Alternative Timing in Digital Logic

by

George G. Conover

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 8, 2016

Keywords: Asynchronous Circuits, Elastic Circuits, Elastic Clock, GALS

Approved by

Vishwani Agrawal, Chair, James J. Danaher Professor of Electrical and Computer Engineering
Victor Nelson, Professor of Electrical and Computer Engineering
Adit Singh, James B. Davis Professor of Electrical and Computer Engineering

Abstract

For many decades using a system clock has been the go-to method of timing circuits. CPUs in particular have been at least partially defined by the speed of their clock. As technology moves forward, this is proving more and more problematic. At first, clock rates increased as transistor sized reduced. Now, transistor sizes still go down while clock rates remain stable. As a result, the focus has shifted to trying to do more with each cycle. A greater emphasis has been placed on efficiency, because less power draw in each cycle means either less battery drain for mobile devices or more things that can be done within power limitations for circuits with a less transient power supply. To that end, I propose that alternative timing schemes have as yet untapped potential and warrant further industry focus and research. To demonstrate this, various methods of timing are discussed and analyzed, and a demonstration is provided for techniques that have no available statistics.

What follows is an examination of existing and new ideas in circuit timing, with a focus on microprocessors. The first method discussed involves eliminating the clock entirely. The resulting asynchronous circuits are a well studied and discussed idea, which was dismissed previously as being not worth the cost. The progress of processor design in the last few years indicates a renewed study of asynchronous circuits is warranted. The other option explored is when the clock becomes aperiodic. If this elastic clock is one whose width can change from cycle to cycle, instructions with varying worst case timing can control the clock to run a system closer to average case time. This method has not received the same attention as asynchronous circuits, so some new ideas are proposed and demonstrated for generating and utilizing elastic clocks.

Tests were run on a custom CPU design to prove the elastic clock design viable. The single-cycle processor was implemented with 45nm technology, and simulated using NanoSim.

The results show that while the average power increases, the total energy required to execute the test program decreases. The savings are enough to offset the power overhead the new components require. The area overhead is 3% or less; better, if used in more complex designs. Given the complexity of typical pipeline CPUs, the area and power savings of a single-cycle design combined with the throughput improvement shown by the test makes this an interesting alternative for low power applications. Other uses of this technology are discussed and logically analyzed.

Acknowledgments

I would like to thank my major professor, Dr. Vishwani Agrwal for guidance, advice and ideas. I would also like to thank those other professors who lent an ear and gave advice when asked. I would like to thank my family for their support and a providing place to stay while I wrote this. I would like to thank my father in particular for acting as a sounding board for my ideas. Finally, I would like to thank my fellow graduate students for the mutual support and encouragement throughout the last few years.

Table of Contents

Abstract ii
Acknowledgments iv
List of Figures vii
List of Tables viii
1 Introduction 1
1.1 Current Designs 2
1.2 Modification of Synchronous Design 4
2 Asynchronous Circuits 8
2.1 Design Methods 9
2.1.1 Benefits 12
2.1.2 Drawbacks 13
2.2 Asynchronous Microprocessors 14
2.2.1 CalTech 14
2.2.2 ARM 15
2.2.3 Others 16
2.3 Theory vs Reality 16
2.3.1 Power 17
2.3.2 Throughput 18
2.3.3 Others 19
3 Elastic Circuits 21
3.1 GALS 21
3.2 Elastic Circuits 23
3.3 Elastic Clocks 24

3.3.1 Methods of Generation 27
3.3.2 Comparisons 31
4 Test of Elastic Clocks 34
4.1 Method 34
4.2 Results 35
5 Conclusion 43
Bibliography 45
Appendices 49
A VHDL Code Used for Proof of Concept 50

List of Figures

1.1 Intel processor speeds (1993 - 2008) 2

1.2 ARM Opteron pipeline 3

1.3 Glitch Example 6

1.4 Clock Skew 6

2.1 Glitch Free Example 10

3.1 Introduce redundent hardware into a pipeline to increase performance. 22

3.2 2-phase handshake with unsynchronized clocks can take varying ammounts time. 23

3.3 Multiphase Clock 28

3.4 Multiphase Elastic Clock 28

3.5 Multiphase Clock Selector 29

3.6 Multi-Ring Oscillator (results of operation shown in Figure 3.7) 30

3.7 Multi-Ring Oscillator Simulation 31

3.8 Multi-Ring Oscillator Alternative 31

4.1 CPU simulation with 4.6ns clock. Top bus: Data, Bottom bus: Next instruction 36

4.2 CPU Simulation with Elastic Clock via Counter Design 38

4.3 CPU Simulation with Elastic Clock via Multiphase Design 39

4.4 CPU Simulation with Elastic Clock via Stop Clock Design 39

List of Tables

2.1 Truth Table for Figures 1.3 and 2.1 10

2.2 CalTech MiniMIPS Test Results 15

2.3 Lutonium Performance (simulated) 15

2.4 Amulet vs ARM 6/7/8 16

4.1 CPU Instruction Set 37

4.2 CPU execution of Fibonacci Sequence 38

4.3 Power and area additions for each method 40

Chapter 1

Introduction

The clock is a device designed to get everything working together. If the clock edge arrives, all data should be stable, correct, and ready for reading. The clock has been used as a way to gague processor performance, and an easy way to throttle that performance when using automatic voltage scaling. But in the last decade or so, we've seen the problems that can arise with the clock. First, look at Figure 1.1. The figure shows clock speeds of Intel's Pentium lines, as well as their multi-core designs [2]. The trend shown in the 90's shows an exponential growth in clock rate. Then, around 2004, it hit a cap. High performance processors have held at around 4GHz for a while, changing the maximum rate when switching to a new architecture. Rather than just increasing the clock speed, we increase throughput using parallel processing. This can increase performance, but not every program can take advantage of it. Even those processors that can are often not optimized for multi-threading. So the question becomes: is it possible to increase performance on one processor core using different timing schemes?

Before jumping to increase throughput, though, it is necessary to know why we can't just keep increasing clock speed like we have been for the last few decades. The key is power. For one thing, too much power going through too small a line causes problems. Too much power draw also means too much heat generation. Running a modern processor at the maximum speed theoretically possible would destroy the processor. So we lower the voltage, making the switching speed of transistors slower. This makes the critical path through the circuit longer, which necessitates slowing down the clock. Designers can also make a processor have fewer pipeline stages. The reduced number of stages means each stage is longer, again making a slower clock necessary. As transistor sizes shrink, these measures keep the clock

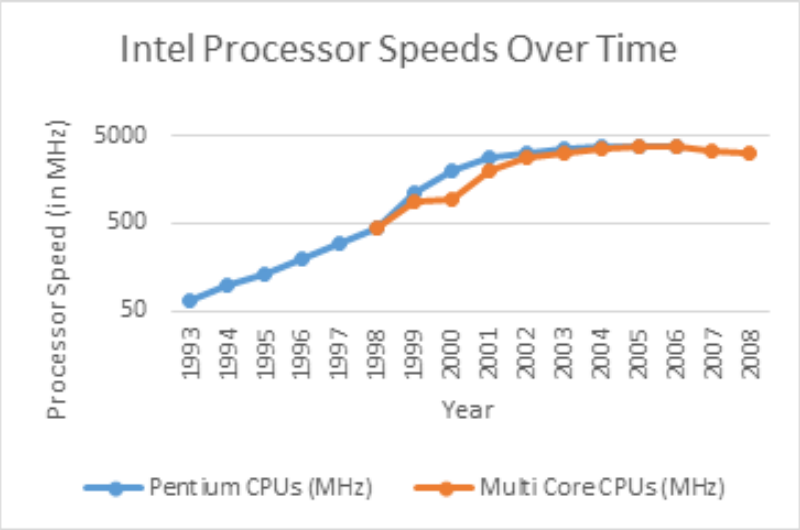


Figure 1.1: Intel processor speeds (1993 - 2008)

speed stable, while reducing the power draw. Typically, these power savings are reinvested in adding more cores in the processor or additional specialized hardware in each core. In addition to increasing single core throughput, then, is there a way to increase efficiency by using alternate timing schemes? That is, is there a way to reduce the amount of power draw per instruction executed?

1.1 Current Designs

The most basic design of a processor is a single-cycle design. The processor pulls an instruction from memory, decodes it, executes the necessary operation with the specified data (either by storing the data in a register or using the data already stored), and decides what instruction will be executed next (the next one in memory, unless jumping or branching through the program). As the name implies, all this can be done in one clock cycle. The problem with this simple of a circuit is that the instructions all take different amounts of time to execute. An unconditional jump only has to set the necessary registers so that the next instruction fetched from memory is correct. Loading a register requires decoding the address of the data needed, then actually pulling the data from memory (on top of fetching the instruction). In a synchronous system, the longest instruction will determine the clock

period. This means that any instruction that takes less than this maximum time (most of them) will just have to sit completed for a long time before the next instruction can start.

This inefficiency leads to the next design approach. A multi-cycle processor is one where the processor is split into multiple stages. Each stage completes part of the instruction. Every instruction will use the first few stages, but the rest are executed or ignored based on the instruction type. The clock can then run as fast as the slowest stage, rather than the slowest instruction. This minimizes down time between instructions, but now it only completes one instruction every few clock cycles.

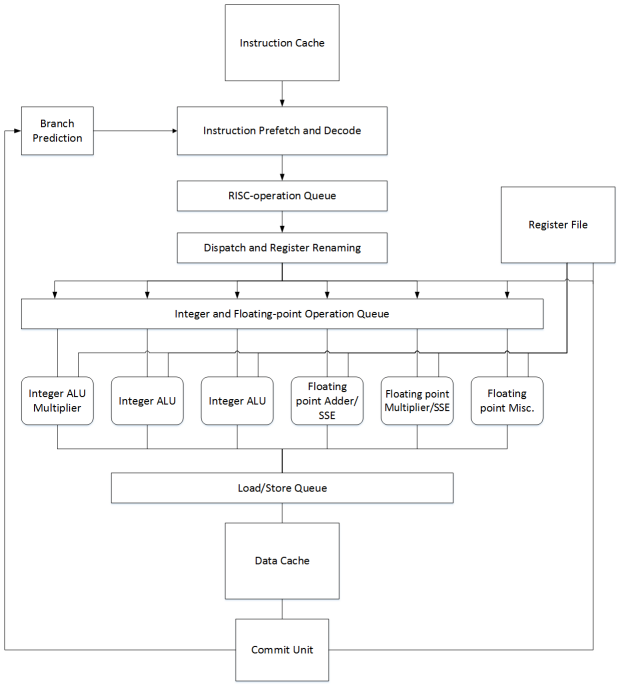


Figure 1.2: ARM Opteron pipeline

This leads to the pipeline. Once an instruction is started, the next one is queued up. If one instruction is started every cycle, a clever design can lead to very close to one instruction completed per clock cycle. Some problems do exist, but this is the way virtually all processors work today. The question for most processors is how many stages to use. Some designs have used more than 30 stages, but the power draw of such systems was a major problem [1]. Most designs today range from 3 or 4 stages in low power designs to around 12 or 13 stages for high throughput [7]. An example of a real world pipelined CPU is shown in Figure 1.2.

1.2 Modification of Synchronous Design

There are three problems that will be addressed by the various timing schemes that will be presented in this thesis. Improving the rate at which operations are executed has been the primary concern of digital design until recently, when mobile devices began to permeate society. With the demand for greater capability in battery powered circuits, both reduced power draw and increased efficiency are necessary. And finally, greater capability has recently been achieved with greater complexity, both with parallel processing and with System on a Chip (SOC) design. These large scale designs present timing challenges, some of which have already been solved with the timing schemes that will be presented later in this paper. There are designs that have been created for traditional synchronous systems to address these issues that will be relevant to the methods presented here, either as similar alternatives or as related ideas implemented in different ways.

To improve on throughput without increasing the number of pipeline stages, it has been suggested that the clock be run faster than the critical path would normally allow. This Better-than-Worst-Case design operates on the assumption that the worst case paths are infrequently used. If this is the case, error detection/correction circuitry can be added to the system that allows the circuit to handle the rare cases where one of the paths longer than the new clock period allows are used. As long as these cases are rare, any time wasted recovering from an error should be made up for by the fact that the clock is running faster. This also can provide some baseline improvement, given that with this method the 10% to 20% overhead usually placed on the clock to ensure correct operation is no longer necessary.

[9] provides a look at the error detection/correction hardware that is needed for Better-than-Worst-Case design, as well as the potential gains such circuits can provide. The initial design method proposed in [12] is called Razor. This used a set of *shadow latches* that get the values going to each register on a slight delay. This means that while the data is stored in the register's flip-flops, the value stored in the latches is essentially on a longer clock cycle. If an error occurred due to a critical path violation, the latch and the flip-flop will

disagree, and the error will be detected. This was initially proposed as a way of controlling automatic voltage scaling, as the voltage and frequency would change when there were a given frequency of errors. As pointed out in [10], these shadow latches are susceptible to the short clock problem (Figure 1.4b), given that there is a deliberate mismatch between the flip-flop's clock and the latch's clock. While this can be solved by putting delay buffers on the short paths, this requires a large power and area overhead. The proposed alternative to the buffers was placing latches at key points to maintain the short path's old value until the shadow latch has a chance to update. They reported an average 15% performance gain and a best case 32% performance gain with an acceptable area overhead.

To improve power consumption, there is a technique called clock gating. The basic idea is: if a memory element is not going to change this cycle, it does not need to be clocked. The simplest way to accomplish this is to add a set of XOR gates whose inputs are the current and next state for the memory elements in question. The XOR gates will be '0' if there is no change. If all the XOR gates in a register are '0', the clock is hidden from the register. Doing this means adding gates between the clock and memory elements, exacerbating any clock skew problems (see below). This is acceptable for low power circuits, given the amount of power it can save. In [11], it was shown this technique saved up to 60% power with about a 12% area overhead.

Another method of addressing reducing power consumption is to reduce glitches. Figure 1.3 shows a glitch, and one possible cause for it. If X is high and Y is low, a transition from high to low on Z should maintain a 1 on the output. Because the inverter takes time to transition, the top AND gate will not receive the updated signal until well after the bottom AND gate, leading to the output shown. Remember that CMOS gates are designed so that they only draw power when switching states. The system can be redesigned to reduce glitches (a delay could be inserted on Z after the tap for Z so that both AND gates see the updated signal at the same time, for example). Doing so prevents unnecessary transitions, thus saving otherwise wasted power.

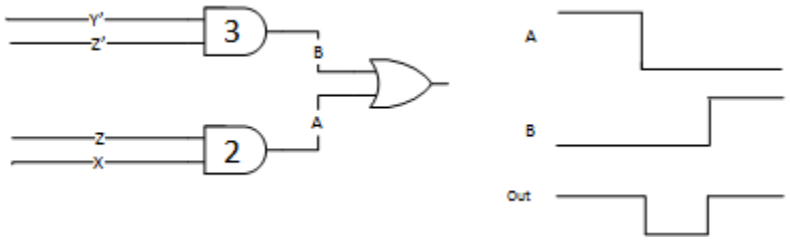
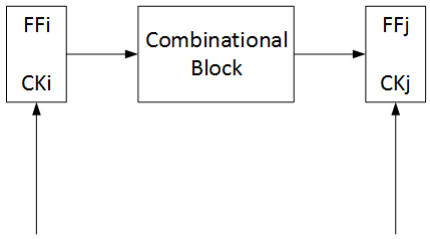
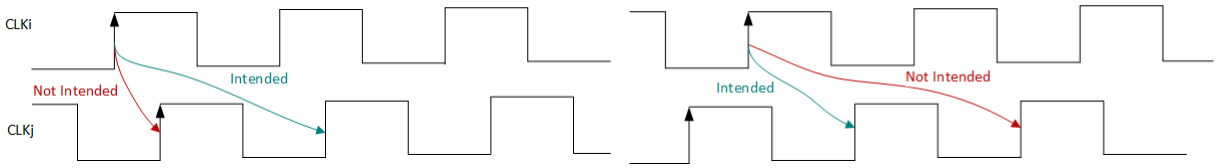


Figure 1.3: Glitch Example

One particular problem in large systems like microprocessors, beyond power or throughput, is the necessity of having the clock reach every recipient in the circuit simultaneously. If the clock arrives at a data recipient shortly after the sender, it can lead to a problem called a double clocking (see Figure 1.4). If the clock arrives at the recipient shortly before the sender, we can get a problem called zero-clocking (the critical path is longer than the time between the sender's rising edge and the intended recipient's rising edge). Either way, the data is captured on the wrong edge and the circuit gives an erroneous result [8]. The problem of the clock arriving at different places at different times is known as clock skew.



(a) Circuit with Skewed Clock



(b) Double Clocking

(c) Zero Clocking

Figure 1.4: Clock Skew

Clock skew is most often addressed by very carefully designing the routing of the clock. A clock is normally distributed through what is called an H-tree [8]. To demonstrate, we'll assume 4 inverters per level. Start with the clock signal, which feeds 4 inverters. Those 4 inverters then each feed 4 more inverters. Those 4 feed 4 more, and so on. As long as the

number of inverters at each level (beyond the one fed by the clock) is equal, the capacitance (and thus delay) should be equal for any inverter at that level. As long as the number of levels between the clock and the flip-flops is equal, the clock should arrive everywhere at the same time. Of course, the distance between the initial clock and the various recipients can cause clock skew, as can process variation. The answer most often used here is simply to put some leeway in the clock, making it longer than it strictly needs to be.

Other techniques are used to address these problems, but these are the ones that will be used, modified, or referenced throughout this discussion.

Chapter 2

Asynchronous Circuits

The design of asynchronous circuits has long been motivated by the potential to eliminate the power draw and speed restrictions of a system clock. The main problems cited for the lack of wide spread adoption of them have been the complexity of such designs and the lack of support in design automation tools. Several asynchronous microprocessors were developed in the late 80's, 90's and early 00's, but none of them reported having achieved all of the advantages promised. They were a test of the potential for the design method at the time, and one which only showed a modest improvement for a high cost. With current design advances providing less and less benefit for their costs, it may be time to look at them again.

There are five main advantages to asynchronous circuits: low power, low electromagnetic interference (EMI), high speed, high tolerance to some types of errors, and modularity. As will be shown, these advantages are not guaranteed in microprocessors. Careful designing is required to get even some of these, and often the design will sacrifice one or more advantages to realize the others. In particular, the processors that will be discussed tended to function with a lower power draw than their synchronous equivalents while at best matching the throughput.

There are three significant drawbacks to asynchronous circuits: they are difficult to design (made worse by a lack of design tools), they are impossible to fully test without additional design for test (DFT) circuitry, and they are more susceptible to some types of errors. The limits of design tools can lead to poor performance, even when the circuit works. The DFT circuitry can cost more than a 50% increase in area to ensure full stuck-at testability [22].

2.1 Design Methods

In creating a circuit without a clock, the ideal design would be one where as long as the circuit starts in a valid state, it cannot reach an invalid state due to timing errors. This is known as a delay insensitive design. The problem is, even in small circuits, such designs are not always possible. As the circuit gets larger, the probability that a fully delay insensitive design exists becomes vanishingly small. As a result, components must be self-timed.

There are several ways to design an asynchronous circuit. Some cases involve designing it around the gate and/or wire delays. Most of the practical designs that will be examined later assume unbounded gate delays and negligible wire delays. These quasi delay insensitive (QDI) circuits allows for more robust designs, and allowing them to tolerate to delays that many designs claim as a primary benefit. It is also possible to take a synchronous design and desynchronize it. This may be accomplished either with delay elements between control circuits or with specialized hardware for completion detection.

If the delays of gates and wires provides the timing that allows for correct operation, it is imperative that there be no cases when a gate changes states before it is supposed to. This can happen when there is a glitch, as described above. As mentioned, glitches can be caused by many things. The example shown was a glitch caused by signals arriving at gates at different times. The solution mentioned was to ensure the signals all arrive at the same time. In traditional asynchronous designs, gates are expanded and made more complex to eliminate glitches.

Take Table 2.1 from [24]. The solution previously proposed was to introduce a delay on Z to ensure simultaneous arrival of the signals. The problem is, trying to exactly gage and manufacture the required delay is difficult at best. Instead, consider the solution in Figure 2.1. When the transition from XY'Z' to XY'Z occurs, XZ goes to 0. In the initial circuit, this causes the output glitch. Here XY' maintains the correct output. As long as only one input signal changes at a time, this is a glitch free implementation. There is a whole set of rules and optimizations based around this kind of design that allows asynchronous circuits to

be created glitch free at the cost of greater complexity. This solution is for Huffman circuits, where gate and wire delays are bounded and only one input is allowed to change at a time. There is a solution for QDI circuits, but it is more complex and will not be demonstrated.

X	Y	Z	Out
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Table 2.1: Truth Table for Figures 1.3 and 2.1

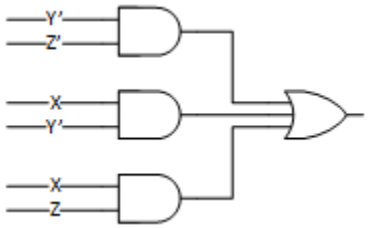


Figure 2.1: Glitch Free Example

Glitches can be ignored in synchronous circuits. If a glitch occurs, just waiting a moment will allow the correct result to be seen. As the clock is based on the worst case path, all gates will have stabilized to their correct value before each clock edge. In asynchronous circuits, however, any value can have an effect on the control signals used to arbitrate between components. An erroneous pulse on a control signal can cause another part of the circuit to start processing bad data. Because of this, asynchronous circuits must be carefully designed to avoid such glitches. This requirement costs asynchronous circuits greatly, as will be seen in a later section.

The microprocessors presented here use a quasi-delay insensitive (QDI) design. The key to the design lies in what are called isochronic forks. That is, it is assumed that any fanout

has negligible delay difference between recipients. In designing QDI circuits, gate delays are estimated and all wire delays are neglected. This forces a tolerance to timing delays, which is what give these asynchronous circuits their robustness in varying environments. The technique also relies on a component often used in asynchronous circuits called a c-element. This gate will hold its current value when the inputs are different, and have a 1 when both inputs are 1 and 0 when both inputs are 0. C-elements are used for asynchronous arbitration between multiple control lines. QDI designs will continue to work under varying voltage, temperature, and other environmental conditions. In nanoscale systems, this becomes a problematic design method, as wire delays become much more influential on circuit timing.

It is also possible to create a system whereby synchronous logic is used in an asynchronous environment. If the method of self-timing allows glitches to occur, optimization can greatly improve throughput. The easiest way of doing this is to insert an artificial worst case path through each component. This means that no matter which path is activated, the new worst case path will take longer, guaranteeing the operation is done. The drawback is that this runs the circuit at the worst case timing for each component, rather than the average case timing. If the new worst case path is just a bunch of inverters, feeding the output back to the input makes it a ring oscillator. This will be the basis for the pausable clock discussed later.

Another interesting possibility is proposed in [15]. They took each component and added some circuitry to monitor the current draw. When the current draw drops below a specific threshold (roughly equal to the static current draw), it means there are no more signal transitions going on in the component. If there are no more transitions, the output is stabilized and the operation is done. While this is a difficult technique to implement in a fabricated circuit, it does allow a speed optimized circuit to run at truly average case time.

In cases where the synchronous circuit is pipelined, the asynchronous circuit can also be pipelined. The technique (called micropipelining) allows each pipeline stage to communicate with the others, computing as necessary and remaining idle when not. Micropipelines often

require more stages than synchronous pipelines to get similar results[13]. Other methods and adaptations of micropipelines have been proposed to increase speed or throughput[39][37].

2.1.1 Benefits

The system clock will often represent up to 40% a system's power draw[21]. The clock signal must reach every flip-flop in every register in the system. As a result, the clock has massive fanout. This can be handled by inserting an H-tree as described earlier. This keeps the speed high, but requires a significant number of additional gates, each of which draws power twice per clock cycle. Each flip-flop the clock reaches also has gate transitions twice per clock cycle. This can be mitigated with a technique called clock gating (also described earlier), where the clock signal is blocked from registers that are not updated. Asynchronous systems have no clock, no clock tree, and no clocked registers. The registers in an asynchronous circuit only update when there is valid data on their input. This is often equated to perfect clock gating, as no register updates unnecessarily. The registers also often use latches rather than flip-flops, potentially saving space and power by using fewer transistors.

The clock also runs on worst case time. That is, the clock is set such that the longest path than can possibly activate has had time to do so, even if that path is not always activated. This can cost performance if many operations do not require as much time to complete, and so must sit idle waiting for the next controlling clock edge. In asynchronous circuits, each operation will signal its completion as it occurs. Some component may need to wait on another to finish, but in general the system will work as fast as operation completion will allow. We therefore say that asynchronous circuits work on average case time.

The quasi-delay insensitive design allows for a high tolerance to timing errors. This includes having no clock skew (given that there is no clock) and mitigating the penalties normally incurred when there are long connections between components. The advantage of delay insensitive designs most often referenced is that any environmental or operational

variances do not impact the correctness of the circuit operation. The circuit timing will naturally adjust itself, both slowing down when necessary and speeding up when possible. Because the throughput is reliant wholly on the switching speed of the transistors, lowering the voltage will cause the circuit to slow down without needing to adjust a clock, leading to naturally occurring automatic voltage scaling. Extreme temperatures also cause the circuit to naturally adjust.

Asynchronous components communicate using a set protocol. The nature of this protocol is such that additional components can be added without needing to adjust the rest of the circuit. No new timing model needs to be developed, as any new worst case path will not have an effect on the existing circuit. This gives asynchronous circuits high modularity. This can be very useful for expanding on proven designs or improving specific components within a system without requiring a complete rework of said system.

2.1.2 Drawbacks

The two biggest problems with asynchronous systems is difficulty in design and difficulty in testing. These issues lead to a lack of good tools, which makes it harder to design competitive asynchronous circuits. A lack of competitive asynchronous designs leads to the design method being ignored. The method is ignored, so no one solves the design and testing problems. It's a vicious cycle.

In designing synchronous circuits, a designer can simply create components that work as independent combinational circuits and connect them with some control logic. The clock ensures all the parts will complete on time. In an asynchronous circuit, each component may have feedback, and must be more carefully designed to generate accurate completion signals and avoid glitches on these signal lines. This costs area and power overhead and makes the design more difficult.

If the design difficulty makes people hesitant to try asynchronous circuits, the problems with testing them make them nearly useless in the real world. Modern testing is primarily

focused on scan testing for stuck-at faults. That is, every register can be pre-loaded with a specific input that will generate a specific output. If at any point a node in the circuit is stuck at 1 or 0 and cannot reach the other state, one of the test vectors scanned in will activate that fault and propagate it to the output. The output will then be incorrect, and this will signal that there is an error in the circuit. This method of testing catches the vast majority of other errors in circuits as well. The problem is, this method is incapable of handling feedback loops. Asynchronous circuits are built on feedback loops. Thus, asynchronous circuits cannot be tested using current test methods without adding a lot of DFT hardware to allow it to emulate synchronous behavior.

2.2 Asynchronous Microprocessors

2.2.1 CalTech

Caltech has designed and tested several asynchronous microprocessors. In [19], statistics are given for simulation and fabricated tests for their MiniMIPS architecture and the results were scaled for comparison to other architectures. The results from Table 2.2 show that the MiniMIPS outperformed all the others in simulation, but the fabricated chip had only 60% of the simulated throughput. The reason given for the difference in simulation and fabricated testing is twofold. For one, the company fabricating the chip did so poorly. More important to this analysis, there was a long interconnect due to an error in routing that cost about 20% of the total performance. Even with the fabricated result, the MiniMIPS show an excellent balance of power (4W) and throughput (180 MIPS) at 3.3V.

After the MiniMIPS, the Caltech team began work on a specifically low power design. This design is called Lutonium. As can be seen from Table 2.3, the simulation shows voltage scaling as it applies to asynchronous circuits. There is no clock that needs to be slowed; execution time slows automatically. This demonstrates the delay insensitive properties discussed above, as well as showing that no additional hardware needs to be added for voltage scaling (now a common practice) to work.

#	Processor	Word	Tech [/um]	Freq [/MHz]	Power per bit	Energy $[/10^{-10}J]$	Et^2 $[/10^{-26}Js^2]$
1	MiniMIPS (sim)	32	0.6	280	0.219	7.8	1.0
2	MiniMIPS (fab)	32	0.6	180	0.125	7	2.1
3	R3000 (CPU)	32	1.2	25			
4	R3000A (CPU)	32	1.0	33			
5	VR3600 (CPU+FPU)	32	0.8	40			
6	R4600	64	0.64	150	0.0719	4.8	2.1
7	21064	64	0.6	200	0.469	23.5	2.1
8	R4400	64	0.6	150	0.234	15.6	7.0
9	SH7708	16/32	0.5	60	0.018	3	8.3
10	P6	32	0.6	150	1.8	120	52

Table 2.2: CalTech MiniMIPS Test Results

V	MIPS	mW	pJ/in	MIPS/W
1.8	200	100	500	1800
1.1	100	20.7	207	4830
0.9	66	9.2	139	7200
0.8	48	4.4	92	10900
0.5	4	0.170	43	23000

Table 2.3: Lutonium Performance (simulated)

2.2.2 ARM

[34] provides a look at the AMULET2e who's statistics provide an interesting picture of asynchronous performance. The table in question is shown in Table 2.4. As shown in Table 2.4, the AMULET 2e had a higher throughput than, and similar power performance to, a low power circuit made in the same technology. At the same time, it had lower throughput than the high performance circuit, but again had very low power. The important statistic to note is power per instruction, in which it outperformed all of its peers. When the microprocessor was made, performance was the primary concern. Now, we judge a processor's effectiveness based on that power per instruction. This means that judging by today's standards, this was the preferable circuit.

In [16], the AMULET 3 is examined. New technologies and design advancements improve the efficiency over that of the AMULET 2e. Reports of the execution of the AMULET 3 state the throughput as equivalent to the ARM9, maintaining the same functionality (the same instruction set, including THUMB mode) while demonstrating an advantage in power consumption. Specific numbers are difficult to find, but the circuit was designed and fabricated for commercial use. This makes it ideal for study as it is a practical design rather than a proof-of-concept.

uP at 5.0V	*Frequency (MHz)*	*MIPS*	*Power (mW)*	*MIPS/mW*
AMULET 1a	-	12	150	0.08
ARM 6	20	18	150	0.12
uP at 3.0V	*Frequency (MHz)*	*MIPS*	*Power (mW)*	*MIPS/mW*
AMULET 2e	-	40	150	0.265
ARM 710	25	23	120	0.190
ARM 710	40	36	500	0.072
ARM 810	72	86 Drystone	500	0.170

Table 2.4: Amulet vs ARM 6/7/8

2.2.3 Others

A group at SUN proposed a counter-flow architecture [20] that would have the instructions and data moving through a pipeline in way that would be inefficient in a synchronous system, but might actually work better than standard methods in an asynchronous one. The Tokyo Institute of Technology designed TITAC, and others have created their own designs. These designs tend to have less published on them, however, making them less useful for this analysis.

2.3 Theory vs Reality

As can be seen by the microprocessors analyzed above, not all of the benefits promised by asynchronous designs can be realized. Some of them can, and some speculation on what

is really going on are presented here. The cause of the performance drop is likely because of optimization, routing, and the effectiveness of synchronous pipelining compared with the difficulties and necessities of asynchronous design.

2.3.1 Power

Power draw in an asynchronous system has been consistently shown to be less than that of a synchronous system. In the case of microprocessors, the power per instruction is often lower than that of other circuits. That said, the power performance has not been as good as is usually promised. The reason the full power savings are not always seen is often because of a large number of small sacrifices made to get a robust asynchronous design working. First is the area overhead for communication. While the clock buffers are gone, each module must have the ability to coordinate with other modules. This overhead is small (often less than 5%), but it is only the beginning. The next part is in designing the self-timing logic. This not only represents additional overhead, but also cannot be as thoroughly optimized as logic that allows glitches. Finally, micropipelines can often require more stages than synchronous pipelines. More stages means more registers. Even with perfect clock gating, more registers means more communications in the system and thus more gate transitions. There may be more causes, but without access to the microprocessors for testing more cannot be said.

As stated, the power required to operate an asynchronous microprocessor is less than that of its synchronous counterpart. The glitch free design eliminates glitch power, which more than pays for the additional gates used to achieve it. The additional registers and communication are offset by the fact that on any given operation, many of them will not see a transition. The perfect clock gating is achieved without additional logic, making it very efficient.

The big savings reported is in idle power[19]. If a system receives no new input, nothing in the circuit changes. In a synchronous system, there would be a constant power draw associated with the clock running while nothing is happening. Even if the additional control

was added to allow the clock to stop when not in use, the startup in synchronous circuits takes time. In contrast, an asynchronous system will simply wait, drawing nothing but leakage power, until new input is ready. Once it is, the circuit can immediately begin the next operation.

2.3.2 Throughput

Most of the microprocessors mentioned in this paper reported equivalent performance (in MIPS) to their contemporaries. This implies they lose the purported throughput advantage asynchronous systems are supposed to have. The cause of the performance drop is likely because of optimization, routing, and the effectiveness of synchronous pipelining

Instructions in a microprocessor have varying lengths. If each instruction is taken from start to finish in one cycle (a single-cycle processor), a lot of time is wasted in synchronous systems. This is because the clock is set by the worst-case path, and any instruction that takes less time (most of them) must spend the rest of the clock cycle idle. An asynchronous system would simply start the next instruction as soon as the current one is complete, eliminating the idle time. This is not how modern processors are designed, however. Pipelining splits the processor into multiple stages with roughly equal critical paths. An instruction that uses the long path in the single-cycle system now uses every stage, while shorter instructions may use fewer stages. Thus, each instruction completes almost as fast as it would in an asynchronous single-cycle implementation, with the added benefit of being able to start the next instruction before the current one is finished. This, combined with a heavy optimization in timing closure (reducing the critical paths until the average case for each stage is close to the worst case) allows for very high performance in synchronous circuits.

Most asynchronous microprocessors also make use of micropipelining. The problem comes when considering optimization. A self-timed circuit must have specific paths that are carefully designed to avoid glitches. This means that these paths cannot be easily optimized, restricting the amount of speedup any given stage might see. Another factor in reducing the

advantage of asynchronous systems is the regularity of instruction execution. Each stage of execution in a pipeline tends to have similar timing from one instruction to the next. This again makes the average case time close to the worst case time, meaning the asynchronous system will see similar timing and throughput as its synchronous counterpart.

When laying out a synchronous circuit, the design tools may be set to focus on maintaining timing, and any poor routing will cause a new critical path, and will likely cause the system to fail. Tracking the cause back to the interconnect is usually straightforward. As was seen in the MiniMIPS from CalTech, a long interconnect may not cause a failure in the system, and will simply appear as reduced performance compared to the simulation. As such, determining that an interconnect is the cause and which one it is can be difficult or even go unnoticed.

2.3.3 Others

One of the benefits that was universally acknowledged was in EMI. A synchronous system draws peak power at or immediately after the clock edge, where every flip-flop in the system transitions at once, and every changed bit causes the gates fed by those flip-flop to begin transitioning. As the cycle goes on, paths settle and transitions become less frequent, reducing power draw. This periodic peak causes the electromagnetic field to interfere with other components as though it was a signal broadcasting at the clock frequency. An asynchronous system has a much more even power draw, reducing the interference. Another benefit is the tolerance to environmental conditions and timing delay errors. This is seen in the MiniMIPS, where an unexpected routing error was not caught by functional tests because the long interconnect did not prevent the processor from functioning as would have been the case in a synchronous circuit. In the raw (unpublished) data report from Caltech, they also gave statistics for the circuit under various voltage and temperature conditions similar to those in Table 2.2. Finally, modularity is one of the benefits that drives the popularity of globally asynchronous, locally synchronous (GALS) designs.

Given that the problem limiting the speed at which microprocessors execute instructions is directly related to power draw, asynchronous microprocessors may finally be able to show an increase in throughput. It was shown in the various examples discussed that the throughput of an asynchronous microprocessor can be made comparable to that of a synchronous design, while using less power at the same supply voltage. If, instead of simply using this power advantage a designer increases supply voltage, the throughput would increase while the circuit draws a comparable amount of power. This may not work as well as one might expect though, as the problem is often in the amount of power used (and heat generated) in specific areas, rather than in the circuit as a whole. If one area of the asynchronous circuit is too active, it can limit the safe operating speed regardless of the overall power draw.

Chapter 3

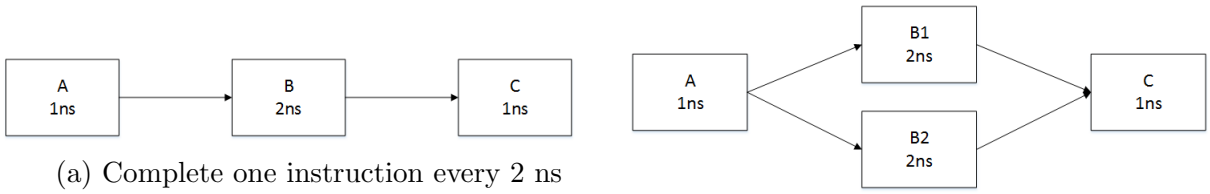
Elastic Circuits

3.1 GALS

Due to the issues discussed in the previous chapter, asynchronous circuits are not used for large systems. One advantage they provide specifically to such systems (the timing-independent communication over long interconnects) can be realized without full asynchronicity. Globally Asynchronous, Locally Synchronous (GALS) systems are those that use asynchronous communication protocols between synchronous components. This means that each component has its own clock, and the clock edges act as the completion signal.

Consider a three stage pipeline (Figure 3.1a). If both 'a' and 'c' have a 1ns critical path, and 'b' has a 2ns critical path, the fully synchronous clock would run at 2ns. The latency (time from the start of an instruction to its completion) would be 6ns. The average throughput would be one instruction every 2ns. If each component is given its own clock, the latency drops to 4ns, but the average throughput remains unchanged. However, if 'b' is duplicated (as shown in Figure3.1b), the throughput improves. If the circuit alternates between using 'b1' and 'b2', an instruction can be completed every nanosecond. The net result in this example is doubling the throughput without doubling the area. This does introduce hazards, but pipelines already use hardware (out-of-order execution components) that can solve this problem.

This is similar to superscalar architecture, where redundant control hardware allows different components to operate in parallel, even when those components are part of the same pipeline. The current AMD architecture uses this in their 6 and 8 core designs[7]. They use 3 or 4 actual cores, each with superscalar designs to effectively double the number of instructions that can be executed simultaneously. The additional hardware overhead and



(a) Complete one instruction every 2 ns

(b) Complete one instruction every 1 ns

Figure 3.1: Introduce redundent hardware into a pipeline to increase performance.

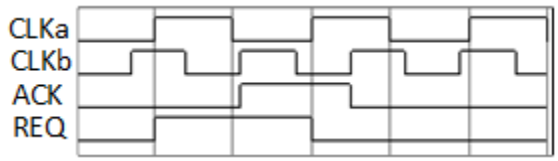
capability is enough to justify calling it a core, though it does not take the same area another full core would cost.

Speaking of which, one of the most obvious uses of GALS is in multicore processing. Giving each core its own clock and coordinating those using asynchronous protocols has a lot of potential advantages. First, GLAS is often used for high level communication over long distances, making it ideal for communication between cores. Second, the ease of communicating between processor cores with different clocks makes it easier to dynamically adjust those clocks. This is similar to what is already done. Currently, CPUs use a slow clock, the frequency of which is multiplied. The multiplication factor can be changed for each core independently for frequency scaling, but they all still run on one clock.

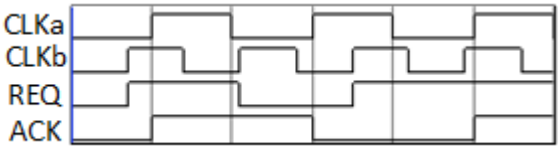
The other place where this is most useful is with a System on a Chip (SoC) design. GALS systems allow each component its own clock, and in the case of a SoC, each component tends to have drastically different timing. As can be seen in [5], this is a common solution.

Using multiple clocks to control the flow of data through a system can lead to some severe performance penalties. Consider the case in Figure 3.2 (an example of Two-phase Bundled Data). Component A is the sender, and component B is the receiver. As seen in the figure, the signals may only appear on or shortly after the controlling clock edge. This can lead to cases where the response time for a component may vary. In a fully asynchronous system, this would not be the case.

The figures provided show the results assuming no hold time on the registers. If the data ready signal arrives before the output of the memory elements has stabilized, it will wait



(a) Component with clock a sends request, component with clock b acknowledges.



(b) Component with clock b sends request, component with clock a acknowledges.

Figure 3.2: 2-phase handshake with unsynchronized clocks can take varying ammounts time.

until the receiver's next clock cycle, making the penalty worse. This performance penalty in GALS is usually negated by the fact that each local clock is faster than a global worst case clock would be. While one component will be operating at the worst case, the total system will be running at much closer to average case. Unlike asynchronous systems, there is no need for glitch reduction. Optimization and timing closure for each component means the average case time will actually be faster than the worst case fully synchronous time.

The purpose of GALS is usually to create easy communication and control between large systems, but it has been applied within a processor as well. [3] describes one such processor, showing up to 20% improvement over synchronous designs. The problem is that providing more than one clock for a processor may not be practical. The increased area and power draw, as well as issues arising from manufacturing variation, may reduce its effectiveness. Potential issues aside, it is a promising way to improve single core performance.

3.2 Elastic Circuits

There are other ways of using creating a globally asynchronous system. The general term for those systems that have some allowable slack in timing (thus allowing resilience to timing variation) is elastic circuits [25]. Elastic circuits can range from fully synchronous circuits whose long interconnects are allowed to hold information for longer than one clock cycle, to fully asynchronous circuits.

Synchronous elastic circuits are those in which the global clock is maintained, and the long interconnects can stall the system to allow multiple clock cycles for communication.

The overhead for this is very low, but the efficiency is also low. The only way this could be useful is if the long interconnects requiring the elasticity are infrequently used. These paths would otherwise be the critical paths, thus requiring a slower clock to allow for correct communication. As long as the long interconnects are infrequently used, the faster clock will make up for the occasional stalls. This is similar to the Better Than Worst Case design.

The next level of elasticity is GALS. The overhead for GALS is greater than a synchronous circuit, as there are multiple clock domains. Each clock domain not only requires its own clock, but also the communication circuitry. The benefit of this method is not only the much more rapid response to communication, but also the fact that each clock working at its own speed means the system runs much closer to average case time. As more clock domains are added, the circuit gets closer to this average case timing and becomes more and more resilient to timing variation, but at the cost of more overhead. The exact point at which the cost outweighs the benefit is up to the designer to decide.

The next step is a quasi-delay insensitive design, which is fully asynchronous. The pros and cons were discussed before, including the necessary overhead and the additional benefits beyond simply timing resilience. The purest form of an elastic circuit is then a fully delay insensitive design. As stated before, delay insensitive designs range from impractical to impossible.

3.3 Elastic Clocks

While elastic systems are already being used (primarily with GALS design), there is another way to change how a circuit is timed. Rather than using multiple clocks (or no clock at all), a design can simply vary the clock cycle. This is known as an elastic clock (or aperiodic clock). This seems counter-intuitive at first, as the point of a clock is to ensure regularity in operation. Asynchronous circuits use a completely different method of timing, and GALS still uses traditional clocks. So what can be gained from using an elastic clock?

Why hasn't it already been done (or at least, done more often)? This was the idea that started this whole timing scheme analysis, so there must be something to it, right?

The idea behind an elastic clock is to create a clock signal whose period changes from cycle to cycle. As described above, this can be used to create a situation where communication time between components is minimized while maintaining the clock for completion timing. Discussed here are other potential uses for this technology, many of which have not been previously discussed or tested.

In a processor, each type of instruction flows through the circuit differently. Knowing the layout of the processor, the designer can predict roughly how long a given instruction will take to go through the circuit. This knowledge is usually used to design a pipelined system, allowing the system to not only run each instruction closer to its minimum time, but to run several instructions at once. However, the more stages a pipelined system has, the more power is drawn due to the clock. There is a balance that must be struck. Some modern low power ARM microprocessors run on a three stage pipeline, while high performance ones run on a thirteen stage pipeline [7]. This being the case, a single cycle system running on a variable clock should be considered.

Using the variable clock, the length of each cycle can be tuned based on the instruction being executed. This is most useful in a single-cycle design, where each instruction takes a radically different path through the circuit. Some components may not even be used in a given cycle. While this can significantly reduce the effectiveness of superscalar designs, this can still be used in parallel processing by treating each core as an asynchronous processor and using GALS protocols for cohesion.

An elastic clock can also benefit better-than-worst-case designs. As it stands, the current designs will often relax the clock with a clock divider. This is a very simple way to do it, though it can require substantial unnecessary time for the slower execution. If the error rate is one instruction in one thousand, doubling the execution time is acceptable. That said,

the simpler elastic clock generator methods may prove viable to get that last little bit of efficiency.

Another point worth considering is what happens when the system must be idle. If there is a cache miss, for example, the system must wait for the cache to update before continuing. If the speed of the main memory fetch is known, the width of the idle cycles can be tuned like any other instruction to minimize response time and number of clock cycles. If there is another component the CPU is waiting on with unknown timing, the system can check as rapidly as the input clock will allow, minimizing response time at the cost of power. If the system is simply waiting for a new input to begin working again, it can make the idle clock cycles as long as the elasticizing circuit will allow, minimizing the number of cycles (and thus power) from the clock during these idle periods.

This idea may even find a place in pipelined systems to improve throughput. If a low number of stages are used, each stage can report its required timing for the instruction being executed. The overall worst case between the sections is then used as the cycle length. This could allow for some improvement in throughput for this type of system without requiring the designer to coordinate multiple clocks. If multiple clock were used, it could allow a GALS system to get even closer to the average case timing asynchronous circuits strive to provide.

So far, the focus has been on microprocessors, as they can derive benefit from elastic clocks with the worst case timing provided by each instruction. There are other ways the timing can be known. In testing, not only is the timing of each test pattern knowable, the cosumed power can also be measured and recorded. With the timing information, elastic clocks can be applied to the test clock (though not the scan clock). With the power data, the voltage applied to the circuit can be altered for each test. The point of this is that there is a maximum amount of power allowable for testing the circuit, and most tests do not draw that much. Adjusting the voltage means each test can be run at maximum power, translating to running at minimum time. The elastic clock is what enables that minimum

time execution, though it can also be done asynchronously if the necessary area overhead is acceptable. [31][32][33].

3.3.1 Methods of Generation

One conceptually simple way to generate an elastic clock is to use a counter. Having the counter always counting using a fast input clock, an input number can be provided. The circuit would then compare the input to the current count. On a match, the output clock signal from the counter is toggled and the count is reset. This allows for a fine control over the new clock period, and allows for the fast clock to be supplied from an external source. This also means that one fast clock can be supplied to multiple components, each of which has its own counter. The input to the control can be based on the circuits inputs (such as the opcode for a microprocessor), and the exact timing for each operation can be stored (allowing for modification based on operating conditions). The duty period is also maintained at 50%, which can be advantageous when there is significant fanout from the clock. The problem with this method is that the supplied clock needs to be very fast for fine resolution in the output period. There is also the potential problem of the delay between the output clock edge and the new timing input. If the circuit passes the point where it should toggle the output without the new timing input correctly showing up, it could cause problems. There is also the fact that this requires some potentially significant area overhead.

Another way of creating an elastic clock is to use a multi-phase clock generator. As seen in Figure 3.3, a multi-phase clock generator provides n signals, each at CLK/n speed. One use of this that has been explored is in parallelization (see Figure 3.2). Each phase signal is sent to a parallel component. The instruction issuing and committing hardware is run at the base clock speed. Each component is then run on a different phase signal. The lower clock speed allows the voltages to be significantly lower, so that even with the extra hardware the overall power draw is reduced.

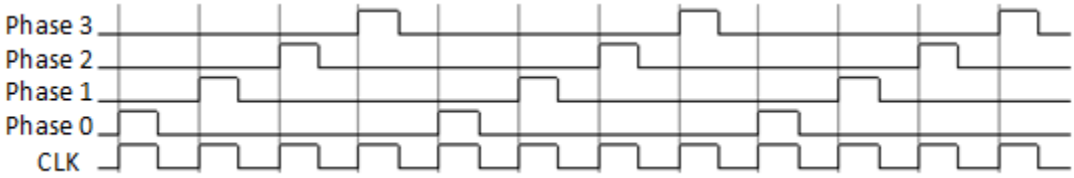


Figure 3.3: Multiphase Clock

The multi-phase design can also be used to create an elastic clock, as seen in Figure 3.4. The circuit (Figure 3.5) takes the desired cycle length as control signals. These signals are combined with the current selected phase, and used to select the next phase. The selected phase is then used as the current clock signal. The end result is the output clock cycle width changing as different phases are selected.

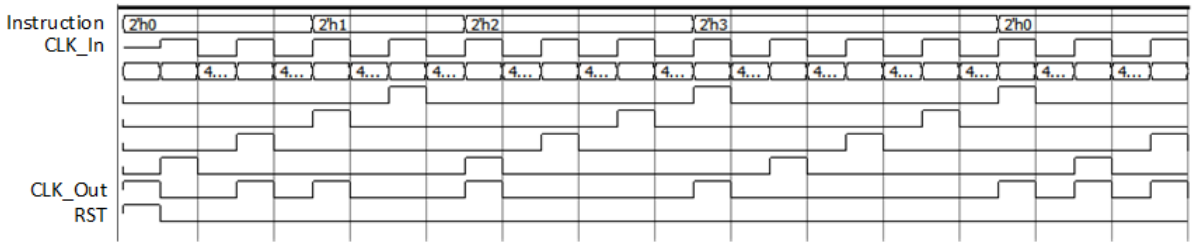


Figure 3.4: Multiphase Elastic Clock

To clarify, suppose the current clock output is seeing Phase 0. The inputs to the clock multiplexer are "00". If the next clock cycle is supposed to be one base clock cycle long, the select lines will be 00. The first flip-flop will see its own Q, inverting the bit. The second flip-flop will see Q1 XOR Q2. The new flip-flop values become "01", selecting Phase 1 as the next clock. Note that the flip-flops are falling edge triggered, so that the update will occur after the current phase has held a stable high value. If the next clock is supposed to be four base clock cycles long, the inputs will be "11". The flip-flops that select the phase signal to be used (currently holding "01") will update to "01". That is, there will be no change. This system has a few drawbacks, including area overhead and speed limitations. This will be explained later.

The Advanced Configuration and Power Interface (ACPI) specification[44] uses a clock throttle circuit that can be modified into another method with many of the advantages

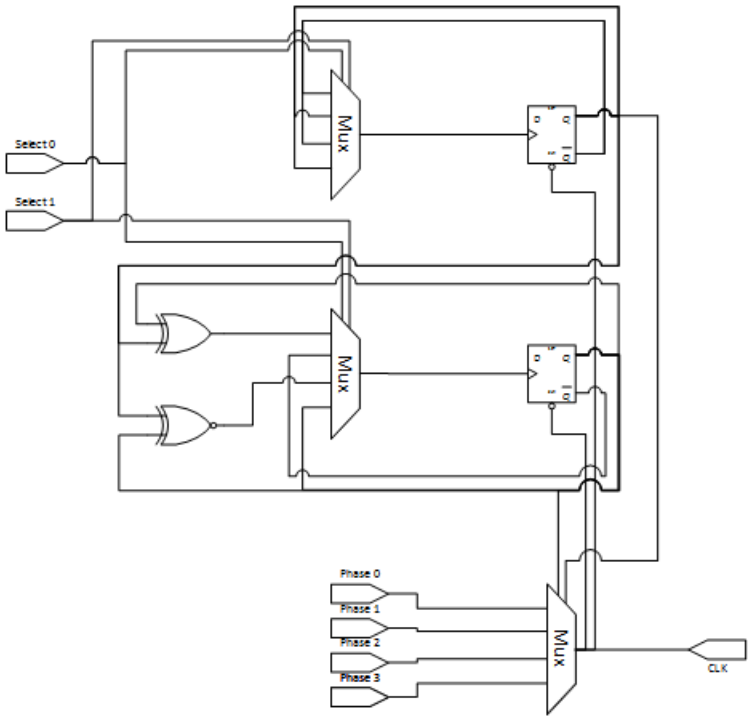


Figure 3.5: Multiphase Clock Selector

of both previously mentioned methods. This would inserts a "stop clock" signal for some number of input clock cycles to prevent the clock from running while the signal is on. The original design has the clock running for x/n cycles and off for (n-x)/n cycles. The new method simply allows the clock through once every x cycles. This makes the output look like the multiphase design, while keeping the simplicity of the counter.

Simpler than modifying a clock input signal is to have the clock generator capable of varying. GALS systems usually rely on each component or group of components having a locally generated clock, so using a variable clock generator in GALS will require minimal redesign and overhead compared to designs which rely on modifying outside clocks. Two methods of using this idea will be presented here. Both methods rely on modifying a ring oscillator.

The first method is the stoppable, or pauseable clock[24][28]. If one of the inverters is replaced with a NAND gate (as seen in Figure 3.6), then the oscillator can be disabled by an external source. If the line from the ring that feeds into the NAND gate is tapped, it can act

as a worst-case completion signal for the component. If every component in a GALS system uses this method, these completion signals (along with the communication protocol) can be used control each clock. The problem of variable response time in GALS is eliminated with this variation, as each clock will wait until communication is complete until starting the next clock cycle.

This concept can be thought of as an asynchronous system where each component is self-timed not by logic comparison and glitch-free pathing, but by adding a worst case path through the circuit. This loses the low-power advantage, as well as some of the fine grained resistance to timing variation that makes asynchronous circuits attractive. At the same time designing each part becomes much easier, and as stated in the section on asynchronous systems, optimization can provide even better performance for this setup than asynchronous systems can usually accomplish.

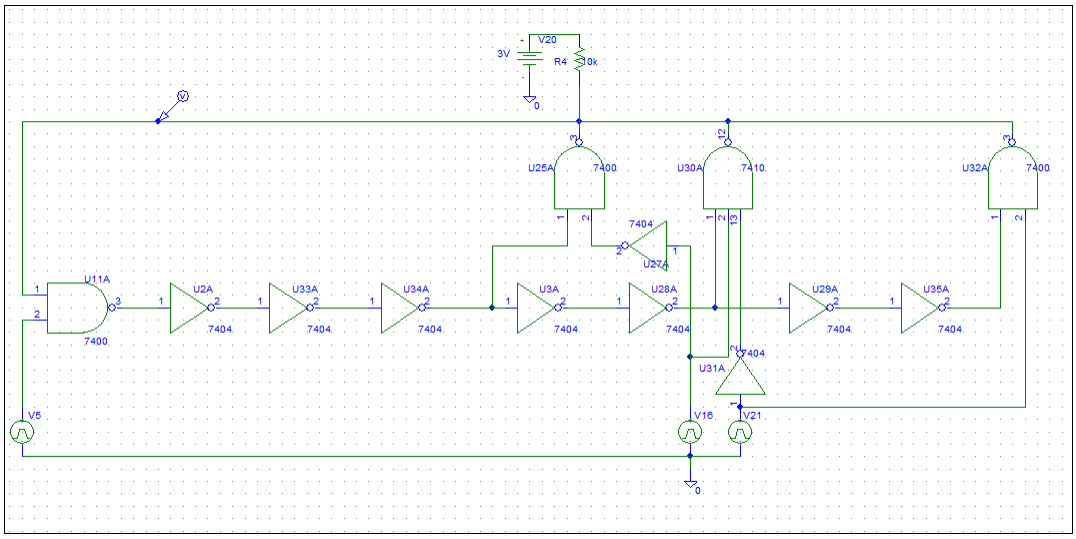


Figure 3.6: Multi-Ring Oscillator (results of operation shown in Figure 3.7)

An alternative to the pauseable clock is the multi-ring oscillator. This approach does not have the advantages in GALS systems that a pauseable clock provides, but can be used like the other designs in this section. Figure 3.6 shows one such clock, with the simulated results in Figure 3.7. Note that the method shown here is actually a bad design, as the common line has a direct connection from the power supply from the unused paths (holding

a logic high while not used) to ground on the active path on a logic low. The whole point of CMOS is to avoid this situation. An alternative (Figure 3.8) was explored where each line was opened while not in use by way of a pass transistor, but the realities of transistor physics made designing such a circuit difficult. No successful simulations were obtained using that method.

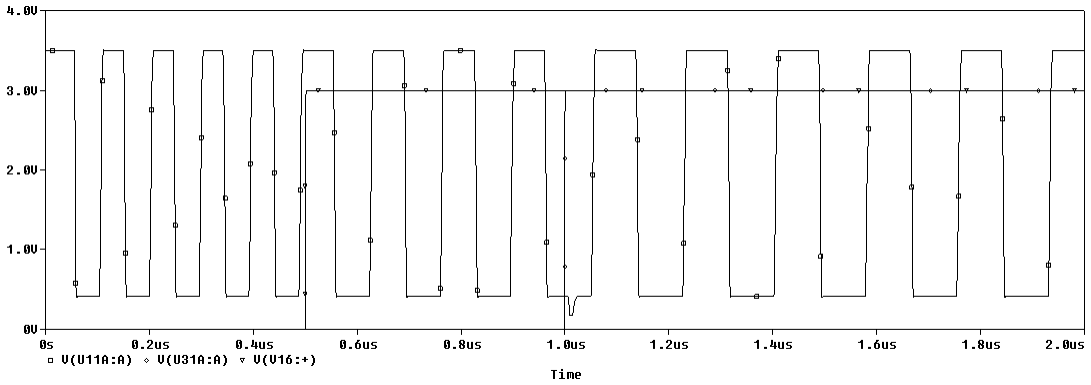


Figure 3.7: Multi-Ring Oscillator Simulation

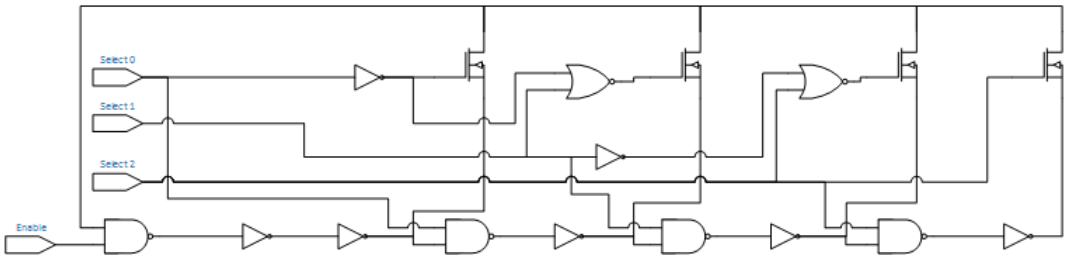


Figure 3.8: Multi-Ring Oscillator Alternative

3.3.2 Comparisons

When using an elastic clock, it is important to know the context in which it will be used. The area, power, and performance requirements, as well as the bigger system in which it will be used, can all have an impact on the choice of method of elastic clock generation.

When high throughput is the focus of the design, a pausable clock will likely work best. It provides an improvement to the GALS design with minimal additions. The improvement allows the system to run closer to average case time, which is one of the key benefits touted

by GALS. It also does not require designing for a response time between the start of an instruction and the next instruction being known. The other methods can potentially provide better execution time per component, but they still have the communication mismatch that this method was designed to solve. This communication mismatch can be mitigated in the counter, multiphase and stop clock methods by having the operating clock mirror the fast input clock and run at the slower specified timing when the operation starts. The pausable clock also costs the least in area overhead, assuming local clocks were in use from the beginning.

When GALS design is used, the system could be designed without local clock generation. If having multiple clocks proves problematic, the multiphase clock generator/selector and stop clock circuitry can be used to enable GALS design. This is particularly useful when the design requirement includes allowing an external clock, as often happens in microcontrollers. It also means that automatic voltage scaling needs only to slow down the one clock, rather than worrying about each one individually. The methods based on modifying ring oscillators obviously cannot offer the same benefit. The area cost is an interesting tradeoff between multiphase and counter design, though. With the multiphase design, one unit creates the multiple phases and each component selects the phases independently. This makes the transistor count lower than it would be if a counter was put on each component, but it means that every phase line must go to every component.

If the requirement is lower power, the counter, stop clock and multi-ring oscillator provide good efficiency gains for low overhead. As long as the clock is locally generated by a ring oscillator already, the multi-ring design provides a variable clock at the lowest area overhead possible. If the clock is generated through another method or provided from outside, the counter and stop clock are very simple and effective methods, as will be demonstrated later. The multiphase design can also work, though there is a tradeoff to consider. The multiphase design requires more logic gates (though admittedly not many more) for one output clock, but can send the phase signals to multiple locations, reducing the total number

of gates when compared to the other methods. The stop clock provides the same output with fewer gates for one clock. It is also more scalable, in that the design can allow for more time intervals selectable without drastically increasing the area. The counter provides an adjustable duty cycle, which may prove useful and can provide more control over exact cycle width. The problem is the requisite speed of the input clock. The multiphase and stop clock designs can provide similar fine grained control with half the input clock speed. The counter is still viable with the high speed clock generation methods available today, but the power cost for each type is still a concern.

The advantage of the counter, the stop clock and the multiphase design is that they can be reprogrammed. Each instruction can be tested and its timing adjusted to match the results. Because of this, the whole system would not need to be slowed down if only one instruction type fails at its specified speed. This can also prove beneficial to designers, as they can change the clock period for each instruction as they modify the design. Without it, they would either have to redesign the clock each time they change something, particularly when adjusting an existing established design.

Chapter 4

Test of Elastic Clocks

4.1 Method

To prove the concept viable, a single-cycle reduced instruction set processor was designed and simulated both with and without an elastic clock. Because this was done as a proof of concept rather than an attempt to analyze the potential of this idea, the CPU was designed to be as simple as possible. It was capable of basic logic and arithmetic, some program flow control, and memory access. Both the instruction and data memories were 64 byte asynchronous read/synchronous write blocks of registers rather than true cache memory. The clock provided to the memory forced the input signals to specific values, ignoring the capacitance involved with sending one signal to 16,000 flip-flops. The rest of the circuit used an h-tree, inserted by hand, to reduce the capacitance on the clock line and make the circuit fast enough for testing.

The test conducted calculated the Fibonacci sequence over a number of iterations. Average power was calculated during the execution without the programming and setup time considered, and the average power will be discussed in the next section. The test output value was first obtained by running the circuit for the correct number of iteration with a very slow clock. The clock speed was increased until the circuit failed, thus giving the approximate maximum clock speed. Each instruction was then independently tested for worst case operation, and the results are shown below.

The elastic clock methods used for testing this circuit required a very fast input clock. As a result, the CPU needed to know what the next instruction would be as soon as possible. Normally, the CPU would hold the address of the next instruction to be executed, pulling that instruction from a synchronous read memory on the clock edge. This means the delay

between the clock edge and the instruction arriving at the CPU may be substantial. Indeed, attempting to fetch each instruction from memory on the clock edge prevented the elastic clock circuits from running at acceptable speeds. What was done instead relied on the fact that any memory for this kind of system would have to be asynchronous. The clock edges would be arriving at seemingly random times, so the whole system would be treated as a GALS system. The result of that, combined with the single cycle nature of the CPU, means that the instruction can be pulled from memory before the clock edge. Doing it this way allows the instruction to be pulled concurrently with the execution of another instruction, giving a better worst case execution time. The only exception is the branch instruction, which must wait until which instruction will be executed next is decided, then pull it from memory. Using this configuration, the CPU will know the next timing requirement as soon as this data makes it through the register on the clock edge and out to the clock component. This allowed components with very fast input clocks to operate correctly.

4.2 Results

The CPU used for the tests was described above, and its instruction set is shown in Table 4.1. This was implemented using 45nm technology, where the CPU used low power gates and the clock modification circuit used high performance gates. Note that the memory model used does not reflect reality. Rather than an actual cache model, the memory used was simply a large bank of flip-flops that were forced to clock at a specific rate. It was still useful, given that the timing for memory operations was at least different from the other instructions, and a range of instruction execution times was the point. That said, the memory was not included in the power analysis. It is also important to note that because of the simplicity of the design, what were called branch instructions in the code and the algorithm were instead conditional skip instructions. If the program is supposed to branch, it will go to the next instruction, which must be an unconditional jump. If it is not supposed to branch, it will skip the jump.

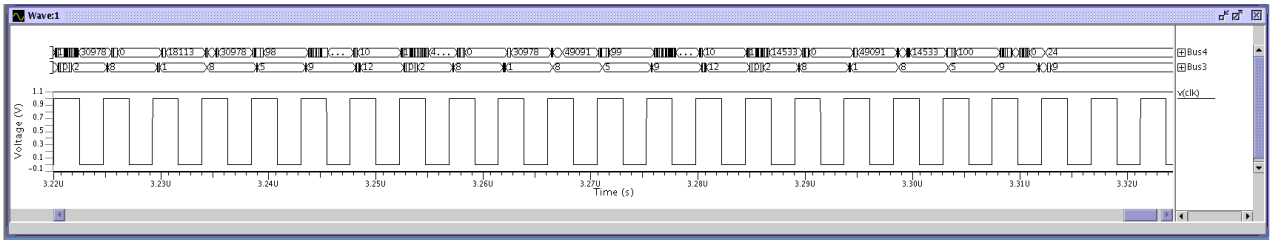


Figure 4.1: CPU simulation with 4.6ns clock. Top bus: Data, Bottom bus: Next instruction

Two methods of calculating the Fibonacci sequence were used (shown in Table 4.2). The counter, stop clock, and multiphase methods were used. The tests were conducted with a 1V power supply, and all the timing was estimated using a program that attempted to execute worst case scenarios for each instruction. These numbers were used to determine the speed of the input clock, such that the output closely matched the required times. The CPU clock operated at 0.65ns intervals, requiring a 0.325ns input clock for the counter. This was chosen because each instruction's worst case timing was below and very close to a multiple of 0.65 ns. While the clock could have been run slightly faster, the multiples of the faster clock that would be used for each instruction did not match as closely to the critical paths. For example, if one instruction required 1.2 ns to complete, the 0.65 ns clock would stretch to 1.3 ns, while the 0.5 ns clock would have to be stretched to 1.5 ns.

The circuits worked as expected (See Figures 4.2, 4.3 and 4.4). The initial test, done at a steady 5ns clock, completed with plenty of room and gave a baseline for the correct result. Reducing the clock to 4.6ns reduced the execution time to 3.30us. When the elastic clocks were implemented, the total time for the program was 2.187us (including setup). The execution time was the same with all clock types, given that they all operated on 0.65ns increments.

The multiphase design took a bit more work than the others to get running correctly. It required only half the clock speed of the counter, but had a problem that prevented it from working in the initial tests. It turned out that the synthesized circuit had a glitch that could have interfered with normal operations. This glitch was tied to how the reset line could select the clock output through complex gates, and changing the gates to a larger and-or

Instruction	Assembly	Format	Worst Case Timing (ns)
Load Word	LW $destination, [$address]offset	0001 DDDD AAAA OOOO	1.7
Store Word	SW $source, [$address]offset	0010 SSSS AAAA OOOO	1.7
Load Immediate	LI $destination, immediate	0011 DDDD IIII IIII	0.75
Add Immediate	Addi $destination, $source1, immediate	1000 DDDD AAAA IIII	3.8
Add	Add $destination, $source1, $source2	1100 DDDD AAAA BBBB	3.8
Subtract	Sub $destination, $source1, $source2	1101 DDDD AAAA BBBB	2.5
And	And $destination, $source1, $source2	1110 DDDD AAAA BBBB	
Or	OR $destination, $source1, $source2	1111 DDDD AAAA BBBB	
Branch if Equal	Beq $source1, $source2	0101 XXXX AAAA BBBB	4.6
Branch if Greater	Bgt $source1, $source2	0110 XXXX AAAA BBBB	4.6
Jump	Jmp address	1001 IIII IIII IIII	1.2
Jump Return	Jr	1010 XXXX XXXX XXXX	1.2

Table 4.1: CPU Instruction Set

configuration fixed the problem. That done, other issues were brought into focus. The circuit did not work with the 0.65ns input clock because of how the clock was selected. The clock that will be seen on the output is selected at the falling edge of the current output. If the circuit has not updated the instruction lines before that falling edge, the next clock selected will be incorrect. This had also been a concern with the counter method, and both were solved with the same solution. Both clock generators were simulated with high performance transistor models, where the CPU was simulated with low power transistor models.

The stop clock design was changed to generate a more stable output. Instead of passing through the clock, which would be susceptible to the same glitch the multiphase design saw,

Test 1	Comments	Test 2	Comments
LI $1, 0	Starting first value is 0	LI $1, 0	
LI $2, 1	Starting second value is 1	LI $2, 1	
LI $4, 64	64h is the number of iterations	LI $4, 64	
LI $5, 0	Initialize the counter	LI $5, 0	
Add $3, $1, $2	Add the two current values	Add $3, $1, $2	
SW $2, [$0]	Store the result	Addi $2, $3, 0	Given the problems
Addi $1, $2, 0	Shift the value from $2 to $1	Addi $1, $2, 0	with memory, test 2
LW $2, [$0]	Store the result in $2	Addi $5, $5, 1	runs without it
Addi $5, $5, 1	Incriment the counter	Beq $4, $5	
Beq $4, $5	If counter = 64h, skip the jump	J inst5	
J inst5	Jump back to the add until counter = 64h	J inst11	
J inst12	Jump to current instruction, halting the program		

Table 4.2: CPU execution of Fibonacci Sequence

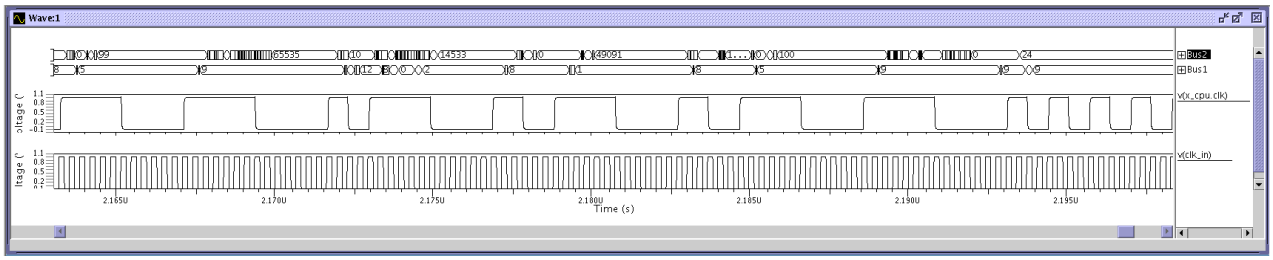


Figure 4.2: CPU Simulation with Elastic Clock via Counter Design

the output was forced high when the count was '0' and forced low when the count was anything else. This means the output is high for one full input cycle, rather than matching the clock on that cycle. The output is more stable, but the minimum output clock cycle is two input cycles. Because of that minimum, output cannot match the input, but in this implementation there was no instruction that required that.

Using high performance transistors did impact the power, but the total overhead remained low. Table 4.3 shows the overhead, both in power and area. The multiphase design is larger than the others because it includes both the multiphase clock generator and the clock selector. That said, its power draw is less than the counter method because it uses

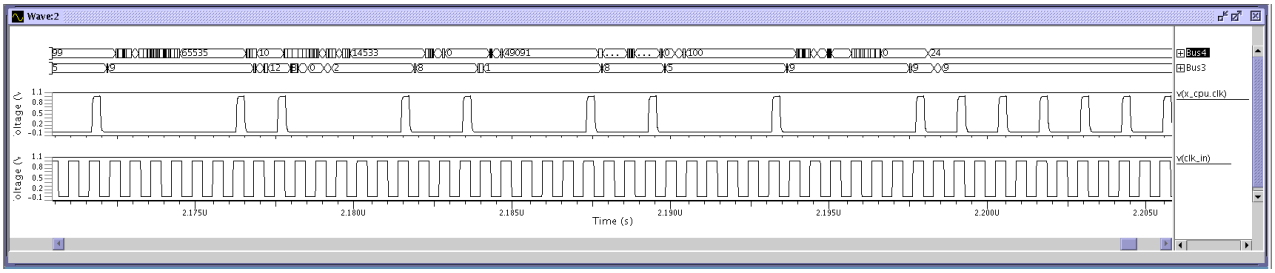


Figure 4.3: CPU Simulation with Elastic Clock via Multiphase Design

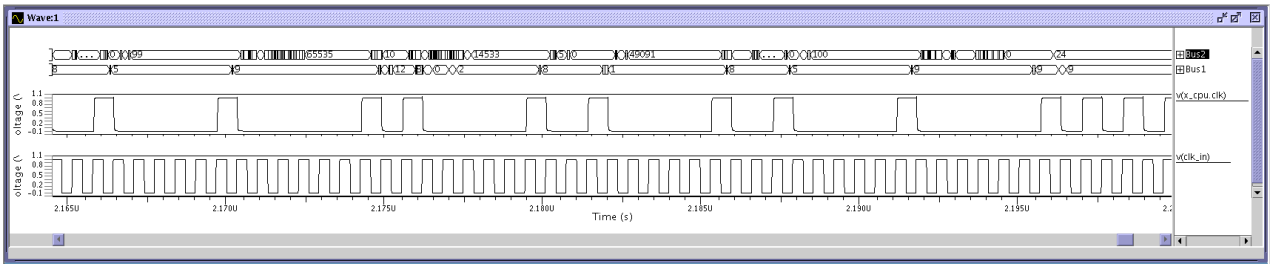


Figure 4.4: CPU Simulation with Elastic Clock via Stop Clock Design

half the input clock period of the other without having twice the activity per output clock cycle. The stop clock design had the best of both. In all cases, the average CPU power was increased because there was less idle time. If multiplied over the total time, the CPU using the modification requires less energy to execute the program. This is actually enough to offset the energy used by the new circuits.

Also of note is that the area overhead is about 3% for the multiphase design and less than 2% for the others. This includes the circuitry that determines the length of each instruction, but as a set combinational logic circuit rather than a programmable memory. The memory would likely be more useful, but take more area. This additional area would then be offset by the fact that any design that uses it would be more complex to begin with, meaning even a larger circuit would still be a very small overhead.

The issue of the input clock speed shows a difference between the methods demonstrated. On the one hand, the multiphase and stop clock designs uses the input clock speed to achieve nearly the same fidelity as the counter. On the other hand, as long as the slowest allowable output clock in the counter is longer that the requisite minimum for correct operation the output can have any graduation the input clock can allow. If the any of the designs are used

	Area (# gates)	Power (avg, mW)	Power (RMS)	Execution Time (us)
Base CPU	2709	0.58885	0.85320	3.1648
CPU + Elastic Clock	X	0.79538	0.79745	X
Comparator	51	0.16337	0.29986	2.0608
Multiphase	82	0.1290	0.26299	2.0608
Stop Clock	49	0.10033	0.22718	2.0608

Table 4.3: Power and area additions for each method

in a globally asynchronous system, it can be set so that it will always use the same period. This will bypass the minimum input requirement and allow for any level of fidelity necessary.

Comparing the results from the elastic clock and the inelastic clock is not necessarily indicative of the results that would be seen in a real system. For one thing, the memory model used was not realistic. For another, there would be many more instructions. Each additional instruction increases the amount of memory required to store timing information. This additional memory could have an adverse effect on the maximum speed at which each method could work. That said, a real system would likely have a much longer critical path relative to the rest of the instructions, making the potential time savings that much better. What's more, the percentage overhead the elastic clock generation represents would be greatly reduced as circuit complexity increased.

The real comparison that would need to be made is not against the single cycle system used, but a pipeline system that would arise from the same design style. The purpose of simulating the circuit presented here was to prove the idea could work, so no such comparison was made. Of course, even if this had been done, the results would still be a poor indicator of the performance gains from elastic clocks. The circuit would still be too simple to give an accurate demonstration of its potential.

To speculate at what the results of more comprehensive tests would show, it is necessary to understand what was used here and what was missing from the design. The processor described here is a 16 bit architecture and uses only 12 instructions. MIPS, a reduced instruction set architecture commonly used for such comparisons, is a 32 bit architecture

and uses 31 instructions in its core instruction set[1]. The difference between the design used here and a commercial design means there could be a wider disparity between the fastest and slowest instructions. It also means a much larger CPU and longer critical paths for each instruction, which means more time between instructions for the counter or multiphase design to update. There is also the possibility that the increased complexity of the clock layout and drive strength requirements reduces the effectiveness of such a design.

When comparing to a pipeline system, there are several factors that will favor the new single variable cycle CPU. First, area overhead of the counter or multiphase module is minuscule next to all the components that are required for a pipeline. A pipeline needs registers to hold data between stages. A pipeline needs to have error detection/correction hardware. Most modern pipelines use branch prediction and code reordering to mitigate the problems inherent in the design. The code reordering and commit units are often their own stage. All of this is unnecessary when using a single cycle processor, which means much less area required and much less power consumed. Remember, single cycle processors are not used because their throughput is so poor. If a simple 4 stage pipeline is used, it may take at best 1/4 the single cycle design's worst case time to finish. Less, in fact, because of the difficulty in matching the length of pipeline stages and the various points of overhead. A single cycle system running on average case time will be dependent on the program being run, but can potentially run at 1/2 worst case time or less. What's more, the idle power draw of a variable cycle system will be drastically lower than a pipeline system. The longest clock period in this system will be longer than even the worst case cycle time of the basic single cycle system, and will be several times slower than a pipeline's clock. This means fewer clock transitions during idle time, reducing power draw. All of this is good enough that the power and area savings make variable clocks viable for low power designs.

If implemented for throughput, a pipeline system may be able to operate with a variable clock. Each stage will know what is required of it, and can have the time that stage will require for that instruction stored. Each stage then reports its required time and the worst

case of those reports is used for the system clock. The problem likely to arise from this is one stage having the worst case time every time (if pulling the next instruction from memory is the worst case time, for example). This does present some interesting potential for optimization, however. If parts of the system are optimized for area, the speed penalty for doing so could be reduced by this method. Say one part is rather large, and would only exceed the worst case time occasionally if optimized for area. It could be monitoring the instructions for the occasional one that could activate the worst case path. It would then slow the system for one cycle to allow for correct operation. In fact, this could be considered predictive better-than-worst-case design. The one problem here is that a very fast clock cause issues depending on how quickly the required clock width can be discovered.

Chapter 5

Conclusion

A clock is not the only way to coordinate operations in a circuit. Asynchronous circuits are a proven technology that can reduce power consumption without reducing throughput. The drawbacks of such circuits were considered too severe in the past, but between advances in the technology and a shift in focus in circuit requirements, they are worth considering again. If fully asynchronous circuits are not viable, other timing methods can address the same problems while being easier to design. GALS is an excellent solution for multi-component communication, and has shown promise in use on a smaller scale. Elastic clocks can provide improvement anywhere a clock is normally used.

Three methods of elastic clocks were demonstrated here on a bare bones CPU. The area overhead was 3% or less, and would only represent a smaller investment for a more complex CPU design. The power overhead was such that the energy savings gained by reducing test time negated it. Again, a more complex system would see more benefit in power. The execution time was reduced by about a third. This is more program dependent, but is indicative of the magnitude of time that can be saved by using this design. It may not match the throughput of a pipeline processor, but is much, much simpler and requires much less power while still having acceptable performance. Methods of improving performance with elastic clocks are also proposed, though not tested.

Regardless of the method chosen, gains can be made by changing how we time circuits. In a time where we can no longer simply increase the speed of the clock and multi-threading has proven difficult for many programmers, elastic circuits provide a way to improve throughput on a single core. In a time where system-on-chip design is gaining in popularity, these provide solutions for high speed communication between components. In a time where power

is a major focus, these methods can show significant reduction in power draw with minimal performance loss. Alternative timing schemes are a promising avenue for research, and warrant further investigation.

Bibliography

[1] D. Patterson and J. Hennessy, "Computer Organization and Design", 4th ed., Morgan Kaufmann Publishers, 2009

[2] Microprocessor Reference Guide, http://www.intel.com/pressroom/kits/quickreffam.htm

[3] S. Dropsho, G. Semeraro, D. Albonesi, G. Magklis and M. Scott, "Dynamically Trading Frequency for Complexity in a GALS Microprocessor", in *Proceedings of the 37th International Symposium on Microarchitecture*, 2004

[4] G. Semeraro, D. Albonesi, S. Dropsho, G. Magklis and M. Scott, "Improving Application Performance by Dynamically Balancing Speed and Complexity in a GALS Microprocessor" in *Workshop on Application Specific Processors*, December 2003

[5] Zhoukun Wang and Omar Hammami. "A 24 Processors System on Chip FPGA Design with Network on Chip", http://www.design-reuse.com/articles/21583/processor-noc-fpga.html

[6] John S. Seng, Eric S. Tune, Dean M. Tullsen, "Reducing Power with Dynamic Critical Path Information", in *Proceedings of the 34th International Symposium on Microarchitecture*, December, 2001

[7] Arm Infocenter, http://infocenter.arm.com/help/index.jsp

[8] J. P. Fishburn, "Clock Skew Optimization," in *IEEE Trans. Computers, vol. 39, no. 7*, July 1990

[9] T. Austin, V. Bertacco, D. Blaauw and T. Mudge, "Opportunities and Challenges for Better Than Worst-Case Design" in *Proceedings of the IEEE Asia South Pacific Design Automation Conference (AsP-DAC)*, Shanghai, 2005

[10] R. K. Uppu, R. T. Uppu, A. Singh, "Better-than-Worst-Case Timing Design with Latch Buffers on Short Paths", *27th International Conference on VLSI Design and 13th International Conference on Embedded Systems*, 2014

[11] J. D. Alexander, "Simulation Based Power Estimation for Digital CMOS Technologies", Masters Thesis, Auburn University, December 2008, Section 3.8

[12] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Dlaauw, T. Austin, K. Flautner and T. Mudge, "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation" in *36th Annual International Symposium on Microarchitecture (MICRO-36)*, December 2003

[13] Scott Hauck, "Asynchronous Design Methodologies: an Overview", in *Proceedings of the IEEE, Vol. 83, No. 1*, January 1995

[14] Jens Sparso, "Asynchronous Circuit Design: a Tutorial", Technical University of Denmark, 2006

[15] O. C. Akgun, J. Rodrigues, and J. Spars, "Minimum-Energy Sub-Threshold Self-Timed Circuits: Design Methodology and a Case Study" in *Proc. 16th IEEE Int. Symp. Asynchron. Circuits Syst., 2010, pp. 41-51*

[16] S.B. Furber, J.D. Garside, D.A. Gilbert: "AMULET3: A High-Performance Self-Timed ARM Microprocessor" Proceedings ICCD98 (October 5-7)

[17] Alain J. Martin, Mika Nystrm and Catherine G. Wong. "Three Generations of Asynchronous Microprocessors" in *IEEE Design & Test of Computers, special issue on Clockless VLSI Design*, November/December 2003

[18] Alain J. Martin, Mika Nystrm, Paul Penzes and Catherine G. Wong "Speed and Energy Performance of an Asynchronous MIPS R3000 Microprocessor" Caltech Technical Report, June 2001

[19] Alain J. Martin, "25 Years Ago: The First Asynchronous Microprocessor", Caltech Technical Report, January 2014

[20] R.F. Sproull, I.E. Sutherland, C.E. Molnar, "The counterflow pipeline processor architecture," Design & Test of Computers, IEEE, vol.11, no.3, pp.48, Fall 1994

[21] K. Van Berkel, et al., "Asynchronous Does Not Imply Low Power, But ...," Low-Power CMOS Design, A. P. Chandrakasan and R. Brodersen (Eds.), New York: IEEE Press, 1998, pp. 227-232

[22] F. Beest, A. Peeters, M. Verra K. van Berkel and H. Kerkhoff, "Automatic Scan Insertion and Test Generation for Asynchronous Circuits", in *IEEE Test Conference, 2002. Proceedings. International*

[23] S. Unger, "Asynchronous Sequential Switching Circuits", Wiley-Interscienct, 1969

[24] Chris J. Myers, "Asynchronous Circuit Design", John Wiley & Sons, Inc., 2001

[25] J. Carmona, J. Cotadella, M. Kishinevsky and A. Taubin, "Elastic Circuits", in *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 28, No. 10*, October 2009

[26] E. Tuncer, J. Cortadella and L. Lavagno, "Enabling Adaptability Through Elastic Clocks", in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, July 2009

[27] J. Cortadella, L. Lavagno, D. Amiri, J. Casanova, C. Macian, F. Martorell, J.A. Moya, L. Necchi, D. Sokolov, E. Tuncer, "Narrowing the margins with elastic clocks," *IC Design and Technology (ICICDT), 2010 IEEE International Conference on*, June 2010

[28] Edith Beign, F. Clermidy, S. Miermont, P. Vivet, "DVFS Architecture for Units Integration within a GALS NoC", CEA, 2008

[29] Augustus K. Uht, "Uniprocessor Performance Enhancement through Adaptive Clock Frequency Control", in *IEEE Transactions on Computers, Vol. 54, No. 2*, February 2005

[30] Michael L. Bushnell and Vishwani D. Agrawal, Essentials of Electronic Testing for Digital, Memory & Mixed-Signal VLSI Circuits, Kluwer Academic Publishers, 2000

[31] Priyadharshini Shanmugasundaram, *Test Time Optimization in Scan Circuits*, Masters Thesis, Auburn University, 2010

[32] Sindhu Gunasekar, *Finding Optimum Clock Frequencies for Aperiodic Test*, Masters Thesis, Auburn University, 2014

[33] Praveen Venkataramani, *Reducing ATE Test Time by Voltage and Frequency Scaling*, PHD Dissertation, Auburn University, 2014

[34] Marc Belleville and Cyril Condemine "Energy Autonomous Micro and Nano Systems", John Wiley & Sons, Inc., 2012

[35] R. Jaeger and Travis Blalock, "Microelectronic Circuit Design", 3rd ed., McGraw-Hill, 2008

[36] Multiple clock domain microprocessor, patent #US7739537 B2

[37] John Hansen and Montek Singh, "Multi-Token Resource Sharing for Pipelined Asynchronous Systems", *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012

[38] Ankur Agiwal and Montek Singh, "An Architecture and a Wrapper Synthesis Approach for Multi-Clock Latency-Insensitive Systems", *IEEE/ACM International Conference on Computer-Aided Design, ICCAD-2005*, 2005

[39] M. Singh, S.M. Nowick, "MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol.15, no.6, pp.684,698, June 2007

[40] Feng Shi, Y. Makris, S.M. Nowik, M. Singh, "Test Generation for Ultra-High-Speed Asynchronous Pipelines", *IEEE International Test Conference*, 2005

[41] Ivan E. Sutherland and Jon K. Lexau, "Designing Fast Asynchronous Circuits", *Seventh International Symposium on Asynchronus Circuits and Systems, ASYNC 2001*, 2001

[42] E. Kilada, S Das, K Stevens, "Synchronous Elasticization: Considerations For Correct Implementation and MiniMIPS Case Study", *VLSI System on Chip Conference (VLSI-SoC), 18th IEEE/IFIP*, 2010

[43] S.M. Nowick, M.E. Dean, D.L. Dill and M. Horowitz, "The Design of a High-Performance Cache Controller: a Case Study in Asynchronous Synthesis", *Conference on System Sciences, Proceeding of the Twenty-Sixth Hawaii International*, 1993

[44] "Advanced Configuration and Power Interface Specification", Copyright 2014-2015 Unified EFI, inc.

Appendices

Appendix A

VHDL Code Used for Proof of Concept

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

--This is the top level entity. Due to the way the memory was made, trying to synthesize this top level returned
--  nothing. The only way to get it simulated was to synthesize the Decode entity (with registers and ALU) and the
--  memory model separately and create a top level spice file by hand after layout.

entity CPU is
  port (CLK, RST, Prg: in std_logic;
        InstAdd: in std_logic_vector (9 downto 0);
    InstIn: in std_logic_vector (15 downto 0));
end entity CPU;

architecture CPUctrl of CPU is

  signal Inst, Data, WData: std_logic_vector (15 downto 0);
  signal InstMid, InstAddr, DataAddr: std_logic_vector (9 downto 0);
  signal WE: std_logic;

component Decode is
  port (CLK, RST: in std_logic;
  Inst, Data: in std_logic_vector (15 downto 0);
  WData: out std_logic_vector (15 downto 0);
  InstAddr, DataAddr: out std_logic_vector (9 downto 0);
  WriteEnable: out std_logic);
end component Decode;

component Mem is
  port (CLK, WE: in std_logic;
  Addr: in std_logic_vector (5 downto 0);
  I: in std_logic_vector (15 downto 0);
  O: out std_logic_vector (15 downto 0));
end component Mem;

begin
  InstMid <= InstAdd when PRG = '1' else InstAddr;
  DataMem: Mem port map (CLK, WE, DataAddr(5 downto 0), WData, Data);
  InstMem: Mem port map (CLK, Prg, InstMid(5 downto 0), InstIn, Inst);
  Conrtol: Decode port map (CLK, RST, Inst, Data, WData, InstAddr, DataAddr, WE);
end architecture CPUctrl;
```

```
--This is a very simple way of creating memory. This just makes a huge bank of registers. A real
--  memory would be significantly more complex, and significantly more efficient.

entity Mem is
  port (CLK, WE: in std_logic;
    Addr: in std_logic_vector (5 downto 0);
    I: in std_logic_vector (15 downto 0);
    O: out std_logic_vector (15 downto 0));
end entity Mem;

architecture Memory of Mem is
  subtype Word is std_logic_vector(7 downto 0);
  type MemBlock is array (0 to 63) of Word;
  signal Mem: MemBlock := ((others=> (others=>'0')));

begin
  process(CLK, Addr, Mem)
  variable Add: integer range 0 to 63;
  begin
    Add := conv_integer(Addr);
      if (CLK 'event and CLK = '1') then
      if (WE = '1') then
        Mem(Add+1) <= I(7 downto 0);
        Mem(Add) <= I(15 downto 8);
      end if;
    end if;
  O(7 downto 0) <= Mem(Add+1);
  O(15 downto 8) <= Mem(Add);
  end process;
end architecture Memory;
```

```
--The   Decode   file handles decoding and control for the CPU. It determines which registers go where,
--  when the write enable signals are used, what the next instruction will be, etc.

entity Decode is
  port (CLK, RST: in std_logic;
    Inst, Data: in std_logic_vector (15 downto 0);
    WData: out std_logic_vector (15 downto 0);
    InstAddr, DataAddr: out std_logic_vector (9 downto 0);
    WriteEnable: out std_logic);
end entity Decode;

architecture CPUctrl of Decode is

  signal Instruction, A, B, Bmid, I, ImmExt, AddImm, AddrExt, result, InstExt: std_logic_vector (15 downto 0);
  signal InstAdd, CInst: std_logic_vector (9 downto 0);
  signal Inst_mid, R1, R2, W1: std_logic_vector(3 downto 0);
  signal addsub: std_logic_vector (1 downto 0);

  component RegFile is
    port (CLK: in std_logic;
    R1, R2, W1: in std_logic_vector (3 downto 0);
    I: in std_logic_vector (15 downto 0);
    A: out std_logic_vector (15 downto 0);
    B: out std_logic_vector (15 downto 0));
  end component RegFile;

  component ALU is
    port (A: in std_logic_vector (15 downto 0);          -- input 1
    B: in std_logic_vector (15 downto 0);                -- input 2
    addsub: in std_logic_vector (1 downto 0);   --function select bits
    result: out std_logic_vector (15 downto 0));         -- output
  end component ALU;

begin
  Reg: RegFile port map (CLK, R1, R2, W1, I, A, B);
  Math: ALU port map (A, Bmid, addsub, result);
  CurrentInst <= Instruction;
  Inst_mid <= Instruction(15 downto 12);
  InstAddr <= InstAdd;
  InstExt(15 downto 10) <= "000000";
  InstExt(9 downto 0) <= InstAdd;
  ImmExt(15 downto 8) <= "00000000";
  ImmExt(7 downto 0) <= Instruction(7 downto 0);
  AddImm (15 downto 4) <= "000000000000";
  AddImm (3 downto 0) <= Instruction (3 downto 0);
  addsub <= "01" when Inst_mid = "0110" else "00" when Inst_mid = "0001"
    else "00" when Inst_mid = "0010" else Instruction(13 downto 12);

  WriteEnable <= '1' when Inst_mid = "0010" else '0';
  WData <= B;
  R1 <= Instruction(7 downto 4);
  R2 <= Instruction(11 downto 8) when Inst_mid = "0010" else Instruction(3 downto 0);
  W1 <=  "0000" when Inst_mid = "0101" else "0000" when Inst_mid = "0110"
    else "1111" when Inst_mid = "1001" else "0000" when Inst_mid = "1010"
    else "0000" when Inst_mid = "0010" else Instruction(11 downto 8);
  I <= Data when Inst_mid = "0001" else ImmExt when Inst_mid = "0011"
    else InstExt when Inst_mid = "1001" else result;
  Bmid <= AddImm when Inst_mid = "1000" else AddImm when Inst_mid = "0001"
    else AddImm when Inst_mid = "0010" else B;
  DataAddr <= result(9 downto 0);

  InstAdd <= Instruction(9 downto 0) when Inst_mid = "1001" else  A(9 downto 0) when
    Inst_mid = "1010" else CInst + 4 when (Inst_mid = "0101" and result = "000000000000000") or
    (Inst_mid = "0110" and result(15) = '0') else "0000000000" when RST = '1' else CInst + 2;

--When selecting which instruction to execute next, note the branches always select PC+4. Branches here
--  simply skip over an instruction, namely a jump. If branch is true, go to the   next   instruction.
--  If false, then jump to another part of the code.

  process (CLK, RST) begin
    if (CLK 'event and CLK = '1') then
      CInst <= InstAdd;
      Instruction <= Inst;
    end if;
  end process;
end architecture CPUctrl;
-------------------------------------------------------------------------------------------------------------------
--This ALU was taken from an old project by Myers Hawkins and myself. There is no multiplication or division.
--  The fact that there are only 2 select bits is used in the control for a variety of simplifications.

entity ALU is
  port (A: in std_logic_vector (15 downto 0);  -- input 1
    B: in std_logic_vector (15 downto 0);  -- input 2
    addsub: in std_logic_vector (1 downto 0); --function select bits
    result: out std_logic_vector (15 downto 0)); -- output
end entity ALU;

architecture arith of ALU is
-- add when select input is 00
-- subtract when select input is 01
-- logical AND when select input is 10
-- logical OR when select input is 00
```

```
begin
  result <= std_logic_vector(signed(a) + signed(b)) when addsub = "00" else
  std_logic_vector(signed(a) - signed(b)) when addsub = "01" else
  A and B when addsub = "10" else
  A or B when addsub = "11";
end architecture arith;
```

```
--This register file was adapted from a project by T.J. Lowery. An H-Tree had to be inserted by hand
--  into the spice model.

entity RegFile is
  port (CLK: in std_logic;
    R1, R2, W1: in std_logic_vector (3 downto 0);
    I: in std_logic_vector (15 downto 0);
    A: out std_logic_vector (15 downto 0);
    B: out std_logic_vector (15 downto 0));
end entity RegFile;

architecture Regs of RegFile is
  subtype SingleRegister is std_logic_vector(15 downto 0);
  type RegisterArray is array (0 to 15) of SingleRegister;
  signal AllRegisters: RegisterArray := ((others=> (others=>'0')));

begin
  process(CLK, R1, R2, W1, AllRegisters)
    variable R1Add: integer range 0 to 15;
    variable R2Add: integer range 0 to 15;
    variable WAdd: integer range 0 to 15;
  begin
    R1Add := conv_integer(R1);
    R2Add := conv_integer(R2);
    WAdd := conv_integer(W1);

    AllRegisters(0) <= "0000000000000000";

    if (CLK 'event and CLK = '1') then
      if (W1 /= "0") then       --Cannot overwrite R0, so W1 = "00' means no write
        AllRegisters(WAdd) <= I;
      end if;
    end if;

    A <= AllRegisters(R1Add);
    B <= AllRegisters(R2Add);
  end process;
end architecture Regs;
```

```
--This is used to combine the Clock Generator with the width selector.
--The selector would normally be a small memory bank, but was synthesized as a set
--  value for each instruction to simplify the simulations.
entity ClockTop is
  port(CLK_in, RST: in std_logic;
    Inst: in std_logic_vector(3 downto 0);
    CLK: out std_logic);
end entity ClockTop;

architecture CT of ClockTop is

  signal CMP: std_logic_vector (2 downto 0);
component ClockInt is
  port (CLK_in, RST: in std_logic;
  CMP: in std_logic_vector(2 downto 0);
  CLK_out: out std_logic);
end component ClockInt;
component CmpSel is
  port (Inst: in std_logic_vector (3 downto 0);
  CmpOut: out std_logic_vector (2 downto 0));
end component CmpSel;

begin
  C: ClockInt port map(CLK_in, RST, CMP, CLK);
  CS: CmpSel port map(Inst, CMP);
end architecture;
```

```
--This is a simple counter which toggles the output bit and resets when it matches the input.
entity ClockInt is
  port (CLK_in, RST: in std_logic;
    CMP: in std_logic_vector(2 downto 0);
    CLK_out: out std_logic);
end entity ClockInt;

architecture CLKINT of ClockInt is
  signal count: std_logic_vector(2 downto 0);
  signal CLK_mid: std_logic;

begin
  CLK_out <= CLK_mid;
  process (CLK_in, RST) begin
    if (CLK_in 'event and CLK_in = '1') then
      if (RST = '1') then
            count <= "000";
                CLK_mid <= '0';
          elsif (count >= CMP) then
```

```
                count <= "000";
                CLK_mid <= not(CLK_mid);
          else
                count <= count+1;
          end if;
        end if;
  end process;
end architecture;
```

```
--This is a very crude version of the multiphase generator/selector combo. A issue with the synthesis
--  software meant the clock gate was tied to the reset signal in a way that caused a glitch.
--  This glitch required manually changing a complex gate to an and-or combo to work. As it is,
--  this is not an optimized implimentation, but it works.
entity MPhase is
  port( CLK_in, RST: in std_logic;
    Inst: in std_logic_vector(3 downto 0);
        CLK_out: out std_logic);
end entity MPhase;

architecture MPGen of MPhase is
  signal CLK_mid, CLK0, CLK1, CLK2, CLK3, CLK4, CLK5, CLK6, CLK7: std_logic;
  signal PCount, CLKsel, Sel: std_logic_vector(2 downto 0);

component CmpSel is
  port (Inst: in std_logic_vector (3 downto 0);
        CmpOut: out std_logic_vector (2 downto 0));
end component CmpSel;

begin
  S: CmpSel port map(Inst, Sel);
  CLK_out <= CLK_mid;
  CLK_mid <= CLK_in when RST = '1' else CLK0 when CLKsel = "000" else CLK1 when CLKsel = "001" else
        CLK2 when CLKsel = "010" else CLK3 when CLKsel = "011" else
        CLK4 when CLKsel = "100" else CLK5 when CLKsel = "101" else
        CLK6 when CLKsel = "110" else CLK7 when CLKsel = "111" else CLK7;
  CLK0 <= CLK_in when PCount = "000" else '0';
  CLK1 <= CLK_in when PCount = "001" else '0';
  CLK2 <= CLK_in when PCount = "010" else '0';
  CLK3 <= CLK_in when PCount = "011" else '0';
  CLK4 <= CLK_in when PCount = "100" else '0';
  CLK5 <= CLK_in when PCount = "101" else '0';
  CLK6 <= CLK_in when PCount = "110" else '0';
  CLK7 <= CLK_in when PCount = "111" else '0';

  process(CLK_in, CLK_mid, RST) begin
        if (CLK_in 'event and CLK_in = '1') then
          if (RST = '1') then
                PCount <= "000";
          elsif (PCount = "000") then
                PCount <= "001";
          elsif (PCount = "001") then
                PCount <= "010";
          elsif (PCount = "010") then
                PCount <= "011";
          elsif (PCount = "011") then
                PCount <= "100";
          elsif (PCount = "100") then
                PCount <= "101";
          elsif (PCount = "101") then
                PCount <= "110";
          elsif (PCount = "110") then
                PCount <= "111";
          elsif (PCount = "111") then
                PCount <= "000";
          end if;
        end if;
        if (CLK_mid 'event and CLK_mid = '0') then
          if (Sel = "000") then
                CLKsel <= CLKsel + 1;
          elsif (Sel = "001") then
                CLKsel <= CLKsel + 2;
          elsif (Sel = "010") then
                CLKsel <= CLKsel + 3;
          elsif (Sel = "011") then
                CLKsel <= CLKsel + 4;
          elsif (Sel = "100") then
                CLKsel <= CLKsel + 5;
          elsif (Sel = "101") then
                CLKsel <= CLKsel + 6;
          elsif (Sel = "110") then
                CLKsel <= CLKsel + 7;
          elsif (Sel = "111") then
                CLKsel <= CLKsel;
          end if;
        end if;
  end process;
end architecture MPGen;
```

```
--This is a version of the stop clock method. The stop clock name is a bit misleading here, as the clock is
--  not passed through and stopped. When the count is "0", the output is '1'. Otherwise, it is '0'. This
--  provides a more stable output at the cost of increasing the minimum output clock to 2 input cycles.
entity SClock is
```

```
  port( CLK_in, RST: in std_logic;
    Inst: in std_logic_vector(3 downto 0);
    CLK_out: out std_logic);
end entity SClock;

architecture SCLK of SClock is
  signal count, CMP: std_logic_vector(2 downto 0);
  signal CLK_mid: std_logic;

component CmpSel is
  port (Inst: in std_logic_vector (3 downto 0);
    CmpOut: out std_logic_vector (2 downto 0));
end component CmpSel;

begin
  S: CmpSel port map(Inst, CMP);

  CLK_out <= CLK_mid;
  process (CLK_in, RST) begin
    if (CLK_in 'event and CLK_in = '1') then
      if (RST = '1') then
        count <= "000";
        CLK_mid <= '1';
      elsif (count >= CMP) then
        count <= "000";
        CLK_mid <= '1';
      else
        count <= count+1;
        CLK_mid <= '0';
      end if;
    end if;
  end process;
end architecture SCLK;
```

```
--This selection circuitry was used by both methods described above, and was determined by the timing simulation
-- and the 0.65ns input clock that would be used. Using programmable memory would be more useful, but take
-- more space. It might be faster or slower, depending on the speed of a memory read vs the speed of the
-- following combonational circuit.
entity CmpSel is
  port (Inst: in std_logic_vector (3 downto 0);
  CmpOut: out std_logic_vector (2 downto 0));
end entity CmpSel;

architecture Cmp of CmpSel is
begin
  with Inst select
    CmpOut <= "010" when "0001",
    "010" when "0010",
    "001" when "0011",
    "110" when "0101",
    "110" when "0110",
    "101" when "1000",
    "001" when "1001",
    "001" when "1010",
    "101" when "1100",
    "011" when "1101",
    "011" when "1110",
    "011" when "1111",
    "111" when others;
end architecture;
```