating temperature and humidity, variations in the input data, as well as other factors. As a result of these wide variations, and the necessity to guarantee the operation of the digital system in the worst-case situation (e.g., temperature extremes), the clock period must be set to a higher value (lower performance) than is necessary in most, typical cases. Consequently, the average user will experience significantly lower performance than is actually necessary, perhaps half as much or less.

TIMERRTOL gets around this reduction in performance, allowing speeds corresponding to the actual delays (usually typical) in the digital system, by increasing the speed (frequency) of the clock until one or more errors occur, then backtracking to a known good state, discarding the erroneous computation, and resuming operation from there. If the error rate gets too large, the operating frequency is reduced to a value resulting in an acceptable error rate. The adjustment of the operating frequency can be done statically (fixed at system design time), or dynamically, as the system operates; the latter is preferred. The dynamic case requires special circuitry; it is discussed later in this document.

3 Timing Error Toleration: TIMERRTOL

3.1 The Crux of the Timing Error Toleration Idea

The basic idea is to perform a digital computation with a lower than worst-case-required clock period (faster). At the same time, perform the same computation with a larger, worst-case-assumed, clock period (slower) on a second system with identical hardware. At a later time, compare the two computations. If there is a difference in the two answers, the faster computation must be in error, a *miscalculation* has occurred, and the digital system uses the answer from the slower system.

The question arises, aren't we then limited by the speed of the slower system, and have gained nothing? No, because we actually have <u>two</u> copies of the slower system; thus, although they each run half as fast as the main system, they still produce results in the aggregate at the same rate as the main system, which is running at a much faster rate than possible without TIMERRTOL. Hence we have improved performance, albeit with more hardware.

The rest of this section is organized as follows. The first, *motivating*, solution in the TIMERRTOL family is described, following the description above. An alternative solution is next described, using less hardware and power; in this, the *proportional* solution, the hardware cost and power consumption are proportional to the nominal performance improvement factor. It is usable with pipelined systems. The last solution is then given, the *sub-proportional* solution, which has cost growing slower than the nominal performance increase, although the power may grow quadratically; it is also applicable to any digital system, not just a pipelined one. The last subsection gives an overview of how the clock speed would be controlled in such systems.

3.2 A Motivating TIMERRTOL

The first solution will serve to motivate the discussion and present the basic operating ideas in a circuit which is easy to understand, though requiring much hardware and power. More pragmatic solutions appear in following sections.

This solution is described as it can be realized at the gate and latch level. Realizations at other levels such as with entire systems are straightforward extensions of these ideas.

The motivating solution is shown in Figure 2, with its corresponding timing diagram in Figure 3. The basic idea is to run two additional copies of the system each at half the speed of the main system, one copy replicating the results of the main system in odd cycles and the other in even cycles. The two half-speed systems are operated one main system cycle out-of-sync with each other. Both of the half-speed systems's outputs are compared with the main system outputs in alternate cycles; if there is a difference between the two sets of outputs, an error is detected, and the main system's outputs for that cycle are replaced with those (correct) of the comparing half-speed system. One cycle of operation is lost for every correction necessary; this is called the *miscalculation* penalty.

Referring to the timing diagram, the first three cycles of operation are for the case when no errors occur. The numbers within the individual signals' timing charts indicate which computation the signal is working on or holds at that time. At the end of cycle three (at the asterisk), a comparison of CL.0 (half-speed) with Q_{sys} indicates an error in computation 3. The system then stalls one cycle, with the next state remaining at 3 in cycle 3 (see (3)), which it gets from CL.0, having the correct version of computation 3, and the system resumes operation with the correct result. In cycles 3 and later the ideal computation numbers are shown without parentheses, and the actual (with miscalculation delay) computation numbers are shown with parentheses.

This solution is for the case when performance is to be increased no more than a factor of two from the performance in the original, worst-case delay system. The half-speed systems must not be operated faster than the original worst-case system speed in order to provide a guaranteed error-free computation to compare the high-speed main computation with. This solution requires more than three times the hardware of the original system, and has quadruple the power dissipation. The cycle time of the system is also negatively impacted with the addition of the multiplexors to the critical path.

It is possible to modify the solution so as to allow performance increases greater than

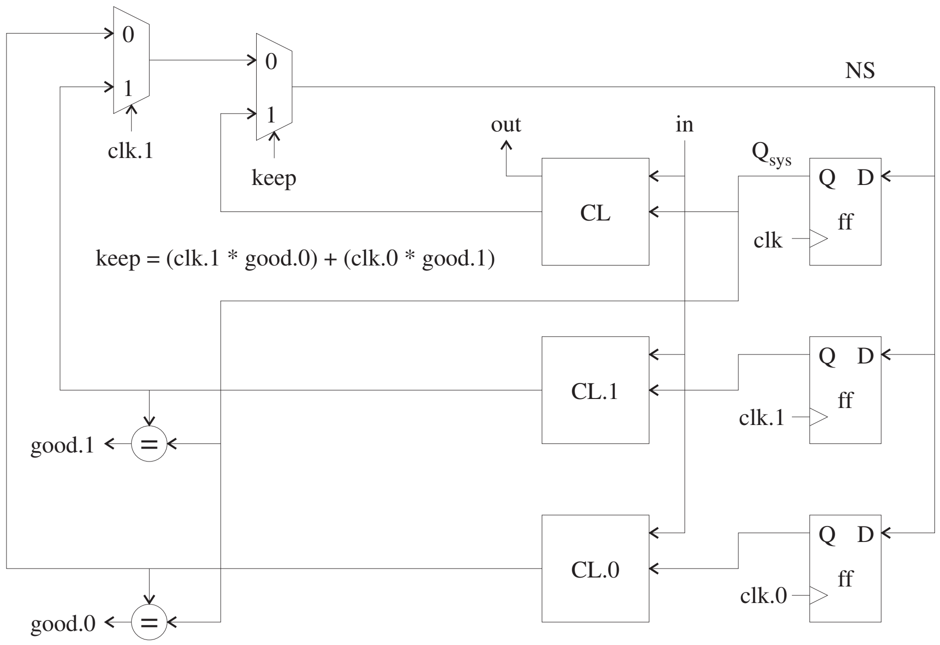


Figure 2: *General digital system employing timing error toleration (TIMERRTOL).* The top combination of combinational logic and flip-flops is the original system, operated at system frequency. The two copies of the original are below the original, each copy operates at one-half the system frequency; see Figure 3 for the details of the timing.

a factor of two. For each increment of factor increase, e.g., increment of one from X2 to X3, another copy of the hardware must be used. Further, the slow comparison systems use a clock an increment of factor slower, e.g., in the X3 performance increase case, the now third-clock systems operate at a third of the frequency of the main system clock. For each increment of factor increase, the miscalculation penalty increases by a cycle, e.g., for the X3 case, the penalty is two cycles. Other cases are handled accordingly. Note that all of the clocks in the overall system are synchronized.

3.3 Second Solution: Performance Proportional to Hardware Used

It is actually not necessary to have three copies of the hardware, as used in the first solution. In fact, the original copy, that operated at the system frequency, can be eliminated. It is also not necessary to use any multiplexors. Thus, the hardware cost approximately doubles for a doubling of performance. For a tripling of performance, the cost triples, and so forth.

This *proportional* solution is also easier to build and does not increase the amount of logic (gate delay) in the critical path. This solution is applied at the functional or register level.

The proportional solution is shown in Figure 4, with a representative timing diagram in Figure 5. In the block diagram, a higher level system than in the motivating solution is assumed. In the proportional solution, we assume that the system is *pipelined*; this is common in current digital systems.

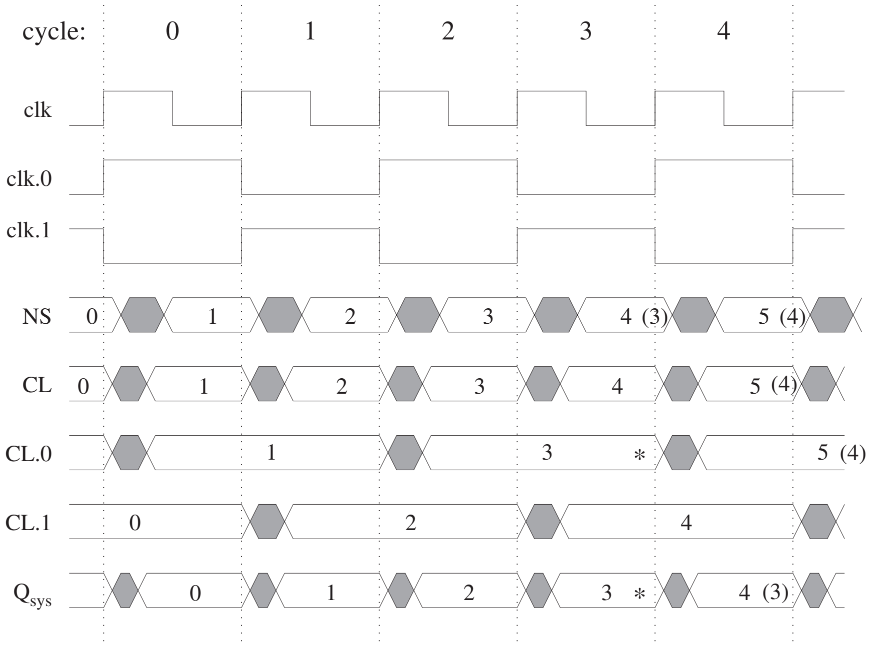


Figure 3: *TIMERRTOL timing.* This is the basic timing of the TIMERRTOL system of Figure 2. The two half-speed clocks are skewed by one system clock cycle. The non-"cycle" numbers enumerate the computation being performed by a set of combinational logic at a given time. The delay through two system "clk" cycles is used for the basic clock period of the low-speed and checking systems, CL.0 and CL.1; this larger delay is made equal to the worst-case delay of the original system.

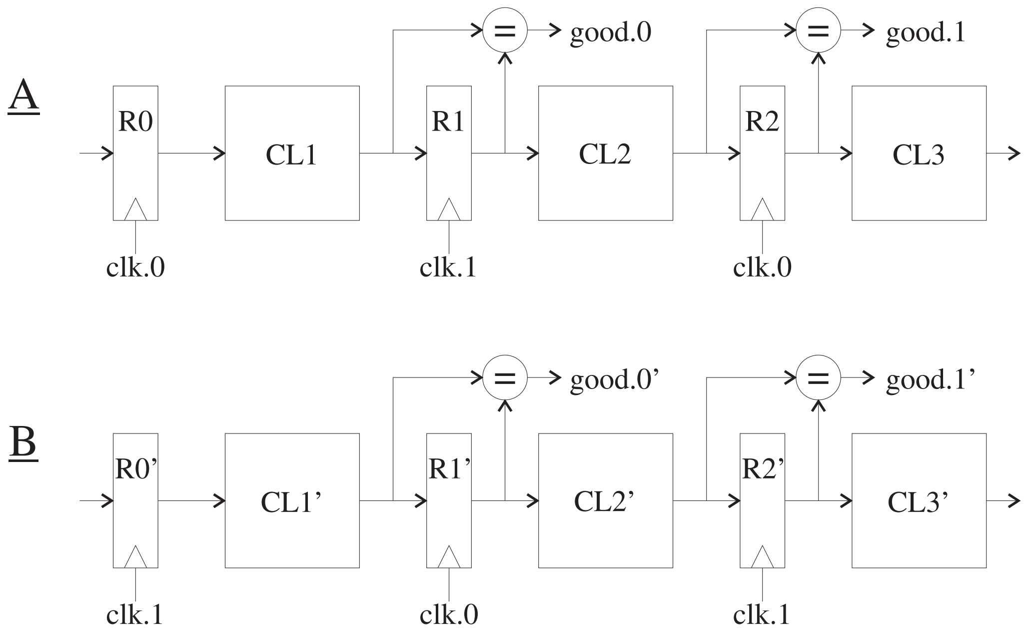


Figure 4: *Pipelined digital system employing timing error toleration (TIMERRTOL) with proportional hardware cost.* The solution uses two identical copies of the original system, adding comparators, and clocking adjacent stages on alternate system clock cycles. The two copies use complementary clocks at corresponding stages.

We first describe the system's operation from an intuitive viewpoint. We take the virtual or implicit *system* clock to be at twice the frequency of the actual clk.0 and clk.1. Typically the system clock would run at twice the frequency as the original non-TIMERRTOL system clock. As a system, therefore, results are coming out twice as fast as before. Inputs alternate between pipe A and pipe B, as do system outputs. clk.0 and clk.1 are 180 degrees out of phase with each other: the pipes operate in a non-uniform fashion, even stages clocked at different times than odd numbered stages.

Let's now look at a single pipeline, pipe A. First, note that the time allowed for signals to go from R0 through CL1 into R1 is the same as the system clock period; thus, CL1 is operating at the full improved speed. However, the inputs to CL1 do not change for another system clock period, until the next rising edge of clk.0. Therefore, at the next edge of clk.0, the current output of CL1 (not held in R1) has had two system clock cycles to settle, i.e., it has had the worst-case propagation time allowed to it and thus it can now be used as the guaranteed correct answer, and is compared with the output of R1 (which only had one cycle to settle) to see if the latter is correct or not. At the same time CL2 has been computing its result based on the faster one cycle computation of CL1. Thus, at the second rising edge of clk.0, two things are true: we know if the output from R1 is correct and we have the result of the next stage's computation ready (from CL2). Finally, similar things go on in pipe B, and since pipe B is out of phase with pipe A, results come out of the entire system at a rate nominally twice as fast as the original system clock speed.

The detailed operation is as follows. Assume that the hardware shown in the diagram

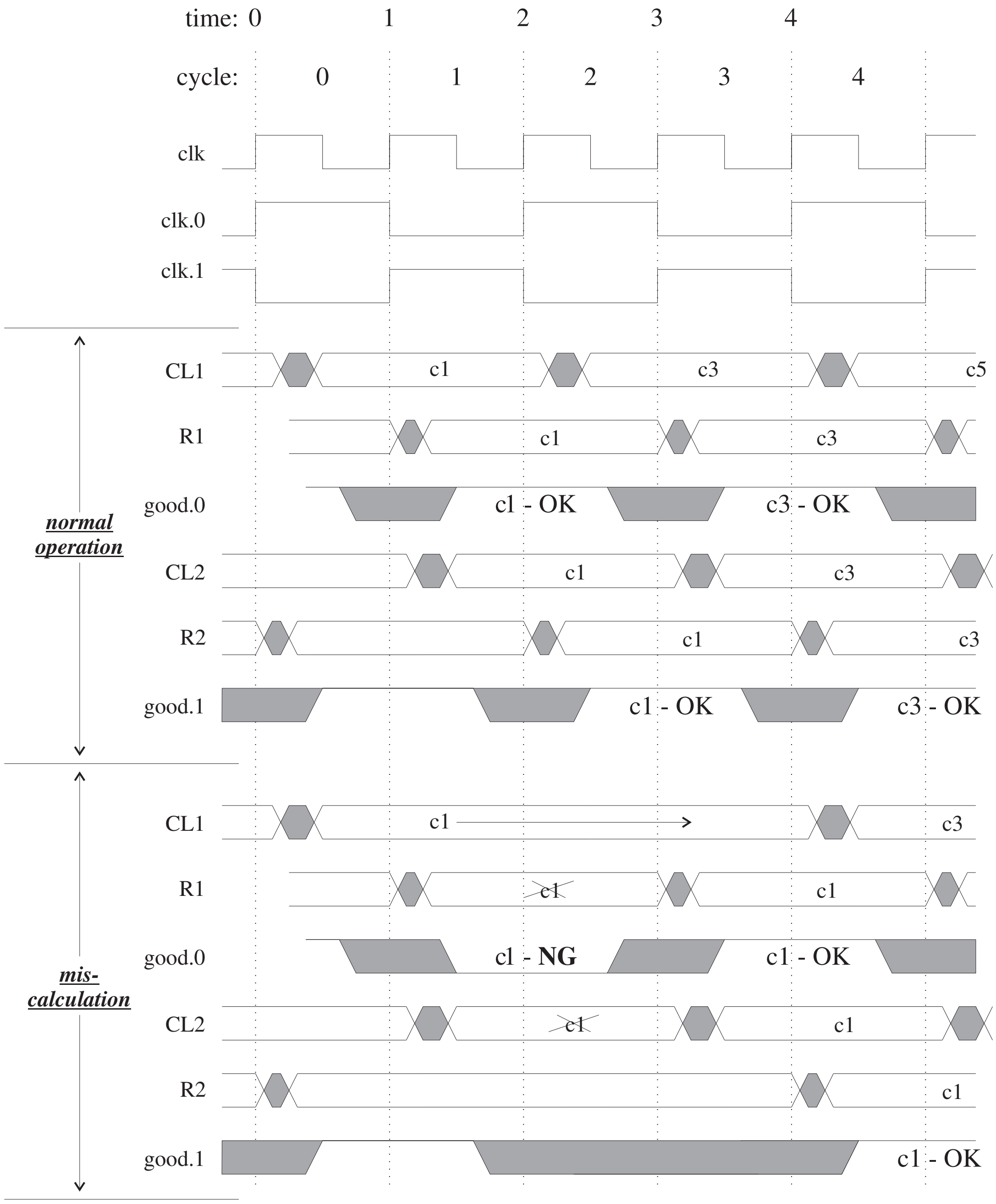


Figure 5: *Proportional TIMERRTOL timing.* This is the basic timing of the proportional TIMERRTOL system of Figure 4. NOTE: only the A pipeline is shown. The two half-speed clocks are skewed by one system clock cycle. The top diagram shows the timing when no errors occur; the bottom shows the timing when an error has been detected at the output of R_1, in pipe A. Computations are labeled "c#". c2 and c4 are in pipe B, not shown.

is part of the system's overall pipeline. The primed (') hardware is a copy of the unprimed (top) hardware. Inputs to the overall system come in at the system clock rate. Note that as least as far as this hardware is concerned, there is no actual clock operating at the full rate. The inputs go to each pipeline in alternate cycles. At time 0, an input is latched into R0 by clk.0. The first computation occurs in Combinational Logic block CL1, and is latched one system cycle later at time 1 into R1 by clk.1. As before, clk.0 and clk.1 run at half the rate of the system clock. Therefore the computation in CL1 as latched in R1 takes 1 system cycle. However, CL1 does not have its inputs changed until time 2. At the end of the second cycle, the output of R1 (one cycle computation time) is compared with the current output of CL1 (two cycles of computation time, hence the guaranteed correct answer). If the two results, slow one and fast one, are equal (good.1 is true) then the fast computation is correct and no action need be taken. At time 2 the output of the second computation, from CL2, is latched into R2. Similar operations happen in the rest of the pipeline A stages, as well as in pipeline B. Results leave pipeline A (and B) at a rate one-half of the system clock rate, where the system clock rate is twice as fast as the system clock rate without the solution. However, there are two pipelines, so results are produced at $0.5*2*2 = 2$ times the rate of the original system. So far, no miscalculations have been assumed, the normal situation.

If a miscalculation occurs, we then have the timing of the lower diagram. In this case, R1 has latched incorrect results from CL1. This is detected at the end of time 2 (good.1 is false). CL2 thus also has an incorrect answer, therefore clk.0 is disabled for all of pipeline A at time 2. CL1 is still computing the same result for the original inputs, and therefore at time 3 R1 latches in the correct result from CL1. CL1 has now had more than two cycles to compute its result, which is thus correct. This correct CL1 result is now in the pipeline, and normal high-throughput operation resumes. The miscalculation penalty is two system clock cycles for pipeline A. Overall, this could lead to a system miscalculation penalty of 1 cycle, but if we require that the outputs from the two pipelines be in order, pipeline B must also be stalled by two system cycles, and hence we assume the penalty is two cycles for a miscalculation in the proportional solution.

If typical delays are one-third the original system's worst-case delays, and we thus would like to improve performance by a factor of three, a third copy of the system would be needed, with three clocks running at a third of the system clock rate, which is itself running three times faster that the original system clock. Note that the power required to operate the new system also increases proportionally to the performance increase; of course, it is not good to use more power, but it is expected. The miscalculation penalty also increases proportionally to three cycles.

One added feature of the proportional solution over the base TIMERRTOL solution is the elimination of the multiplexors. This allows a faster clock, or rather, does not increase the delay through a stage.

Note that the hardware cost does actually more than double, since we need to add the comparators. It is still much less hardware than the motivating TIMERRTOL solution.

Comment on Implementation: Implementing the proportional solution is potentially complicated due to its use of pipelining. Note that as described above the two pipelines are independent, i.e., no computation in one pipe depends on a computation in the other pipe.

Processor pipelines do not typically follow this assumption, in that intermediate computations may be sent back to earlier stages. With the proportional solution, if the feedback is to a stage in the other pipe, design is more complicated.

Nonetheless, it is doable. We have designed a simple RISC processor employing data forwarding using the proportional solution. The main effect on the processor design is to approximately double the number of bypass paths needed in the original pipeline. Further, bypass paths not only go within a pipe A, but also across pipes (pipe A to B and B to A). We will report on our results with this processor in a later paper. It has not yet been tested.

3.4 Third Solution: Sub-Proportional-cost TIMERRTOL

The final solution, the sub-proportional solution, realizes 2x performance for <2x increase in hardware cost; power increases by at most 4x. A major feature is its applicability to all digital systems, via the general digital system model as presented earlier.

The third solution is applied directly to the elemental digital system of Figure 1. Referring to the top part of Figure 6 (above the dashed line), the basic idea is to create a mini-version of a proportional pipe, having its same error toleration characteristics, but construct the stages' combinational logic differently. Assuming the original combinational logic is CL, we now split it into two equal-delay sections, CLa and CLb, i.e., we increase the pipelining by a factor of two. This allows the clock frequency to be doubled. If we then apply the TIMERRTOL idea and use a two-phase clocking system, we can increase the implicit system frequency by another factor of two. However, since we only get a result every complete pass through the pipeline, that is every two system clock cycles, the overall performance increases by a factor of two.

In Figure 6 the logic below the dashed line is necessary to control the unit and handle errors accordingly. The first logic expression generates LDR.a, the synchronous load enable line for register Ra. This register is loaded when LDR.a is true and Ra's clock goes from 0 to 1. Therefore the register is loaded when either there was an error out of CLa, and CLa needs more time to compute its result, or when the prior stage produced a valid result without extra delay. The logic for LDRb is similar.

Referring to Figure 7, the timing of the sub-proportional TIMERRTOL is seen to be similar to the proportional TIMERRTOL. In the sub-proportional case, however, sequential results follow each other in the pipeline, and there is only one pipeline.

The cost potentially increases by less than a factor of two (sub-proportional increase): the number of registers doubles, and we need comparators, but the core combinational logic stays the same. The actual increase in cost is application-dependent.

In other metrics, the miscalculation penalty is two implicit system cycles, or one explicit cycle. Since the number of storage elements doubles, and their frequency doubles, the power consumption quadruples.

As with the proportional solution the performance of the sub-proportional-cost solution can be increased by increasing the number of sections of the system. For example, in order to increase the performance by a factor of three, the combinational logic would be split into three sections, each ending in a register clocked by one distinct phase of a three-phase clock. The cost would again increase potentially sub-proportionally. A three-phase system is used in our test hardware, discussed later.

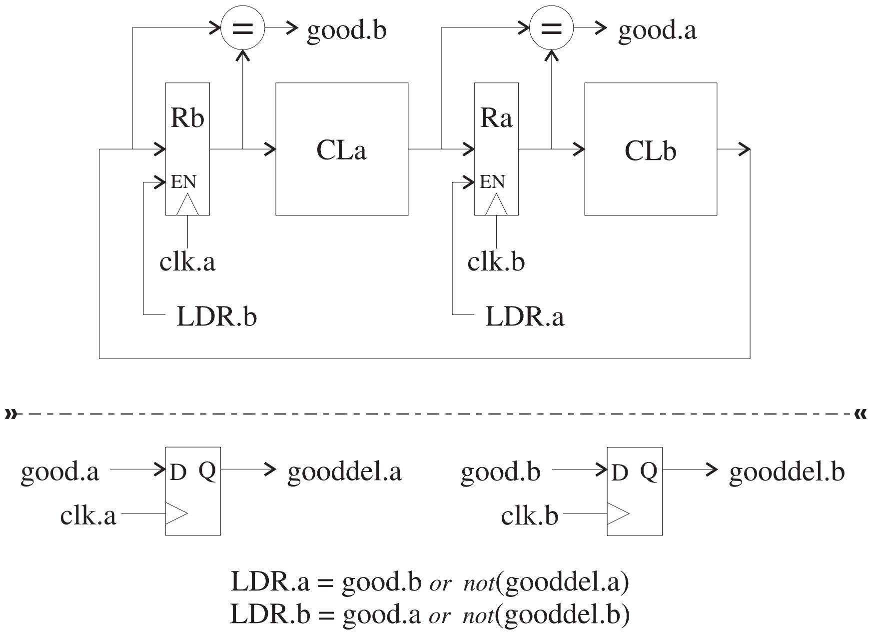


Figure 6: *General digital system employing timing error toleration (TIMERRTOL) with sub-proportional hardware cost.* The solution splits the single combinational logic block of the original system into two blocks, each with its own staging register, as in a pipeline, except the stages are clocked on alternate system cycles. Comparators are also used, but no multiplexors, as was the case in the proportional solution. The system clock frequency is 4x the original. The explicit (physically existing) stage clock frequencies of the solution are 2x the original system clock frequency.

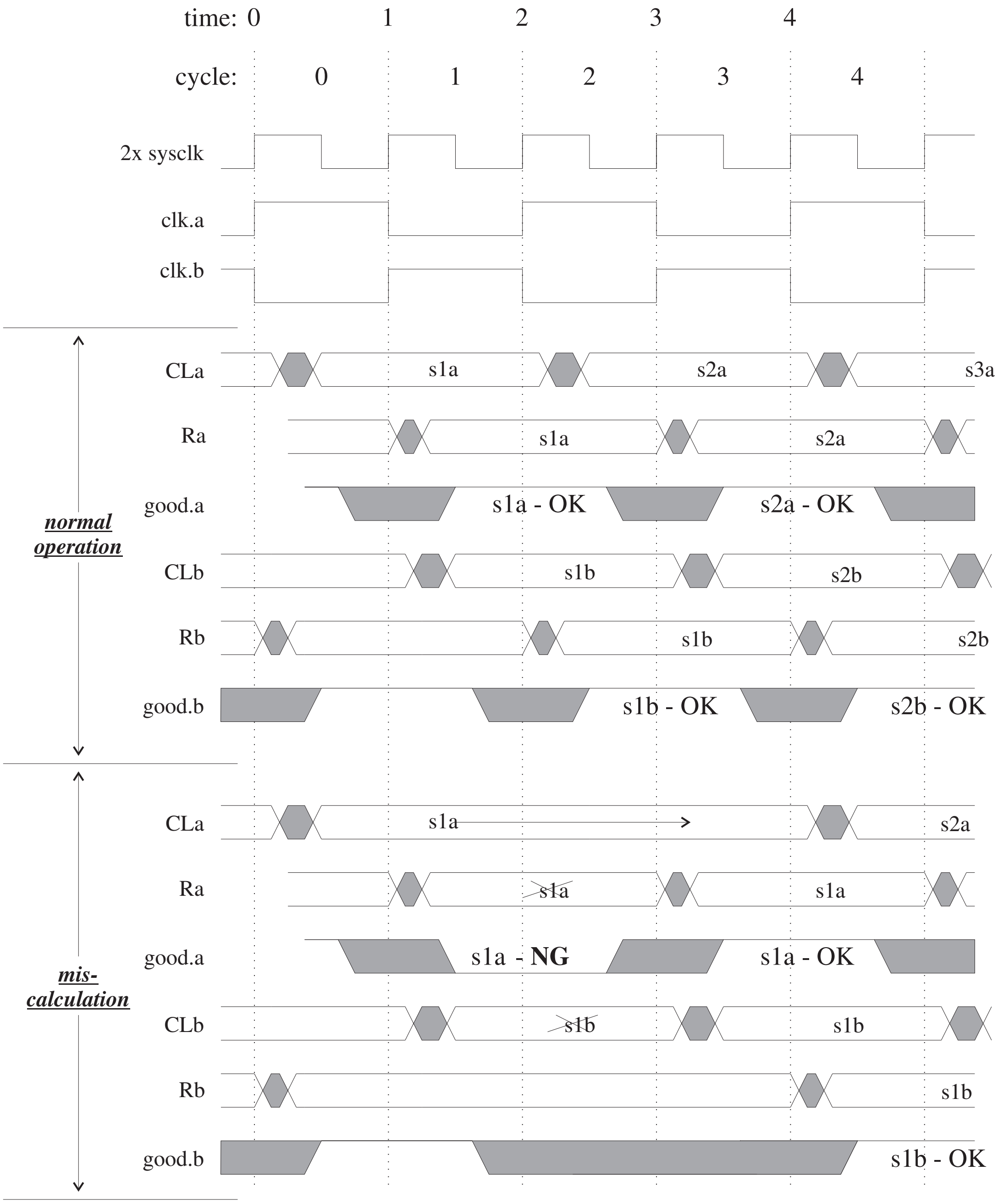


Figure 7: *Sub-Proportional TIMERRTOL timing.* This is the basic timing of the sub-proportional TIMERRTOL system of Figure 6. The two half-speed clocks are skewed by one implicit system clock cycle. The top diagram shows the timing when no errors occur; the bottom shows the timing when an error has been detected at the output of *Ra*. The nomenclature: "s1a" indicates that state 1, part a (the first half of the original state) is being computed.

3.5 Adaptive Performance Maximizing Controller

The problem is how to set the clock frequency to as high a level as possible. As the frequency increases, the basic performance of the system increases, but at some point the degradation in performance due to the miscalculation penalties from an increasing error rate will offset the basic (clock rate) performance, decreasing the performance overall. Thus, we need a device to find the maximum performance point, and we need one that will adapt to changing conditions to adjust the system and have it adapt appropriately, so as to always find the best performance given the actual operating conditions and manufacturing conditions.

The solution is to apply control theory to the adjustment of the system clock frequency in real time, as the system works. The basic operation of such a system would be biased towards increasing the clock rate. At the same time, it would have input from the comparators of the timing error detection circuitry. The system clock drives a counter having a clock enable function. The counter is only disabled when an error is detected (in the case of our performance doubling example, this is for one cycle per error). The overall absolute averaged count rate of this counter is thus a direct measure of the system's performance; as errors increase, it will count less often, although at a faster rate - the same dynamics as those of TIMERRTOL's performance.

The smoothed output of the counter is fed back into the system's clock generator, adjusting the frequency of the clock appropriately. If the averaged counter output is low, it increases the clock frequency (and the counter output will also increase) until the averaged counter output begins to decline; the frequency is then incrementally lowered, increasing the counter output, until the output starts to decline again, at which point the frequency reverses course once again. Put another way, the frequency of the clock increases while the derivative of the performance (integrated counter output) increases; when the latter decreases, the clock frequency is decreased; when the performance begins to increase again, the clock frequency is once again increased.

This kind of system is readily designable using standard control theory.

4 Realization of a High-Performance 32-bit 3-phase Sub-Proportional TIMERRTOL Adder for an FPGA

We designed a self-contained 32-bit test adder using the methods of the sub-proportional TIMERRTOL. It has been realized on an FPGA. The adder could be plugged into any system using a registered adder; additional clocks might be needed, as discussed in Section 3.4. The basic design of the adder is as shown in Figure 8, basically a three stage version of Figure 6. Each stage of the adder is driven by one phase of a 3-phase clock.

Xilinx utilizes special carry paths which make ripple-carry adder realizations the fastest for the FPGA up to about 32-bit long adders[2]. Our results would apply for any kind of adder.

Each of the solution's three stages contains the logic for about 11 consecutive bits of the ripple-carry adder. The carry-out of one stage's adder is pipelined into the carry-in of the next stage. Each stage computes 10-11 bits of the sum output. We sought to obtain a 3x improvement in performance.

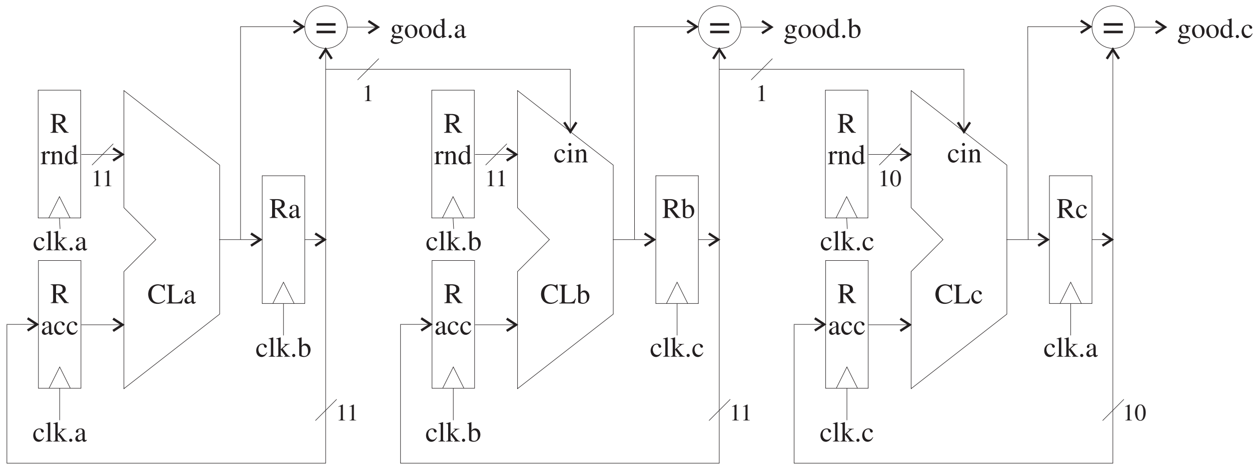


Figure 8: *32-bit ripple carry adder realized with 3-stage sub-proportional TIMERRTOL design.* Each stage contains about 1/3 of a 32-bit ripple carry adder. Only the carry out from an adder section is propagated to the next stage. The registers 'Rrnd' are connected as a feedback shift register for pseudo-random number generation; the interconnections are not shown.

The cost of the test adder is much less than the nominal sub-proportional figures would indicate. The baseline unmodified 32-bit adder requires the same overall combinational logic (combinational adder itself). In a real system at least two 32-bit registers for the inputs (64-bits of registers total), and in some cases an additional 32-bit register for the output would be needed, although in a pipelined system the output register would be counted as part of the next stage. The sub-proportional adder uses 92-bits of registers total and three ten- or eleven-bit comparators. Making a rough assumption that a bit's-worth of comparator costs the same as a 1-bit register, the total hardware cost for the sub-proportional adder is 125-register-bit equivalents. Including the combinational logic (no change), this is less than twice the original cost, much less than the nominal speedup factor (3x). The power dissipation increases by a factor of about 2.5, given the small increase in register bits and the achieved performance increase; this is much less than the nominal increase in power dissipation of a 3-phase sub-proportional system.

For ease of experimentation, the adder was configured as an accumulator, with both adder inputs changing at the same time. This was done so that we could let the adder free-run for many iterations without interaction with and delays from the host. A Linear Feedback Shift Register arrangement was used for the non-accumulator input, employing the generating polynomial: $1 + x^3 + x^{31}$; [8]. The shift register is initialized at the beginning of every run with a C-library generated random number as a seed. However, the adder could be used with completely independent inputs, anywhere a normal registered adder could.

The adder has been physically designed, "constructed" (downloaded) and tested. It works, detecting errors and then obtaining corrections. The entire process proved to be substantially trickier than expected. Our main problem was that the design was mainly limited not by the speed of the 11-bit adder segments, but by the comparators used to check the results. It took many iterations with the Xilinx M1 tools via varying constraints and tweaking the design to obtain a satisfactory result. The adder exhibited a substantial performance speedup over the baseline adder, but not the 3x hoped for. A characteristic of the sub-proportional solution is that unlike the proportional solution, it adds pipeline stages. The delays through these extra registers detracted from the potential performance achievable.

5 Experimental Methodology

All of our experimental work was performed on a Xilinx XC4020E-2-HQ208 Field Programmable Gate Array. The FPGA is contained on an EVC1 virtual computer card made by Virtual Computer Corporation. The EVC1 is mounted inside a Sun SparcStation 1 (143 MHz Ultra Sparc) and is connected to the Sun's general purpose I/O bus, the SBUS. The SBUS is synchronous and operates at 25 MHz. The EVC1 is equipped with a software-settable variable frequency oscillator (360 KHz to 120 MHz). This was extremely useful in measuring performance. We heavily modified software drivers provided to us by Virtual Computer Corp., and used them to communicate with designs downloaded to the FPGA and to run the experiments.

The FPGA has many advantages: it can have its internal wiring and logical structure altered an unlimited number of times; special design programs are used to create the low-level settings for such a device that realize the desired logic functionality of a new digital system. Further, pre-designed high-level functions are available from the device manufacturers that can be combined in arbitrary ways to allow easy construction of complex digital systems on the FPGA.

The designs were entered in the Mentor Graphics Renoir VHDL synthesis tool. Exemplar's Galileo synthesized the VHDL into Xilinx FPGA primitives. These primitives were then combined, mapped, and placed and routed by the Xilinx M1 FPGA design tool (Version 1.5i). Xilinx Logiblox macros were heavily used.

Our software driver allowed us to make individual or multi-pass measurements. One of the latter was a bisection algorithm used to find the maximum operating frequency of the unit under test (UUT). For both sets of experiments, the hardware (in the FPGA) contained circuitry that checked on the correctness of results; the results were also computed on the host (the Sun) and checked with the raw results coming back from the UUT.

6 Experimental Results: Performance Potentials and Actuals

Our first set of experiments sought to validate the basic TIMERRTOL ideas and stage construction by examining the operation of a basic 32-bit adder. The second set of experiments investigated a test adder: a real 3-phase 32-bit sub-proportional adder. We realized it, verified its operation, and measured its performance; a baseline non-phased adder was also examined for purposes of comparison.

6.1 Experimental Verification of the Ideas of TIMERRTOL

Using the hardware and software described in Section 5, we built a complex piece of combinational logic and tested its operation as would happen in our solutions.

The function realized is a 32-bit adder in isolation (not in one of our solutions). The inputs to the adder come from registers using the same clock. There are also two registers on the output of the adder. The first is loaded exactly one cycle after the input registers to the adder are loaded with test data. The second is loaded exactly two clock cycles after the inputs are loaded. A comparator compares the outputs of the first and second output registers, hence at times differing by one cycle. There are two one-bit registers on the comparator output, to save (sample) the comparison output at different times. Thus, we have modeled all of the major basic elements of the solutions. For each event, two random numbers are applied to the inputs of the adder at the same time. The output of the adder is latched both one and two clock cycles later. By adjusting the clock frequency and looking at the output register results and the comparator results, we can see when the adder produces correct results and if correct/incorrect operation is detected by a slower system (the second register, which gives the adder twice the time to compute its result). The overall system is driven and examined by a host computer, which further verifies the additions.

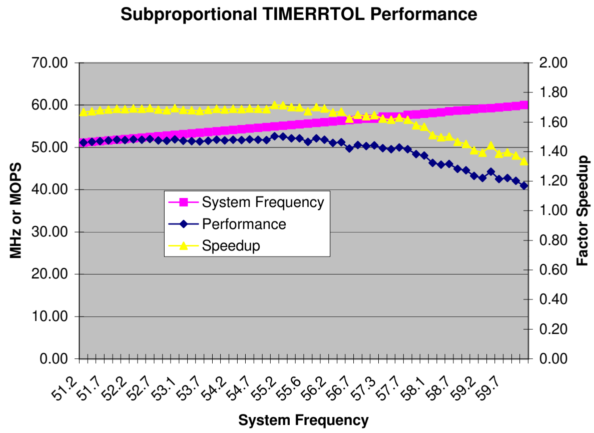
The primary experiment seeks to determine the maximum frequency that the system can operate at without error, or rather, with very few (all tolerated) errors. As a base frequency, we use the results of the design tools which tell us that the adder (in the system, that is, including register delays) can operate at about 30 MHz (30 million adds per second) assuming worst case conditions. That corresponds to a clock period of about 33 nanoseconds.

The experiment consists of a number of passes. Each pass consists of performing twenty different additions on random numbers at one operating frequency. The system is initialized to a low frequency. As previously mentioned, the clock oscillator is variable from about 360 KHz to 120 MHz. The host computer sets the frequency. Using the bisection algorithm mentioned above, it quickly finds the highest operating frequency with no errors among the 20 additions.

After the first run, we found the operating frequency to be about 60+ MHz. However, certain aspects of the data led us to believe that the system could actually be operated faster; the comparator was actually too slow. We re-ran the experiment giving the comparator more time to operate (but still looking at the two output registers clocked at the original times). The operating frequency increased to about 95 MHz. Thus, we can potentially realize about a factor of three improvement in adder performance with TIMERRTOL.

One of the key contributors to synchronous adders' propagation delay is the worst-case carry propagation delay through the adder. However, with typical input data sets this hardly ever happens. In fact, with 20 sets of random input data, the maximum carry propagation length is only about seven bits. TIMERRTOL is able to take advantage of this situation and decrease the actual time allowed for typical additions. This phenomenon occurs in digital circuits in general, that is, it usually does not take the worst case number of gate propagation delays for a signal to fully propagate through a circuit.

From this set of experiments we conclude that the potential of TIMERRTOL is great, at least for common-sized addition operations.



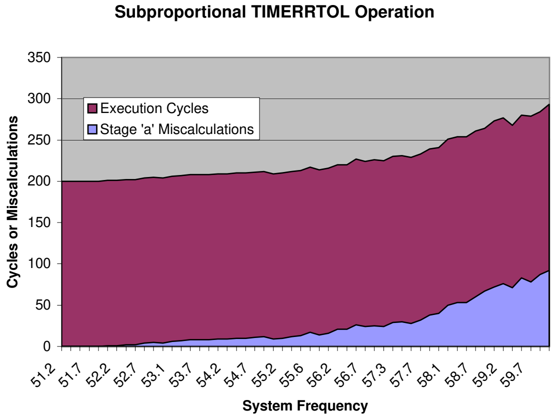


Figure 9: *TIMERRTOL characteristics.*

6.2 Evaluation of the 3-Phase Sub-Proportional Adder

Our test case consisted of running the Adder Under Test (AUT) for 200 iterations per run or system frequency setting, always using the same seed. The resultant sum was checked by the software. The adder was also instrumented with a set of three counters, one per stage, each one counting detected errors in its stage. The tolerated errors should lead to an increased cycle count, but still a correct sum. One other adder was used in the test system, to count the overall number of cycles that the adder was active during an experimental run; its count included miscalculation penalty cycles.

Baseline Data: The 32-bit single-phase adder had a design maximum frequency of 30.67 MHz, corresponding to a worst-case period of 32.6 ns; this assumes a 20% safety margin. This data came from the post-place-and-route timing analysis section of the M1 tool.

The baseline adder was able to run as fast as 81.91 MHz (81.91 million 32-bit adds per second) without error. This is slower than seen in the first set of experiments; it is due both to variations in the automated design results and to requiring correct data over 10 times as many iterations as in the first set of experiments.

3-phase Adder Data: Our data is presented in Figure 9. The system frequency was increased from about 51 MHz to over 60 MHz. At each frequency setting 200 additions of random data were performed by the sub-proportional adder. The right-hand 'Operation' chart shows the total number of system clock cycles necessary to perform the 200 additions, including miscalculation cycles, and the number of miscalculations themselves. At lower frequencies there are no miscalculations, and the total computation takes exactly 200 cycles. At about 52 MHz errors in stage 'a' start to be generated, detected and tolerated; stages 'b' and 'c' do not exhibit errors. Each error results in a one system cycle miscalculation penalty. As the frequency continues to increase, more errors occur during the computations and thus the number of total execution cycles also increases.

Looking at the left-hand chart, 'Performance', the overall performance in MOPS (millions of addition operations per second) is plotted. Also shown is the speedup over the baseline adder (equal to sub-proportional performance divided by the baseline performance), and the system frequency plotted against itself, giving a straight line. The latter is provided to show the trajectory the sub-proportional adder's performance would take if no miscalculations were to occur.

For all data shown, the system tolerated (removed) all of the errors and produced the correct sum. Above 60 MHz, the adder failed as a system, producing untolerated errors. Note that the frequency safety margin is great: about 8 MHz or 15% between error detection and system failure.

Roughly, the performance of the sub-proportional adder is seen to increase steadily below a system frequency of 52 MHz and then ever more slowly with increasing system frequency, up to a peak performance of 52.6 MOPS at a system frequency of 55.0 MHz. At the latter frequency there were nine miscalculations in the 200 additions. Therefore there is an overall improvement in performance of the 3-phase sub-proportional TIMERRTOL adder over the baseline adder of a factor of 1.72 or 72%. This is not the 3x nominal desired, but is good for our first attempt.

It may be possible that we were too aggressive and optimistic in the choice of a 3-phase adder; a 2-phase unit might have performed better. The layout on the FPGA of the baseline adder by the M1 design tool was also substantially better than its layout of the 3-phase adder; more comparable layouts would have likely led to better gains.

7 Other Related Work

To our knowledge, no one has taken an approach anything like ours. The closest work we are aware of is [11]. In this work a microcontroller has been modified so that it can self-tune its clock for "maximum" frequency. It does this by periodically pausing computation for up to 68 cycles, during which time it forces extreme inputs (1 and all 1's) into the ALU. (The ALU has the longest critical path.) The output of the adder is checked: if it is correct, the frequency is increased; if incorrect, the frequency is decreased by a safety margin, at which time the computation resumes. This scheme takes advantage of some attributes of typical delays, but not those coming from typical data. It also must pause operation to perform its tuning. Further, it can not recover from any timing errors introduced by its self-tuning. Therefore TIMERRTOL is more robust and higher-performing than this scheme.

There has been a large amount of work on asynchronous systems. See, for example, [7] for a description of the first asynchronous microprocessor and [4] for a brief tutorial on modern asynchronous circuit design. Modern asynchronous design techniques either use much more hardware than synchronous ones (*self-timed* circuits) or are very hard to design (*delay matching*) [13].

There have been many methods created to improve the performance of synchronous circuits. The main approach is to *retime*[6] the registers or latches so as minimize the worst case necessary clock period. This is done by a variety of methods, including moving the registers or latches in the circuit. Software pipelining has been applied to synchronous digital circuits to generate optimal clocking schemes[1]. However, worst case delays between storage elements must still be maintained.

Multisynchronous systems[5] have also been proposed in which the circuitry on a chip is divided into semi-autonomous modules, each with its own clock. All of the clocks have the same frequency, but may be out of phase. This addresses part of the worst case timing problem, but only at the system level, handling part of the chip clock drive problem.

Wave pipelined arithmetic units have been proposed, but have implementation difficulties[3, 10], including the inability to easily stall the pipeline, since it depends on time-of-flight data storage (like a mercury delay-line). The design of such devices is also difficult; it is hard to ensure that signals arrive at the same time.

In one existing method used in some laptop computers, the temperature of the processor is measured and fed back to control (throttle) the operating frequency. This only adjusts for one parameter and usually the frequency is not increased above the nominal operating frequency. In [9] a control technique is given that does allow the frequency to improve. However, it is an open-loop system: errors are not explicitly detected, and in one variation the temperature is not measured, just estimated. The TIMERRTOL approach subsumes many of the benefits of such systems and can take advantage of more of the typically-valued parameters in a system.

In [12] a hybrid synchronous/asynchronous system is proposed having an on-chip clock generator whose frequency tracks changes in operating temperature and voltage. Therefore the system is able to partially take advantage of typical operating and manufacturing conditions. However, it is an open-loop system: errors are not detected; this limits its effectiveness. Further, the system is unable to take advantage of typical data sets in its synchronous sections.

8 Conclusions

Timing error toleration allows synchronous digital systems in general to operate potentially twice or more faster than in their current embodiments. This is done without changing the basic structure of the existing digital system. The proposed system adapts to existing environmental conditions, pre-existing manufacturing characteristics and actual system data, always obtaining the best performance possible.

This is achieved by operating digital systems without assuming worst-case conditions. In most cases, digital systems today must be operated assuming worst-case conditions, which is overly conservative and results in much worse performance than what could be realized assuming typical (actual) conditions. Typical conditions can only be used if errors are detected and removed. Our designs actually tolerate errors in the digital system: part of the hardware runs at full speed, and part runs at much lower speed, at a speed guaranteed to give correct results. The outputs are constantly compared to detect an error; when one occurs, the correct answer is substituted for the incorrect one, and normal high-speed operation resumes. The operating frequency is adjusted to maximize performance, balancing high clock frequencies with low-enough error rates. The output of the overall system is always correct, there are no errors presented to the user of the system.

A prototype test adder was constructed and tested, demonstrating the functionality of our approach as well as substantial performance gains. One of the key areas of future work is to devise and codify design guidelines and design rules for such systems, to ease their use. Another area is the design of fast comparators, or the development of an alternative for error detection.

We also plan to test our proportional TIMERRTOL CPU and report on the results. Further, it is desirable to actually build and test the feedback control system. However, no surprises are expected from that study since the control system design is not complicated.

References

[1] F. Boyer, M. Aboulhamid, Y. Savaria, and I. Bennour. Optimal Design of Synchronous Circuits Using Software Pipelining Techniques. In *Proceedings of the 1998 International Conference on Computer Design*, 1998.

[2] Xilinx Corporation. *1999 Xilinx Data Book*. Xilinx Corporation, San Jose, Calif., 1999. http://www.xilinx.com/partinfo/databook.htm.

[3] M. J. Flynn, P. Hung, and K. Rudd. Deep-Submicron Microprocessor Design Issues. *IEEE Micro*, :, July-August 1999.

[4] S. Furber. Asynchronous Logic. In *IberChip*. , February 1996. Sao Paulo, Brazil.

[5] R. Ginosar and R. Kol. Adaptive Synchronization. In *Proceedings of the 1998 International Conference on Computer Design*, 1998.

[6] C. Leiserson and J. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, (1):3–35, 1991.

[7] A. J. Martin. Design of an Asynchronous Microprocessor. Technical Report CS-TR-89-02, Computer Science Department, California Institute of Technology, 1989.

[8] E. J. McCluskey. *Logic Design Principles*. Prentice-Hall, Englewood Cliffs, N. J., 1986.

[9] A. Merchant, B. Melamed, E. Schenfeld, and B. Sengupta. Analysis of a Control Mechanism for a Variable Speed Processor. *IEEE Transactions on Computers*, 45(7):968–976, July 1996.

[10] S. F. Oberman, H. Al-Twaijry, and M. J. Flynn. The SNAP Project: Design of Floating Point Arithmetic Units. In *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*. IEEE, July 1997.

[11] M. Olivieri, A. Trifiletti, and A. De Gloria. A Low-Power Microcontroller with On-Chip Self-Tuning Digital Clock-Generator for Variable-Load Applications. In *Proceedings of the 1999 International Conference on Computer Design*. IEEE, 1999.

[12] A. E. Sjogren and C. J. Myers. Interfacing Synchronous and Asynchronous Modules Wihin a High-Speed Pipeline. In *Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI '97)*, pages 47–61, 1997.

[13] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.