DESIGN OF AN APPLICATION-SPECIFIC INSTRUCTION PROCESSOR FOR THE GFM SCUBA DIVING ALGORITHM

A thesis submitted to the faculty of
San Francisco State University
In partial fulfillment of
The Requirements for
The Degree

Master of Science
In
Engineering: Embedded Electrical and Computer Systems

by

Edward S Pataky

San Francisco, California

December, 2009

CERTIFICATION OF APPROVAL

I certify that I have read *DESIGN OF AN APPLICATION-SPECIFIC INSTRUCTION PROCESSOR FOR THE GFM SCUBA DIVING ALGORITHM* by Edward S Pataky, and that in my opinion, this work meets the criteria for approving a thesis submitted in partial fulfillment of the requirements for the degree: Master of Science In Engineering: Embedded Electrical and Computer Systems at San Francisco State University.

Hamid Mahmoodi
Assistant Professor of Electrical Engineering

Hamid Shahnasser
Professor of Electrical Engineering

DESIGN OF AN APPLICATION-SPECIFIC INSTRUCTION PROCESSOR
FOR THE GFM SCUBA DIVING ALGORITHM

Edward S Pataky
San Francisco, California
2009

Abstract: The recently patented Gas Formation Model scuba diving decompression algorithm presents a challenge in terms of practical design for ultra-low-power battery-operated scuba dive computer applications. In its original form, it requires thousands of floating-point calculations to be completed within hard real-time bounds, in order to calculate the diffusion and convection of nitrogen gas in the human body. Combining software optimization techniques with a customized processor core design, this study presents the first attempt to design custom hardware for GFM in order to reduce the energy and area footprint of the processor required, while at the same time meeting aggressive performance goals. The resulting design outperforms ARM9 and ARM7 devices by roughly a factor of 2.5 to 3, amounting to 60%-66% speedup, respectively, with a reasonable gate count and meeting the specifications given in the study.

I certify that the Abstract is a correct representation of the content of this thesis.

Chair, Thesis Committee | Date

ACKNOWLEDGEMENT

This research on efficient hardware design options for the GFM algorithm was done in conjunction with work done on the algorithm itself and its prototype hardware system implementation as an embedded systems design engineer at Pelagic Pressure Systems in San Leandro, CA. The author would like to give special thanks to Pelagic Pressure Systems, and to Dr. Hamid Mahmoodi of San Francisco State University for help and support of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: The Application

Scuba Diving - Introduction

Scuba diving is a sport that involves diving underwater while breathing a particular mixture of pressurized gas. Some divers stay at shallow depths (less than 30 ft) for relatively short times (less than 20 minutes), while others dive to relatively deep depths for considerably longer times. Many so-called "recreational divers" engage in this activity simply for pleasure, while other divers dive as part of their job, or for military purposes. The term "technical diver" refers to divers that dive to great depths, for example inside underwater caves, and follow particular scheduled stops for safety as they ascend to the surface. There are many types of diving and types of divers, but one thing modern divers have in common in recent years is the use of a so-called "dive computer", which will be explained later in this paper. For now we will use the definition of a dive computer being a device used by scuba divers in order to keep track of the state of the gas in the diver's body, and to otherwise provide a safe path to the surface at all times under any situation.

When a diver is under water, as the surrounding ambient pressure on the diver's body increases at depth, the higher concentrations of oxygen and nitrogen molecules in the breathing mixture diffuse into the blood and tissues of the diver's body, and as the diver surfaces, they go in the reverse direction, always seeking an equilibrium state. To understand this more fully, we can consider the Ideal Gas Law [24] from basic chemistry. The law states that the pressure (p) times the volume (V) of an ideal gas is equal to the number of moles of gas (n) times the Universal Gas Constant (R) times the temperature (T) in Kelvin.

This is more simply stated as given in equations (1) and (2).

$pV = nRT$ Equation 1

$n = pV / RT$ Equation 2

When a diver goes underwater to some depth, the weight of the water above the diver contributes to an increased pressure on the diver, proportional to depth. Considering the same (or only slightly different) temperature, and that the average volume in the diver's lungs does not change, we can see from Equation 2 that since V/RT is constant, and p is increasing, the only conclusion is that (n) is also increasing. In other words, when a diver is at depth, there is an increased number of moles of gas in the diver's lungs. To say there is an increased number of moles of gas in an unchanging volume (the lungs) means that there is a higher *concentration* of that gas in that volume. That gas could be any number of mixtures commonly used for diving, but for simplicity let us consider that common air (roughly 21% Oxygen and 79% Nitrogen) is used. The measure of concentration used in the diving industry is called the “partial pressure”, denoted PPX (where X denotes the gas, ie... PPO2), and is calculated as in Equation 3.

PPX = Abs. Pressure * % Gas X in mixture Equation 3

For example, at 6 bar of absolute pressure, in normal air, the partial pressure of Oxygen, termed PPO2, is 1.26 bar absolute, and for Nitrogen (PPN2) is 4.74 bar absolute. Note that there are certain known safety limits for example on PPO2 levels, where breaching those limits can cause maladies such as oxygen toxicity .

The diver's lungs have an increased concentration of gas at depth and we note that the diver's blood and tissues initially do not. If the diver stays at depth for any amount of time, the gas in the lungs with higher concentrations of oxygen and nitrogen will start to dissolve into the the surrounding blood and tissues in the diver's body. Eventually if the diver stays at the same depth for a long enough time, the concentrations in the lungs and in the blood and tissues of the body will be equalized. The different parts of the body have different characteristics of solubility and diffusivity that govern how quickly this occurs, but it occur nevertheless. In other words, a diver that stays at depth for some significant amount of time has an increased concentration of oxygen and nitrogen in their blood and tissues.

The second part to this story is that when the diver surfaces or otherwise ascends to an area of lesser surrounding ambient pressure, the higher concentrations of gas that are in the blood and tissues of the diver's body now come “out of solution” and are exhaled through the lungs. A danger exists when excess nitrogen is coming out of solution as the ambient pressure decreases, whereby if the diver is not careful it can come out too quickly and form free gas in the body, and the diver can get decompression sickness (DCS) [1]-[4]. DCS is commonly referred to as “the bends”, and can be severely painful, and can be fatal. The cause of DCS is believed by many as having to do with free nitrogen gas forming in the body[1][3]. Various theories[1] [3][4] suggest differences in terms of when, where, how, and in what form the free gas develops, but the general belief is that in fact it is this free nitrogen gas that causes it. As far as the recently patented Gas Formation Model (GFM)[1] is concerned, it is precisely that free gas that causes DCS and GFM actually calculates in real-time when and under what conditions fee gas will form, as well as precisely how much is formed.

One function of a scuba diving computer [4] is to compute the amount of time a diver can stay at the depth before surfacing would result in the diver getting sick. The Gas Formation Model calculates the diffusion of nitrogen gas into and out of the blood and tissues in the body and through he lungs, and the precise time that nitrogen gas layers break and allow nitrogen to be free in the body. The amount of free gas volume (FGV) is calculated and correlated to the percent probability of getting DCS. An important function, then, of a GFM-based dive computer is to use the GFM algorithm to calculate the amount of free gas in the body, and to predict the future free gas volume (given various future dive profiles), in order to give the diver a safe path to the surface, under any situation. In addition to these predictive calculations, GFM also can be used to give a quantitative measure of free gas in the body in real-time.

Note that GFM is truly a real-time algorithm, in that it uses ambient pressure as well as real physiological input parameters to calculate real quantities of free gas in the diver's body in real-time. This FGV calculation is a novel approach to the design of a dive computer, and exactly how intensive the algorithm is and what is required to implement it efficiently in hardware is the subject of this thesis. This study represents the first such attempt to explore a custom hardware design for GFM.

Definition of Terms

Before we continue, a brief definition of terms used in this paper is in order, and so the concepts of an "immediate ascent", "future prediction", and a "dive profile" will be defined here. A "dive profile" is a loose term for a set of points, (time, depth), for which we require a calculation of free gas. Note that the dive profile is typically expressed not in (time, depth) form, but in the sentence form "<depth> ft for <time> minutes". For example, suppose a a diver has descended to a depth of 100ft, and has been there for 5 minutes. Then suppose the diver ascends to the surface where he stays for 10 minutes. The dive profile in this case is then "100 ft for 5 minutes, then 0 ft for 10 minutes". Dive profiles are used to describe the input vectors for the GFM algorithm and its application, and they form the basis of how GFM is tested and analyzed.

A "future prediction" in this paper will be used to refer to the calculation of free gas, as calculated by GFM, corresponding to the set of (time,depth) inputs for a particular *future* dive profile. These predictions are calculated exactly the same way as the real-time state of the algorithm is calculated, only that prior to execution, a copy of the algorithm state is made and prediction calculations are executed only on the copied data, so as to not affect the real-time state calculation. We will use the term "immediate ascent" (IA) prediction for a special type of future prediction where the dive profile is such that the diver is assumed to immediately ascend to the surface. This is as opposed to assuming the diver stops at some intermediate depth(s), or stays at the current depth for any amount of time.

How GFM Works

To understand how GFM works, we will carefully review the representation in Figure 1. Figure 1 shows two plots with Depth on the bottom half and FGV on the top half, with the x-axis representing time. The bottom half depicts a dive profile (as defined previously). Suppose the maximum depth is 100ft as in the figure, and at time t1 the diver surfaces. The free gas computation corresponding to that dive profile will have some maximum peak value shown as Max1 at some time t1'. Suppose, the diver stayed at 100ft for another 5 minutes until time t2 (t1 + 5 minutes). For the longer dive there might be a larger peak value of FGV (depending on the circumstance), and the maximum value might be Max2 (> Max1) at time t2'. Now if we consider that a diver is in fact at 100ft at time t1, and these two future dive profiles with corresponding peak FGVs are calculated as predictions while the diver is at depth, we start to understand how GFM works. In this way GFM can be used not only to calculate the real-time FGV in the diver's body, but also and more importantly, the FGV that would result given certain future dive profiles.

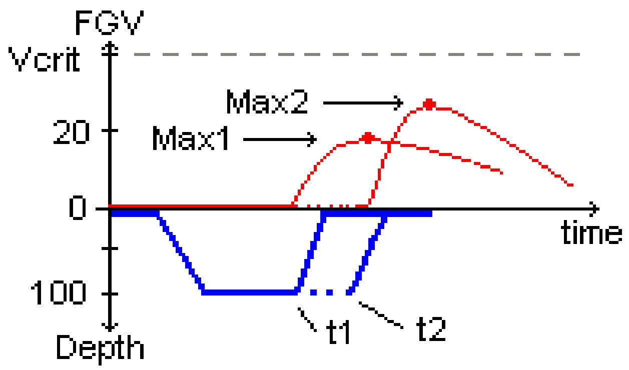


Figure 1: GFM Free Gas Predictions (conceptual)

The last piece of the puzzle is to find the time-to-decompression or no-decompression time remaining, important outputs of any modern dive computer. To do that using GFM, the points (t1', Max1), (t2', Max2), … (tn', MaxN) can be used to predict the time when the peak FGV will breach a certain critical value, shown as Vcrit in the figure. If you had unlimited computational capacity you could run an unlimited number of FGV predictions and pinpoint the NO-D time without much effort, but in reality only a limited number of predictions can be run. In fact, in the simplest case you could simply run one IA prediction and simply report when the time arrives at which an IA to the surface would breach the critical value. The simple approach (not making use of GFM utilize its prediction capability) would provide an accurate real-time result, however without much warning to the diver. How many predictions need to be run to give ample warning to the diver depends on the system design and trade-offs in how accurate the system needs to be, and how much time and power can be spent on computation. In an application such as a top-of-the-line dive computer, it may be required to provide a warning well in advance for multiple future dive profile scenarios. In other applications that are not as time-critical or safety-critical, the last minute update may be sufficient. In the past, some divers relied on their own hand calculations and tables of pre-calculated values for safe dive times and decompression schedules, but modern dive computers are so advanced that many divers now rely solely on the information they provide for their safety. This thesis will consider the case of a safety-critical, real-time, high-end dive computer where the most accurate and fastest possible response time is most desirable.

We can note here that the GFM patent describes the GFM algorithm very well in terms of its formulation, background and concepts, input and output parameters, as well as a detailed listing of the main algorithm routine written in high level code. That document will be referenced here, and the remaining sections will assume some familiarity with the detailed information provided in that document. Note also that for the purposes of this thesis, we are not as much interested in how the algorithm was developed, or in comparisons to other similar algorithms or validation of GFM itself, rather, we will assume its validity and focus instead on the use and performance of the algorithm. In addition, the subject matter of this paper will focus more on how to achieve the highest or otherwise most suitable performance of the GFM algorithm running inside a modern, low-power, battery operated dive computer.

GFM Example – Real-Time FGV Calculation

To really drive home the meaning of the concepts in the previous section, let's first follow a more detailed example of the simplest case of GFM's in-application use. What we want to consider here is what GFM calculates, and when it calculates it, in order to keep track of the real-time FGV in the diver's body. At the simplest level, this is the basic piece of information every GFM application needs in order to function. Additional predictions can be performed, but only if the real-time state is and has been calculating from the start of the dive or (set of dives if more than one dive).

Let's assume then that a diver (let's call him John) has just arrived at a location where he plans to make a dive and he is on the surface, at 0 FT of depth. We will further assume that John's body has had enough time to equalize and otherwise adjust to the altitude he is at, and that this is the first dive of the day, and there are no previous dives even the day before. This information is important for any dive computer because to be accurate, unless the computer has been keeping track of the state of the gas in John's body from changing altitude, previous dives made, etc ... it has to assume that everything is as described above. So John is about to make a dive and he turns on his GFM dive computer. The GFM computer will initialize and start keeping track of time and depth.

As used in the routine given in the GFM patent, we will use 0.1 minutes as the time step for the GFM routine. In other words, every 6 seconds, the GFM routine needs to run in order to keep track of the real-time state. Note that this means the routine needs to complete its computation in no more than 6 seconds, before the time the next routine call starts. We can visualize this using the plot shown in Figure 2. The figure shows a dive profile, starting at the surface (0 ft), and going to a depth of 100ft. Notice that each time the basic GFM routine runs (shown as a program function f(x)), there is a dot marking the execution time. Each execution is separated by a time difference, labeled dt, where dt=6s. Note that each time the routine runs, it runs in its entirety. Each time the routine runs, it needs only the current depth or pressure input at that time, the current time, and any other constants defined during GFM initialization.

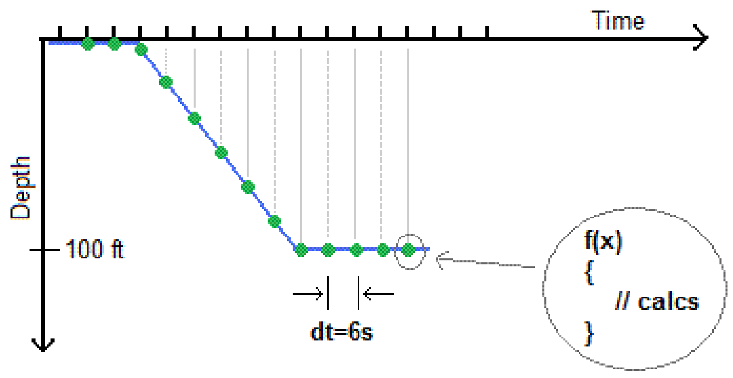


Figure 2: GFM Routine Execution (conceptual)

Using real-time programming vernacular, this type of basic algorithm routine will be loosely termed the algorithm's “kernel” or real-time “tick”, in this case representing the most basic piece of the larger program. The larger GFM program would include the real-time tick, as well as higher level algorithms that include running predictions and any other routines that are needed to calculate the dive computers outputs.

Now, following Figure 2, consider that John completes the dive by ascending to the surface, possibly stopping at intermediate depths on the way up. The GFM kernel continues to run every 6 seconds, and every 6 seconds the real-time state of the gas in John's body is up to date. This computation is essentially all that is needed to keep the state of the John's body current. As mentioned previously, a high-end dive computer based on GFM would require more computations. At a minimum, a high-end GFM-based computer would need to compute predictions of future dive profiles to give John real-time feedback on the consequences of future actions.

GFM Example – Prediction Calculations

Let's suppose our diver John is at 100 ft and has been there for about 15 minutes. John may consider staying at that depth for another 15 minutes. The problem is that this amount time at that depth (assuming the breathing mixture is air), could force John to be required to take what are called decompression stops on the way up. This may or may not be an issue for John. For example, if John has 20 minutes of air left in his tank, but the combination of his extra 15 minutes at depth, plus the time needed to complete the decompression stops means that he would not have enough air to complete the stops, then John will be in trouble if he in fact stays at 100ft for another 15 minutes. This is an example where predictive calculations done at the current time could provide John with vital information about the future.

In fact, one of the most basic outputs of modern dive computers is the Time-To-Decompression, or No-D Time, and is the time remaining at the current depth for which the diver can stay and simply ascend to the surface without taking any intermediate stops (decompression stops) to avoid DCS. In other words, if John's computer says that the NO-D time is 5 minutes, that means that John has 5 minutes where he can remain at depth, then ascend to the surface, without making additional stops on the way up. If John chooses to ignore the NO-D time given by the dive computer, he can certainly do that, but then he will need to perform decompression stops, or otherwise ascend at a specific rate such that he avoids DCS. As mentioned previously, how many predictions or future options are run depends on the application and the dive computer manufacturer's choices made during the design of the specific product. In the dive computer industry, it is generally expected almost by definition that a dive computer at least provides the NO-D time, as well as decompression schedules once that time is exceeded. Beyond that, although a predictive algorithm such as GFM can allow for providing more information, most dive computers traditionally have not provided much additional information beyond simple calculations such as PPO2 levels, breathing rates, and other simple calculations.

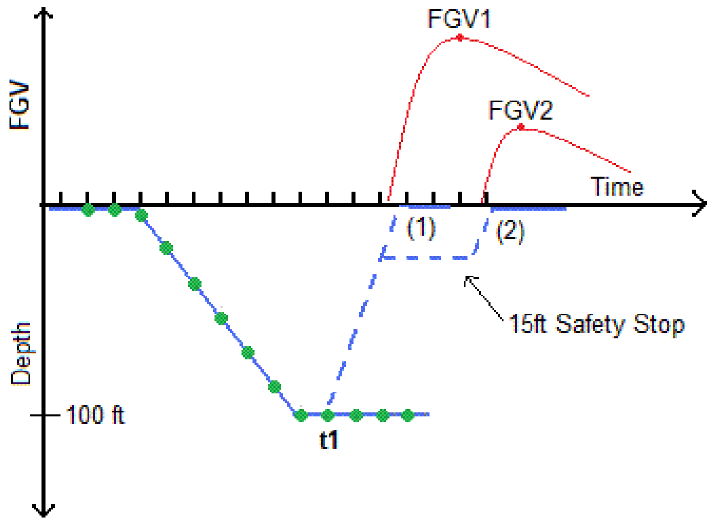


Figure 3: Prediction Calculations (conceptual)

Suppose, for example, that at every tick, additional prediction calculations are run that determine the NO-D time, plus the consequence of taking so-called Safety Stops [3][4] on the way up. Safety Stops are short (< 5 minute) stops made typically at one shallow depth on the way up to the surface. These are recommended by dive instruction and safety institutions such as PADI [5], and are generally followed by divers as a safe practice measure. If a GFM computer ran safety stop predictions, it could not only give John the NO-D time, but also the consequence of taking Safety Stops at different depths, or of different lengths of time, for example. Given that information, John could then decide, for example, that a 1 minute stop at 15 ft is sufficient, as opposed to a longer time that might be recommended. General rules of thumb such as the "3 minutes at 15 ft" safety stop are meant to be a safety measure under a wide range of conditions, however, a true real-time prediction of specific safety stop times and their consequences might be more desirable and useful to the diver.

Figure 3 depicts a dive to 100ft, followed by some time at depth, where the prediction calculations are illustrated at one point in time. The dotted lines represent two different prediction calculations, run at time t1. The first prediction labeled (1) follows the path from the bottom depth up to the surface, without any stops. The second ascent labeled (2), follows a path up to 15 ft for a safety stop, then up to the surface. Also labeled on the upper plot are the resulting free gas volumes corresponding to the predicted ascents (1) and (2). FGV1 is the FGV value that would result in the diver's body if the diver followed path (1) and FGV2 is the FGV value that would result if the diver followed path (2). Note that since GFM correlates the amount of FGV to the probability of getting DCS (PDCS), and because we know that safety stops are recommended as a general safety measure, it makes sense that the plot shows the maximum value from path (2) smaller than the maximum from path (1). In other words it should be clear why FGV2 < FGV1 after taking a safety stop. Since the max FGV value is correlated to PDCS by GFM, it should be clear why this information would be useful to John as he decides what to do precisely at time t1 at depth. If John knows that ascending would produce say 30 ml of free gas (or X percent PDCS), but that if he takes a safety stop on the way up the FGV would be say 5 ml (or Y percent PDCS; $Y < X$), then John has a concrete measure of how effective the safety stop would be. In fact, you can now understand why the more predictions that can be run, the better informed John can be, and the better equipped he is to make the right decisions during his dive. It should also be clear that if the calculation cannot be run efficiently and provide output in a timely manner, that they are useless to John as he tries to make a safety-critical decision.

Dive Computer Systems (General)

Personal Dive Computers (PDCs) have already been defined in terms of purpose and function, and so this section will attempt to describe what a dive computer is as an electronic product. A dive computer is typically a small embedded computing device with 1 to 4 user input buttons, some type of segmented or dot matrix user interface display, and in recent years a PC communication interface. A general-purpose processor is typically used as a master and other processors/devices as slaves. The master handles I/O, timing and clocks, A/D conversion for sensor input, display control, flash memory for data storage, and virtually everything the dive computer needs. Most if not all current devices handle existing algorithm computation from within the main general-purpose processor, due to the relative simplicity of existing algorithms.

PDCs come in many size and shapes, and are made by a few different manufacturers in few different countries. Some of the first and oldest devices were relatively large in size, and used relatively large batteries for power. Display screens were much larger than today, and processing power was much more limited. As a result, the first dive computers had very limited capabilities, but still proved useful to divers who did not want to have to hand calculate dive times and schedules. PDCs took off over the last 20 years and today we have companies that offer everything from the low end basic computer under $100 to high-end, wireless/air-integrated, top-of-the-line units that sell for thousands of dollars. Some units are still hand-held, but they also come in wrist-top units and low-profile watch-style designs as well. Some designs use sophisticated color dot-matrix displays, with multi-featured PC interface and download software, offering everything from so-called *pre-dive planning* to dual-algorithm selection and user customization. For more information a good source of information is the latest article from Scuba Diving Magazine [6] that evaluates the latest and greatest dive computers for 2009 from some of the top manufacturers.

A GFM Dive Computer System

Scuba diving and dive computers in general have been discussed, and this section will focus now on what a GFM computer is and specifically what is meant by a GFM PDC in this study. As mentioned previously, the PDC designer has freedom to design the device in many different ways, depending on the specific requirements of the product being designed, but in this study we consider only the most demanding type of design in terms of power, performance, and area footprint. To begin, we start with the understanding that a GFM dive computer is first and foremost, a dive computer. That means at a high-level, a GFM-based dive computer calculates many of the same things that most all dive computers do. This includes the NO-D time, decompression schedules, PPO2 levels, breathing rates and ascent/descent rates, oxygen toxicity, and more.

In terms of hardware, we further assume the device has some sort of display, whether it be a simple segmented unit for digits and icons, a dot matrix display for graphical output, or a combination of both. We will assume some sort of communication and download interface to a PC, whether it be wired or wireless. We also assume the unit includes several user input buttons for control of the device and general user input. We assume the device includes some sort of on-board memory for storage of dive data, settings, constants, and other information needed by the algorithm or operating system. In terms of external packaging, we consider a device that has a low-profile watch-style design that can only be powered by a very small non-rechargeable coin-cell battery. The batteries used for power in these type of designs are typically 3V with a recommended (short duration) peak current draw of around 20mA, a recommended continuous current draw of less than 1 mA, and a less than 300 mAh capacity.

A GFM PDC needs to be running a real-time operating system of some form. At a basic level, GFM requires that pressure and time are input on a pre-determined schedule. For this study we will consider that the basic time-step or “tick” interval is 0.1 minutes (6 seconds), as given in the GFM patent description. This means that no matter what else is calculated on other timescales, the real-time state of the algorithm needs calculate on a timescale of 6s or less. We also know the importance of prediction calculations for the outputs of a GFM-based PDC. Because of this, we will set the goal that the 6s interval is the timescale in which the real-time state, as well as all prediction calculations must complete. This could be done using more complicated schemes whereby data is interpolated between longer timescales of computation, but to take full-advantage of GFM and to be as responsive as possible, we will say that within 6s, we want the real-time state plus all computation for all PDC outputs to be complete.

Note that for a low-power battery-operated system we would want the resource-intensive computation portion of operation to take up only a fraction of the time in an active/on state (for example at 5% duty), but still as a strict upper-limit real-time bound, we will use 6s. Duty cycle in this case refers to what percent of the time-step timescale (6s) the device is actually active and running. This is in contrast to when the calculations are completed, and the device resumes a low-power state. It is important to note then, that in a low-power device such as a PDC, any opportunity for using a low-power system state is desired for maximum battery life. So for example, although we have 6s to complete the real-time state calculation as well as predictions, if we can finish in 1s, or say 1/3 of a second, that would be even better. The more time the processor can stay off, the longer the battery will last. This type of energy consumption calculation will be specified more clearly in later sections.

C-Code Implementation of the GFM Algorithm

The original GFM algorithm as written by its inventors, is a VB-Code based routine that simply runs the core GFM routine, and does nothing more. This is necessary but not sufficient to run a GFM-based dive computer. There are a couple reasons for this. One reason is that although the VB-code was designed for and runs perfectly well on a PC with relatively large memory and high processor power, for a real embedded device running on a coin-cell it is not optimized in any way. All variables are double-precision floating-point, and for a low-power application this presents a prohibitively large memory footprint.

Secondly, the routine presented in the patent is not designed for a real-time embedded system, rather, it is simply a procedural routine that starts with inputting parameters for a specific dive, and ends with outputting values to a file on a PC. In a real system, real pressure data from analog sensors is sampled at regular intervals, and time is broken up into slots for the operating system to be able to handle other tasks in addition to GFM. A practical GFM algorithm could not simply run through all calculations at once to compute the needed PDC outputs, rather, routine calls would need to be carefully and precisely scheduled to make the best use of processing power, and to reduce battery current draw.

Lastly, the routine in the patent represents only the simplest use of GFM, calculating through one pre-determined dive profile. There are no predictive calculations being run in parallel, and no higher-level algorithms as would be needed in a full dive computer product design. In short, the original version of the GFM routine needed to be re-written in a form suited for a real-time PDC design. The code was re-written in C, and the new routine(s) were optimized to provide a more efficient and suitable starting point for design. It should also be noted that although the patent describes that GFM can be used in a multiple-tissue model, this study focuses only on a one-tissue model for simplicity.

To describe the C-code implementation of GFM, we start by looking at the code at a high level. There are certainly many different ways to implement real-time systems for embedded targets [7]-[9]. Depending on whether you have one processor or more, what type of processors are used, and depending on the specific resources for the application, the code can be greatly optimized for the application. In terms of C-code, at a high-level, specific details of the processor architecture are not so important since the compiler would typically take care of many needed optimizations and low-level detail. For simplicity and for the purposes of writing high-level C-code, we will consider a system that runs on one processor. Furthermore it is noted that in place of having real pressure input from a sensor, routines are needed that simulate pressure changes as inputs to the applicable GFM routines.

A Simulation-Only Implementation Structure

In the next section the program structure for a typical real-time implementation for an embedded target such as a battery-operated dive PDC will be described. In this section we are concerned with what the program looks like when we are concerned about accurate, representative and functional simulation, and not necessarily about the program being suitable for a real-time implementation. This is important because much needed work can be done without having to implement the algorithm as a real-time algorithm. For example, obtaining expected outputs for given input vectors can be found by simulation and the results do not depend on whether the code was run in real-time or not. This type of use is very important for application profiling, to be explained more in -depth in a later section, where extensive simulations are done on a wide range of input vectors to characterize the application program.

A simulation only implementation structure looks very similar to the GFM routine given in the GFM patent. This is not surprising since the purpose of the routine in the patent is similar to that described above, to be used for simulation-only purposes. On the other hand, some changes were made in the C-code version to facilitate moving toward a real-time implementation. In that sense, this version described here is very close to what a final-real-time implementation would look like, except that it does not run in real-time.

The high-level pseudo-code in Figure 4 shows a simple program structure used to run simulations. The main routine initializes variables, then runs various types of dives as required. What dives to run can come from a simple input vector file containing depths and times, or the dives can be hard-coded in special routines used only for testing. Note that to run an ascent or descent, some ascent/descent rate is required, and it is noted that a standard 60 ft per minute (fpm) will be used, although this can be changed anytime by inputting a new rate into the various functions or by setting a global variable holding this parameter. More complicated dives with varying ascent/descent rates can also be constructed similarly.

The *copyRealTimeState* function is essential to run prediction calculations for the following reason: If prediction calculations were done on the same set of variables as the real-time state calculations, the real-time state would change. This function copies all the required variables to another set, in order to perform prediction calculations without affecting the real-time state. Otherwise this simple set of functions is essentially all that is needed to simulate dives using GFM, including running predictions. The program actually used is slightly more complex, however, the pseudo-code representation below is representative of the functionality of the real code. For example, other higher level functions that use these low-level functions also exist, such as “findNODtime” and “predictSafetyStop”.

```
void main() {
   initialize(); // initialize variables and constants
   openLogFilesForOutput(filename, ...);
   runNODDive(depth, time);
   runMultiLevelDive(depth1, time1, depth2, time2, ...);
   runRepetDive(depth1, time1, depth2, time2, ...);
}

// setup constants, calculated values, variables
void initialize(arg1,arg2,...) {}

// step pressure and time to ascend to specified depth
void ascendTo(arg1,arg2,...) {}

// step pressure and time to descend to specified depth
void descendTo(arg1,arg2,...) {}

// wait for a specified amount of time at current depth
void waitFor(int time) {}

// run one GFM time-step (used by other routines)
void doTimeStep() {}

// used to run predictions, copy all state variables
// for use in predictions
void copyRealTimeState() {}
```

Figure 4: Pseudo-code for a simulation-only implementation of GFM

Note that instead of *hard-coding* specific dives to be run (as shown in the main function above), it could also just as easily be written such that dive parameters are read from a file and processed accordingly. What is important here is the overall structure and concept of the program.

A True Real-Time Implementation Structure

As will become clear later in this paper, this study will focus mostly on one portion of a GFM system – the base GFM routine and optimizations related to that routine. It is important, however, to understand what a practical application would look like at a higher level. This section will describe how a true real-time GFM-based dive computer product might be structured, in terms of the operating system and high-level code. Note that there are certainly many different ways to organize and structure real-time programs, and the following description is only one method. Because the target application is an ultra-low power battery-operated system, this description assumes that there is only one processor available.

The high-level pseudo-code in Figure 5 shows how a real-time GFM program running in a dive computer might be structured conceptually. Since the system is ultra-low-power and real-time, the program first initializes, then begins an infinite loop that ends with the processor going to a low-power sleep state. This is typical for a battery-operated low-power system such as a dive computer. The processor can wake up from interrupts such as timed interval interrupts, user input button (i/o) interrupts, or other interrupts setup in the particular system. Some of the functions that need to be performed when the system wakes up are shown to occur at different time intervals.

For example, for the system to be very responsive to user input, user input button interrupts might be checked at every 1/8 second and processed accordingly. Also occurring on regular intervals would be the task of keeping track of depth and time. Updating depth involves sampling an analog pressure sensor via an analog-to-digital converter, and updating time involves simply increasing the current value of the variables used to keep track of time. Being that a dive computer is a real-time system, it is assumed that there is a mechanism that allows for real-time to be tracked. As was mentioned previously, considering that the target application is a watch-style design, it is noted that a standard 32khz watch crystal is typically used in watch designs to keep track of real-time. There are of course other tasks that must be serviced for a real-time dive computer design, but the pseudo-code below includes some of the major components required. Note the GFM algorithm computation happening on the 6 second timescale described previously.

```
void main() {
   initialize(); // initialize variables and constants
   while(1) {
      // process events from ISRs
      if(time interval == 1/8 second) {
          // sample new depth, and keep track of time
          // process user input buttons
      }
      if(time interval == 1/2 second) {
          // update output display
      }
      if(time interval == 6 seconds) {
          // process GFM algorithm: run GFM timestep and predictions
          gfm_timestep();
          runPredictions();
      }
      goToSleep(); // wait for interrupts
   }
}

void timerInterrupt() {
   // flag interrupt for later processing
}
void userButtonInterrupt() {
   // log button press for later processing
}
```

Figure 5: Pseudo-code for a real-time implementation of GFM

It is very important to note that in the real-time pseudo-code in Figure 5, that the GFM time-step routine is meant to be a sub-routine. This is important because the we know that the predictions that need to be run involve many time-steps, and these time-steps may possibly take hundreds of milliseconds to process. The point is that if the required computation were simply in one continuous loop, as in the original VB-coded GFM routine, it may not be able to be run within a strict real-time system with time allotments such as above. In the description of the system above, tasks are scheduled into 1/8 second time-slots, such that any required computation occurring in one time-slot must complete within 1/8 second, or else the system is unstable. With each time-step run as a sub-routine call, this system is easily accommodated. The programmer only has to make sure that if each time-step runs in X ms of time, that no more than (125ms / Xms) time-steps are run in any given time-slot.

For example, if prediction is needed to find the maximum free gas from an ascent to the surface from the current depth, the following represents the calculation needed to schedule the task and perform the prediction. We start by considering that at some depth D, it would require M = (D ft / (ascent rate in ft/min)) minutes of real-time to reach the surface. For M minutes, you need N = M * (10 time-steps per minute) time-step calculations to calculate the ascent to the surface. So if D=60 ft and the ascent rate is 30 ft/min, it takes M=60/30=2 minutes to reach the surface, and it requires N=2*10=20 time-steps to be calculated. Since we want the maximum FGV on the surface after the ascent, we need a maximum of around 15 minutes of calculation on the surface for the free gas to peak, so that makes for another 150 time-steps. In total, to find the peak free gas for an ascent from 60ft at a rate of 30ft/min including 15 minutes of surface time, it would require 170 calls to the GFM time-step routine. Now suppose each time-step routine takes 1ms to process. Even if the routine could be processed that quickly, it would still require more time that is available in each time-slot. The 170 time-steps could however be broken up as a task, where say 50 time-steps are run inside each time-slot. At that rate it would require four 125ms timeslots. As long as the prediction is completed before the next real-time state calculation (every 6 seconds), the system will be stable at least in terms of time-step scheduling and given the time allotments as described here. This type of calculation above will be used frequently, and so the relationship between the amount of real dive time simulated versus the number of time-steps needed is expressed in equation form in Figure 6.

Let

N_{ts} = # time-steps required for prediction

DT = minutes of real dive time being simulated

TSPM = 10 time-steps / minute

CPTS = # cycles per time-step

T_{ts} = Time-step processing time in seconds

T_{pred} = Prediction processing time in seconds

Then,

$$N_{ts} = DT * TSPM \qquad \text{Equation 4}$$

$$T_{ts} = CPTS / freq_{system\_clock} \qquad \text{Equation 5}$$

$$T_{pred} = N_{ts} * T_{ts} \qquad \text{Equation 6}$$

$$= (DT * TSPM) * CPTS / freq_{system\_clock} \qquad \text{Equation 7}$$

Figure 6: Time-step and prediction relation with real time

A Set of Visual Aids for GFM

The concepts used with GFM are complex, and for the most part, the details of how the algorithm works and was derived is left for the reader to review in the algorithm patent document. On the other hand, the concepts are so central to the customized hardware research and design in this study. For that reason, this section will provide some visual-aids to help with some of th concepts. The reader is encouraged to refer back to this section for clarity while reading later sections. The format that follows is a set of images with explanatory captions.

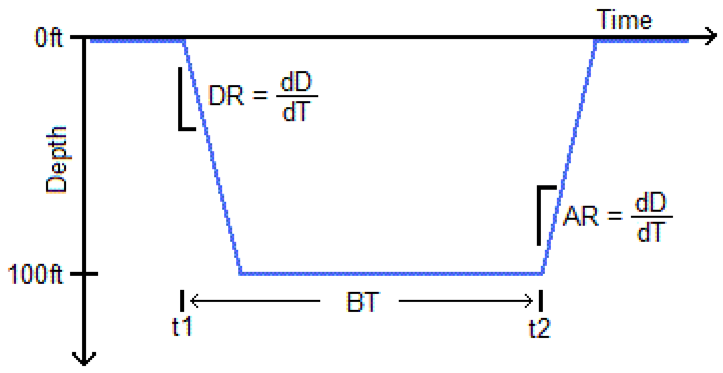


Figure 7: A dive profile for a simple dive: 100ft bottom depth, bottom time BT=t2-t1, ascent/descent rates (AR/DR) in fpm. Note that bottom time for a simple dive includes the time starting from the descent from 0ft (surface).

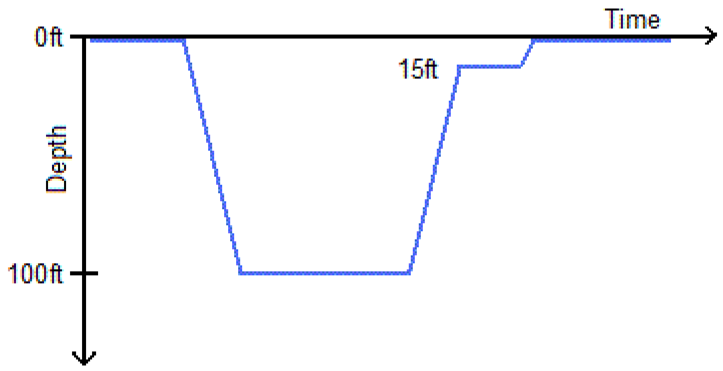


Figure 8: A dive profile for a simple dive with a safety stop. The safety stop is 3 minutes at 15ft.

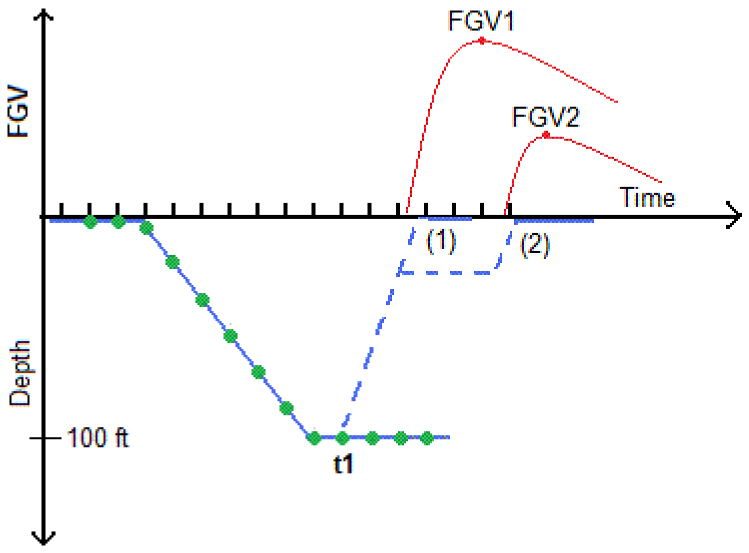


Figure 9: Free gas generation from two different future ascent scenarios (1) and (2) calculated at time t1

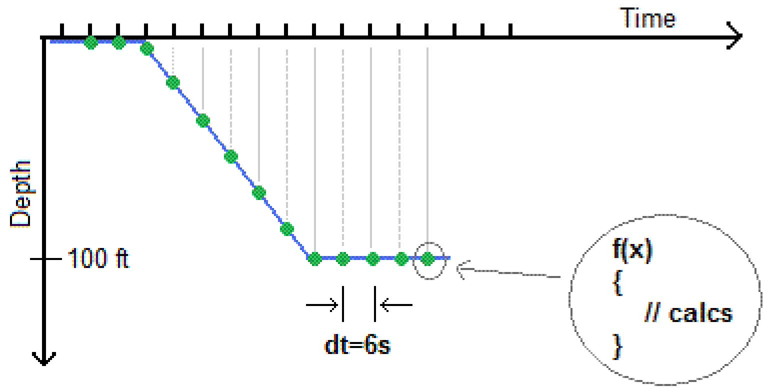


Figure 10: GFM time-step computation every 6s during dive

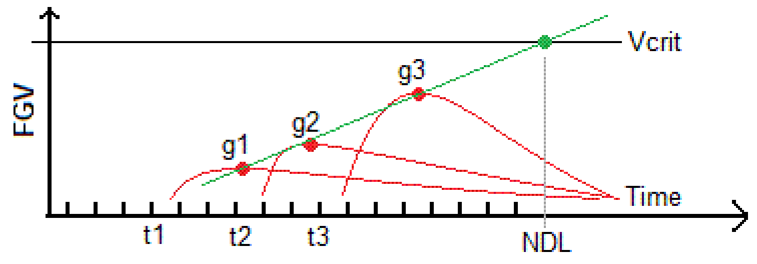


Figure 11: No decompression limit (NDL) calculation concept from points (t1,g1), ... ,(t3,g3). Note that the times t1, t2, t3 refer to the time of ascent associated with resulting free gas volumes g1,g2,g3, and the NDL is the ascent time when the resulting free gas volume exceeds Vcrit.

Standard Set of Dive Profiles

Like many real-time embedded applications, the GFM application takes some input(s) and produces some output(s). In the case of GFM, inputs are physiological constants, calculated but fixed values, as well as time and real-time sensor inputs. Outputs from GFM are data and control values - primarily the current and future (predicted) free gas volumes. To understand the algorithm and application, it is important to simulate outputs for given inputs. As will be clear in a later section dealing specifically with the importance of application profiling, a very robust input vector set is needed to properly characterize the application as part of the design flow for creating customized hardware. Besides the fixed inputs GFM requires, time and depth are the main inputs to GFM. At a higher level, what constitutes a robust, sufficient, and *standard*, set of dive profiles for algorithm characterization is important to define.

To define a set of dive profiles to be used for algorithm characterization is not a simple task. As was mentioned previously, there are many types of divers, and correspondingly there are many types of dives. There are, however, a few sources of information that can be used to help accomplish this. The first is the historical and experimental test data, such as the U.S. Navy / NOAA Dive Tables [10], U.S. Air Force technical reports, and other data used by dive science researchers. The data contained in NOAA / U.S. Navy dive tables, for example, have even been used as a basis for many modern algorithm designs. Secondly, information found from dive instruction institutions and retail dive equipment shops also provides more information.

For example, most dive equipment retailers that sell or rent high and low-end equipment will tell you that most of what they carry is related to recreational diving. Recreational diving refers to relatively shallow depths for relatively short times. What that means exactly is up for interpretation, but we can say that certainly recreational diving does not typically include so-called decompression diving, where divers stay under water after their NO-D time is exceeded. That means most diving is NO-D diving to shallow depths. Concerning air mixtures used, again, recreational diving typically refers to diving on compressed air, as opposed to other mixtures such as Nitrox (nitrogen and above 21% oxygen) or Heliox (includes helium). For Nitrox, mixtures are typically between 22% and 50% Oxygen, although diving is also done above 50% and up to 100%. Heliox is used only in smallest population of the diving community and will not be considered in this study.

A couple more dive types we need to mention here are repetitive diving, high-altitude diving, saturation diving, and multi-day diving. Repetitive diving refers to multiple dives being performed one after another, with some relatively short time in between. For example someone might dive to 40 ft for 20 minutes, surface for some time, then repeat the dive later in the day. This type of diving has its own consequences in terms of the body's reaction and in terms of safety limits. Similarly, high-altitude diving has its own differences from sea-level diving, and consequences in terms of the body's reaction to it. Multi-day diving refers to diving day after day for an extended period of time, and again, has its own consequences. Saturation diving refers to diving for extended periods of time, such that the body becomes saturated with Nitrogen. This is again a specialized type of diving performed by a relatively small population of divers. All of these dives are done in many parts of the world for different purposes, but not by large population of divers.

In a general sense, we need a robust set of input vectors that accurately represents all types of dives for which a GFM-computer would be used. For the purposes of application profiling (to be elaborated on in a later section), it would be useful to be able to run simulations from a script that includes possibly hundreds of dives, that represents this robust set of inputs we seek. The following lists detail dive profiles that are part of this standard set of dives used in various parts of this study. Note that all dives below also include the predictions run at each time-step during the dive, as described in the next section on performance constraints.

It is also noted that although we define the following sets of dives, in fact we do not need to run all sets each time profile information is needed. The reason is that the algorithm code for GFM is so simple, and contains so few branches, that pretty much every line of code is executed every time the routine runs, regardless of what type of dive is run. This is a unique feature of GFM, in that although it is a complex theory based on real physiological phenomena, its elegance is exhibited in its utterly simplistic formulation. As detailed in the patent, complex mathematics take the form of simple discrete formulas, easily implemented in software, and in fact almost exactly the same no matter what dive profile is input.

When a program runs different instructions corresponding to different inputs, it is said to exhibit program *phase* information [11][12]. Profiling techniques [35] become quite complex when programs run in different phases of operation, but for GFM this is not an issue. Definitely there is a difference between dives that produce free gas and those that do not, but as we will see in Chapter 5 the difference is totally negligible, and in fact robust profiling can be done a limited set of dives. In fact out of the dives presented below, only sets 1 through 4 are used for profiling, and results are inferred for all cases from those results.

- **Set 1: NO-D Dives on Air**: Dives to one bottom depth for a specified period of time, followed by ascent to the surface, then a surface time of 20 minutes. Depths starting from 40 ft to 180 ft, each for times of 1 minute up to the NO-D limit for each depth. To reduce the number of simulations but still keep a representative sample, we can step depth by 20 ft increments, and we will simulate out to half of NO-D time and to NO-D time.

- **Set 2: NO-D Dives on Air with Safety Stop**: Same as above but including a 3 minute stop at 15 ft before ascending to the surface.
- **Set 3: NO-D Dives on Nitrox:** Same as the above NO-D dives on Air but using Nitrox mixes of 25% , and 30%, oxygen.
- **Set 4: NO-D Dives on Nitrox with Safety Stop**: Same as above but including a 3 minute stop at 15 ft before ascending to the surface.
- **Set 5: Repetitive Dives on Air**: Dives to one first bottom depth out to NO-D time, followed by ascent to the surface, then a surface time of 2 hours, followed by another NO-D dive to the same depth. Depths range from 40 ft to 190 ft.
- **Set 6: Multi-Level Dives on Air**: Dives to one first bottom depth out to 5 minutes of NO-D time remaining, followed by an ascent to half the first depth, out to the NO-D time, then ascent to the surface for 30 minutes. First bottom depths range from 60 ft to 190 ft.
- **Set 7: Decompression Dives**: Decompression dives following the US Navy Standard Dive Tables ranging from bottom times of 40 ft to 130 ft and including dives with hundreds of minutes of decompression time.

Design Constraints for a GFM Dive Computer

To summarize the information on dive computers and the GFM algorithm, and to move forward in later sections, we need to detail exactly what the requirements are for the system. Here we detail constraints on power, performance, and area footprint for hardware running GFM inside a dive computer. The proposed implementation in Chapter 5 will be based on this and all information leading up that chapter.

Performance Constraints

To begin with, we can specify performance constraints. By this we mean that we will specify here exactly how much throughput is expected per some unit of time. As we saw previously, the computation needed in a GFM system is made up of two main parts. There are (1) the real-time state calculations, and (2) the prediction calculations. The real-time calculations involve calls to the base GFM time-step routine. The prediction calculations can be broken down into routines needed to simulate depth changes, and calls to the GFM time-step routine. For the real-time state, we know that the routine only needs to be called once every 6 seconds with an updated value for time and depth. As we will see in Chapter 5, this is a relatively easy constraint to meet, but it must be met nevertheless. Specifying performance for prediction calculations is more involved, and meeting them makes up the most challenging part of the design. Note that in constraint 1, time refers to computation processing time, and depends of course on details of the hardware used, including processor frequency, and cycle count per time-step. For the purposes of setting constraints, we can simply relate the number of time-steps needed to compute within strict real-time bounds, and the details will be worked out later.

(Processing) time per time-step <= 6 sec Constraint 1

As was mentioned earlier, how many and what type of predictions are needed for GFM is not set in stone, and in fact are up to the product designer in terms of how much useful future information is needed for the specific implementation. That being said, it was already mentioned that this study will focus on the most demanding, high-end type dive computer application, for which the power, area, and performance demands are greatest. As will become clear in a later section, we are focusing only on no-decompression dives, and the results and design choices made will be shown to be equally applicable to other types of diving. For the purposes of this study, we will assume that during a no-decompression dive, we want to be able to calculate the NO-D time, as well as the result of taking a 3-minute safety stop at 15 ft.

Remember from Chapter 1 that to be able to calculate the NO-D time, we need a set of points (t1', Max1), (t2', Max2), … (tn', MaxN) that can be used to predict the time when the peak FGV will breach the critical FGV, Vcrit. This essentially amounts to gathering data points from ascent predictions run after each time-step, and interpolating or extrapolating for the *future* time when the FGV will breach Vcrit.

This assumes that an immediate ascent max FGV has hot yet breached Vcrit, and we are in fact still in a no-decompression dive.

<u>Predictions to be completed within 6 seconds</u>

1. An immediate ascent prediction starting from the current time and depth
2. An immediate ascent prediction after 1 minutes at depth
3. An immediate ascent prediction after 5 minutes at depth
4. An immediate ascent prediction after 10 minutes at depth
5. An immediate ascent prediction after 35 minutes at depth
6. An immediate ascent prediction after 60 minutes at depth
7. The result of a safety stop of 3 minutes at 15 ft

Given the list above, we can calculate the amount of processing that would be needed, considering only the time-step computation and neglecting the subsequent calculation of NO-D time. We will use Equation 4 to indicate how many time-steps need to be processed within the 6s boundary, in order to be able to calculate the NO-D time. To find the number of time-steps, we first need the total number of real-time dive minutes simulated from predictions 1 through 7 above. To be able to figure out how much time the ascents require, we need a depth. Since depth can be any value, we have to assume the worst case where the diver is at 400 ft. The value of 400 ft is chosen only because most dive computers do not even go beyond that limit, and so that depth would give the maximum ascent time for a diver. As mentioned previously, we will use 60 fpm as the ascent rate, and 15 minutes of surface time to wait for the free gas to peak. We have the following simulated dive times, calculated as: <time at depth> + <ascent time> + <surface time>

- For prediction 1: 0 min + (400 ft / 60 fpm) + 15 min
- For prediction 2: 1 min + (400 ft / 60 fpm) + 15 min
- For prediction 3: 5 min + (400 ft / 60 fpm) + 15 min
- For prediction 4: 10 min + (400 ft / 60 fpm) + 15 min
- For prediction 5: 35 min + (400 ft / 60 fpm) + 15 min
- For prediction 6: 60 min + (400 ft / 60 fpm) + 15 min
- For prediction 7: 0 min + (400 ft / 60 fpm) + 3 min + 15 min

The total amount of real dive time (found by adding up the simulated dive times) considering predictions 1 through 7 is roughly 261 minutes. Since we know there are 10 time-steps needed per 1 minute of simulated dive time, we know then that to calculate these 7 predictions will involve running 2610 time-steps. Note that since we still need to run the 1 time-step for the real-time state, corresponding to constraint 1, we will combine the time-steps needed for a total of 2611 to be completed within the same 6s and label this new constraint *Constraint 2a*.

time for 2611 time-steps <= 6 sec

time per time-step <= 2.297 ms Constraint 2a

We make one more note here that it is not sufficient to simply meet Constraint 2a since, that would mean the process just meets the functional requirements, and would need to be processing at full speed 100 percent of the time. For a low-power system, these computations need to be run faster and spread out using a duty cycle percentage, where some portion of the time the processor is active, and the other portion it is in a low-power state. For this reason, we will introduce Constraint 2b which implies a 50% duty cycle, meaning that we wish to meet Constraint 2a while only processing 50% of the time and in a low-power state the other 50% of the time, given some fixed time period (seconds, minutes, etc).

time per time-step (50% duty) <= 1.149 ms Constraint 2b

Note that we are not concerned at this point how these constraints will be met. As we will see, the fact is that whether or not this can be met is a function of the efficiency of the implementation algorithm, program code, and hardware. It is also a function of the chosen processor speed and any applicable real-time scheduling constraints (more later). Of course the processor speed relates to power, which has its own constraints as well. For now we simply state that in this real-time system, a GFM dive computer the base time-step routine must calculate within a strict bound of no more than 1.149ms. This would allow for the real-time algorithm state, as well the predictions outlined above to all be calculated between each time-step run every 6 seconds.

As we will see later this is a very aggressive goal considering available low-power hardware, and if met, would amount to a massive increase in performance. The trick is to make sure not only the performance goals, but also the power and area constraints are met as well. For now this performance goal above will be enough to continue with the design. Later we will see why we need to consider not only no-decompression dives, but also decompression dives. Looking ahead a bit, in fact this is another reason why constraint 2b was set so aggressively, so that the hardware will be fast enough to handle decompression calculations.

Power Constraints

Defining power constraints for a dive computer is rather involved. We know from Chapter 1 that the device is to be a battery-powered, watch-style design that uses a small 3V coin-cell battery. We will use a Sony CR2430 Lithium Maganese Dioxide type battery [13] as the example battery since this and similar types are used with various modern watch-style dive computer products. These type batteries can typically supply 25mA maximum short-duration peak current, with a recommended continuous current of around 200uA. They have a capacity of 300 mAh.

Now, to define a power constraint, we need to first define a set of use-cases. In other words, it is not sufficient to simply say that we wish to have a limit of X mA, or Y mAh, when that is not correlated with the use of the product. For example, if we said that the limit was 300mAh, what would be the reasoning? Would that tell us how long the user will be able to use the product? Of course it would not, because how long the product will last depends on how much it is used, and how much power is drawn from the battery when it is used. The point here is that we cannot use an arbitrary power limit, we need to use something that is practical, and at the same time, relates to the usage of the product.

We will define a power constraint such that it relates the amount of time the product can be used for under a normal and/or worst-case scenario. Many times companies will use a typical-use-case information to define the amount of time their product can be used for. For example, let's consider modern cellphones. As of 10/29/09, from the corporate website, Apple says that the Iphone 3G has a battery lifetime specified as “up to 5 hours” (talk time) on the new 3G network, or 277 hours standby time, where temperature, humidity, and altitude are also specified. That sounds very simple, but upon closer inspection, you will find a separate link to articles [14][15] about how battery life is calculated, and how it can be optimized. In addition to explaining how different applications and application settings affect battery life, one of the more interesting suggestions is that they say you should use your Iphone regularly to keep electrons moving inside the batteries.

What this all boils down to is that whatever the power constraints were that they used in designing the Iphone, they were very specific to some particular usage, and in fact Apple has dedicated multiple web pages to explain what happens when you deviate from standard usage. More appropriate to the discussion is the 2006 press release from Texas Instruments [16] explains how Pelagic's scuba dive computers can last over 2 years in “ordinary dive use” using TI's low-power MCUs. They make note of the fact that Pelagic uses the low-power modes of the MSP430 microprocessor when in watch mode on the watch-style Atom 2.0 dive computer product. What defines “ordinary dive use” is up for discussion, but we will arrive at a concrete definition for our purposes here.

For our GFM dive computer, we will define the following use-case, in order to arrive at a reasonable power constraint. Long dives require large tanks of some breathing mixture, and it is impractical for most divers to carry multiple tanks or re-fill them on the surface mid-dive. Because of that, we will say one, 30 minute, no-decompression dive per day covers typical usage for the general case. We will further stipulate that a typical dive is to one maximum depth, followed by an ascent to the surface at a rate of 60 fpm. We will say that if a diver makes one of these dives per day, the dive computer should last at least one year without changing the battery. The power constraint then, is two-part, given as constraints 3 and 4:

The instantaneous (short-duration) power and continuous (long duration) power draw from the battery must be such that a Sony CR2430 (or equivalent) type battery can be used to power the device. Constraint 3

The device must last a minimum of 1 year, supposing the unit is used in active (dive) mode for up to 30 minutes per day, following a typical dive profile as described in the preceding paragraph. Constraint 4

<u>Area Footprint Constraints</u>

Defining an area footprint for the processor hardware used for GFM means that we wish to put a boundary on how large the hardware is. We note again that the goal is to create the most efficient low-power hardware design that can fit inside a watch-style dive computer product. This is not something that can be easily quantified, but the for the purposes of this study we can again use the Pelagic Atom 2.0 product as a reference. The Atom 2.0 is a low-profile watch-style dive computer product that is fully featured and was at the time of its release, a top-of-the-line product. If you take apart an Atom 2.0, you will see the the printed-circuit-board (PCB) is no larger than a 2.25 square inches. Of course the board has many other components like EEPROM, and circuitry related to user input, and display output, so we need to consider that the processor size will be considerably less than the total board size. We can consider that the x4xx series microprocessors in the TI MSP430 family processor hardware is less than 2 mm in height, and about 12mm square (without pins). We will use Constraint 5 below for our processor hardware based on the information above.

Processor hardware < 12 sq. mm, < 2 mm (height) Constraint 5

Chapter 2: Low-Energy Hardware Design

As will begin to become clear, the goal of this study is to investigate energy-efficient design options for the particular application of a dive computer utilizing the GFM algorithm. For this reason, we need to begin by understanding energy as it pertains to CMOS circuits. Where does the energy used in a processor come from? Are there different types of energy? Do different components use energy differently or at different times? These are all valid questions and important to understand when discussing energy-efficient hardware. Customizing hardware for energy efficiency involves understanding where options exist, and where customization can in fact be made. The goal is to be able to use a fundamental knowledge of energy to inform design decisions from the beginning of the design process.

Sources Of Energy in CMOS Circuits

Whether it be a general-purpose processor (GPP), or an application-specific integrated circuit (ASIC), based on the application requirements we will now assume that some sort of processor needs to be involved in implementing the GFM algorithm most efficiently. More implementation options will be discussed in chapter 3, but for now we will assume that whatever the chosen implementation in terms of system architecture, the hardware will involve modern complementary metal-oxide semiconductor (CMOS) circuit technology used in modern processor design. In this case, we need to understand where the energy comes from in the circuits to understand how to design for an efficient and high-performance system.

Digital CMOS circuits are made of course of CMOS transistors [17]. CMOS circuits dissipate energy simply by being powered up, and by switching. We can describe the power consumption in a CMOS circuit as given in Equation 8.

$$P_{total} = P_{static} + P_{dynamic} \qquad \text{Equation 8}$$

Static power refers to power that is drawn simply by the transistors being present in the circuit. Modern designs try to minimize static power as much as possible, but it cannot be completely be removed unless the transistors are in fact not connected to the circuit. Static power can be broken down further into standby power and leakage power. Standby power is the product of standby current and supply voltage Vdd, and can usually be neglected in modern designs. It does, however, play a role in memory circuits and in some circuit technologies. Leakage power is the product of leakage current and Vdd, and is a function of how modern MOS transistors are designed. As long as power is applied to the circuit, leakage current is a factor in total power consumption. Note that leakage power is increasingly becoming an issue as transistor sizes decrease, and there has been a large amount of research recently dedicated to finding solutions to this issue as we move into the future [18][19]. Equation 9 shows the static power broken down as described above.

$$P_{total} = [P_{standby} + P_{leakage}] + P_{dynamic} \qquad \text{Equation 9}$$

The main source of power consumption, especially in a processor-based design, is the *dynamic* power. Dynamic power refers generally to the switching of the CMOS transistors as the logic circuit operate. Each time the transistors switch there are two types of power that are dissipated, short-circuit power and capacitive power. Short-circuit power and current refers to how when a CMOS transistor switches, one transistor transitions from an *on* to an *off* state, and the other transistor from an *off* to an *on* state. This results in a a small period of time where both transistors are on, and the current is shorted from Vdd to Vss (where Vss is the ground or virtual ground terminal of the transistor). It is noted in [17] that because short-circuit current is a result of switching, it can be considered as an additional capacitance and so dynamic power can be simplified as in equation 11, showing proportionality to switching probability α, total capacitance C_{tot}, V_{dd}^2, and the clock frequency f_{clk}.

$$P_{total} = [P_{standby} + P_{leakage}] + [P_{short\_circuit} + P_{capacitive}] \qquad \text{Equation 10}$$

$$P_{dynamic} = \alpha * C_{tot} * V_{dd}^2 * f_{clk} \qquad \text{Equation 11}$$

Capacitive power comes from average switching probability, the clock frequency, and the switched capacitance in the circuit. It is proportional to $\alpha * C * V_{dd}^2 * f_{clk}$. It is this capacitive power that we are most concerned with in processor design, for obvious reasons. First, it makes the most significant contribution to the overall power profile of the circuit. Modern processes minimize static power as much as possible, but dynamic power is still present when transistors switch. From Equation 11 we can see that reducing any of the factors would reduce the dynamic power, and there exist many different techniques to accomplish this, to be explained in the next section.

Techniques to Reduce Dynamic Energy Consumption

Dynamic energy in CMOS circuits was shown to be proportional to $\alpha*C*V_{dd}^2*f_{clk}$. This means to reduce dynamic energy, we can lower the amount of switching, the capacitance of the circuit, the power supply voltage, or the clock frequency. This seems simple enough, but each choice has its own consequences. Lowering the supply voltage, for example, has a dramatic effect, since dynamic power is proportional to the square of Vdd. However, one trade-off is that this can increase the propagation delay of combinational logic circuits. The voltage cannot be scaled indefinitely either, and there are limits. As supply voltages reduce, leakage power becomes more of an issue, and so again there are trade-offs. Designs can employ complex structures where critical regions are designed such that have short propagation paths, and where uncritical regions are otherwise not optimized. These designs can strike a balance where power is reduced, and issues related to low supply voltages are minimized. In [25], the theoretical and practical limits of voltage scaling are discussed, and in [26], a solution to one particular related issue of degraded performance due to voltage scaling is discussed.

Lowering the system clock frequency is another simple way to reduce power consumption, but has the obvious effect of reducing performance at the same time. Performance for a processor is defined in different ways by different manufacturers. Performance metrics are described differently in terms of what type of processor is being measured. For example, some manufactures of reduced instruction set (RISC) and complex instruction-set (CISC) processors use the metric of *millions of instructions per second*, known as MIPS. How many MIPS a processor is capable of is some measure of the architectural efficiency as well as performance, since, one processor using all the same parameters (technology, supply voltage, etc) as another, can have a dramatically higher MIPS than the other. While lowering the clock frequency lowers the power and correspondingly the performance, some performance can be increased by increasing Instruction-Level-Parallelism (ILP). Architecting more ILP in a processor is one way it can process more instructions per second given a fixed frequency, and there are many techniques that exist [17][20][21] for doing this such as Pipelining and using more complex architectures. More on architecture choices will be discussed in the next chapter.

Lowering the capacitance in a processor can also decrease the power consumption. Capacitance comes in the form of capacitance found at each node, or each transistor, and capacitance that exists with the interconnect structure used between different components in the design. Reducing capacitance means reducing delays and thereby increasing performance. Architectural choices, as well as technological choices affect the inherent capacitance in a design.

If we look at different levels of abstraction in the design of custom hardware, we can see which types of techniques affect power consumption and in what proportion. This is depicted in Figure 12. The figure shows how at the highest level,

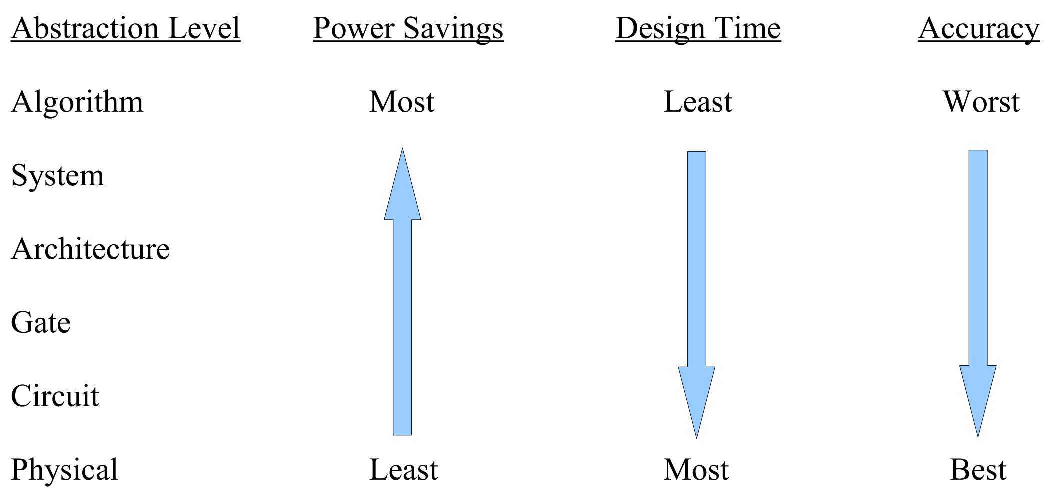


Figure 12: Abstraction Level vs. Power Savings

while algorithm design choices offer the most amount of power savings and take the least amount of time, they can also be the least accurate in determining how much power would be saved. Following a trend of lesser power savings, but increasing design time and accuracy, system level, architectural level, and gate and circuit level choices follow. At the other extreme, physical level design (design at the transistor and silicon level), you get the least amount of power saving, with the most amount of design time needed, but with the best accuracy. This is obvious since if you design each transistor you know exactly where each component of power is coming from, and you have the most fine-grained control over power consumption. At the algorithm level, you need virtually no knowledge of the underlying technology to make decisions, but you can make quick changes that result in huge power savings by optimizing the algorithm and reducing unneeded operations.

This study focuses mostly on the upper four levels of abstraction from Figure 12, the algorithm level, system level, the architectural level, and the gate level. As will become clear in Chapter 4, an ASIP implementation will be recommended for a GFM-based dive computer. The work done in this study follows the same order from the highest level of abstraction performing algorithmic optimizations, down to the gate level where choices are made for a design written in a Hardware Description Language (HDL).

At the algorithmic level, included in this study is a re-write and re-design of the original algorithm. Operations are reduced, and more efficiency is achieved by reducing memory access and high-level operations in a way that makes for possibly the largest memory consumption reduction. Reducing operations, re-ordering and re-scheduling operations, reducing memory access, and reducing memory size are different techniques used to optimize at the algorithm level. As was explained previously, it is difficult to determine how much power is reduced at the algorithm level since it depends so much on the underlying levels of abstractions, but in a general sense we can say that if we reduce operations by 50%, roughly 50% of the power is also reduced, assuming dynamic power in processing operations is the most significant contributor. Besides analyzing the high-level algorithm code and reducing obvious inefficiencies, the most significant technique used in this study is in the conversion from a floating-point algorithm to a fixed-point one. For real-time embedded systems, this is widely considered [20]-[23] to be an efficient way to reduce operations across the board in a computation intensive floating-point algorithm. How this conversion was done is detailed in Chapter 3.

At the system level, the entire dive computer system is analyzed and optimized. This is detailed in Chapter 5 in the sections on application profiling and on hardware/software partitioning. This involves first profiling the application in detail, to determine which are critical in terms of power and performance, and which are not. It is then determined which portions of the algorithm are to be implemented in hardware, which in software, and further which portions of hardware need to be customized or not. Trade-offs between flexibility, performance, area footprint, and efficiency are part of the ASIP design flow, and are central to reducing power at multiple levels of abstraction, including the system, architecture, and gate levels.

At the architecture level is where the ISA is designed/selected and the processor core design/selection occurs. The processor memory system, datapath and control circuits are all designed in order to best fit the application domain, and provide maximum efficiency given the application constraints. Included also is the design/selection of low-level processor firmware, and required toolset as part of the hardware/software co-design.

At the gate level is where the actual design in HDL comes into play. Choices made in terms of clock domains used, clock gating design and other techniques are used to trade-off performance and area and power. Much of this work can be aided by the sophisticated synthesis tools used in processor design, but even a the HDL level there is some level of control the designer has to inform the tools what the design goal is. For example, the Xilinx FPGA design tools used in this study contain complex customization controls to allow the designer to specify how gate-level circuits are to be constructed. One such control allows the designer to specify whether to extract intended multiplexers (MUX) from the design. This allows for the designer to choose whether to use the specific structures coded in HDL, or to possibly optimize those structures by using MUXs in place of possibly more complicated logic. There are many such controls, and the user can set up a design goal strategy such as power reduction, area reduction, or high performance priority, or use a totally custom strategy where each option is specified explicitly.

Chapter 3: Implementation Options and Trade-Offs

In Chapter 1, the GFM application and its related application were discussed in depth with examples. Input vectors for simulation-only and real-time high-level C-code implementations of GFM were described and detailed. Lastly, the constraints on performance, power, and area were defined for an efficient and practical processor handling GFM calculations. In Chapter 2, the source of energy in CMOS circuits as well as low-power design issues and techniques were discussed. At this point we have a good idea of what the GFM algorithm and related application requires, and we wish to detail what options are available for an efficient implementation.

In terms of processors, we have a spectrum of hardware types, each with different characteristics and target applications. If we imagine this spectrum as a one-dimensional line, we can say that on one end we have totally custom hardware and Application-Specific Integrated Circuits (ASICs), and on the other end we have General Purpose Processors (GPPs). In between we have a multitude of different devices, including Application-Specific Instruction Processors (ASIPs), Digital Signal Processors (DSPs), and hybrid architectures such as GPPs with accelerators and most recently, GPPs with embedded programmable logic. The hardware options listed above each have their own trade-offs in terms of performance, area and power, as well as their related design flows. We will discuss these options in this chapter, including why each may or may not be a good fit for a GFM-based dive computer.

The illustration in Figure 8 taken from [17] shows this spectrum of hardware options where the relative flexibility, computational performance, and energy efficiency is compared among the different types. Note how from left to right the flexibility increases, and from right to left both the performance and the efficiency decreases. This is a generally applicable illustration, and there are always exceptions. It is important to note that in recent years many new devices are being made that cross the boundaries between the different types, and can be considered hybrids made of one or more of the different types. Recent examples include general-purpose processors that include pre-defined hardware accelerators. One thing not illustrated in the figure is the time and cost of development, and we can say generally that costs increase from right to left, as does development time.

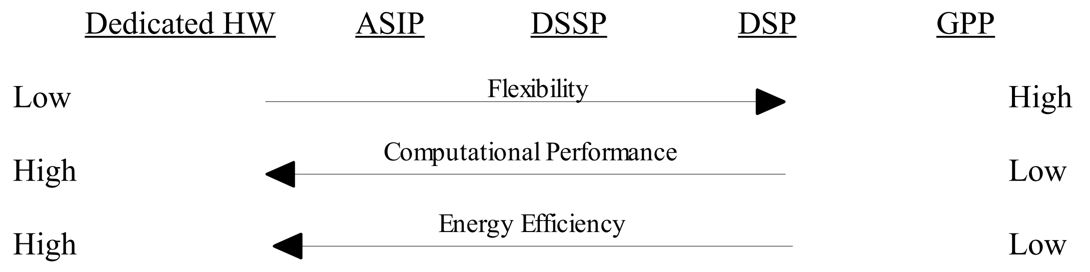


Figure 13: Processor HW Implementation Options

Custom Hardware Implementations and ASICs

We begin here with totally custom hardware and ASIC solutions. With totally customized hardware, you can typically achieve the highest performance. By this we mean that a circuit is constructed to do exactly one thing, as efficiently as possible, and there is literally no overhead in doing other operations or managing other components within a larger circuit. As a simplest case example, suppose we need hardware that inverts one input bit, and adds it to another input bit. The hardware needs two, one-bit inputs A and B, and produces one bit of output X which can be described by the logic function $X = \overline{A} + B$.

If we first consider the most gratuitous use of hardware as an extreme example of what not to do, we consider that we implement this inverter function using a general-purpose processor. Being a processor, suppose its hardware has the capability of processing a variety of instructions, accessing a variety of different memory elements and structures, as well as accessing input output (i/o) ports and other internal substructures and components. If we wish to implement our function X in the GPP, we will first need to program it. The program would involve reading the two input bits (either from memory or from i/o ports), negating A, then adding it to B. If we suppose a RISC-type instruction set, we can suppose that reading the two inputs, negating A, adding $\overline{A}$ to B, and finally outputting the result to some memory bit or i/o port, would require roughly 5 or 6 operations. If we consider the time and energy required in the GPP to fetch the instructions and process them, it should be clear why that solution to the very simple problem involves way more hardware than is needed. The hardware will be relatively slow for the operation performed, require way too much power (mostly in overhead), and will take up way too much space than is appropriate.

The simplest solution might be to use a combination of simple logic gates to perform the function desired. For example, if we connect the A input to an inverter, and the B input to an exclusive-OR (XOR) gate, we can connect the output of the inverter to the other XOR input, and the output of the XOR gate will be the function X. This solution involves only a few transistors contributing to power, will complete in drastically less time than would the first solution, and will take up magnitudes less space.

The idea here is that at least in the case of a very simple operation needed, custom hardware can dramatically increase performance, and reduce area and power. Since we are interested in running a complex algorithm with real-time bounds, this example is a bit lacking, but we can extrapolate the idea to a more appropriate discussion. For the GFM algorithm, we know that thousands of operations need to complete in the shortest time, at the lowest power, and using the least amount of space as possible. Although we will get more into profiling results of GFM in Chapter 5, we can note here that on a the SimpleScalar RISC-based simulator [27] one GFM time-step (in a single-precision floating-point implementation) requires over 5000 different operations. Since we want to perform 2611 time-steps every 6 seconds, that means we need to complete over 13 million operations every 6 seconds.

Now, performing these operations in a digital circuit is not the issue, rather, the issue is performing them efficiently. If we consider making one totally custom hardware implementation not involving some sort of processor, you can imagine that this circuit would be extremely large in area. That fact alone would probably disqualify it as an option, but even if not, it suffers from other problems. The circuit would be very complicated to design, and would most likely be very inefficient. If we are considering only combinational logic circuits without any sort of instructions or even clocking, we can certainly say that a custom hardware circuit implementing over 13 million operations would probably be the worst design option.

In between might be something having the processing capabilities like a GPP, but with more efficiency like a custom combinatorial hardware circuit. This first option we want to discuss is the ASIC. ASIC designs take into account that many times the algorithm or operation that needs to be performed is complex in nature, but requires a degree of efficiency. For example, many papers exist that discuss custom hardware solutions for encryption algorithms [28], and in virtually all cellphones you will find ASICs made for display processing, audio or video decoding, and radio-frequency (RF) signal processing.

An ASIC is generally an IC that may be configurable but not programmable in the same sense as a GPP. The hardware is fixed for one application, and so in that way it has no flexibility. It is typically optimized for performance and power for the one target application only. Its performance is typically the highest of any other implementation, and it is generally the most energy efficient as well. For GFM, due to its complex nature, large number of operations required to be performed during its most basic operation (the time-step), and the fact that many of the operation types are repeated over and over within the time-step, we can say that an ASIC implementation is not ideal or even desired, and that some sort of processor would be more ideal for GFM.

Application Specific Instruction Processors (ASIPs)

As ASIP is closest to the ASIC end of the spectrum, but with an ASIP we introduce processing instructions. As the name suggests, an ASIP is an instruction processor designed specifically for one application. The idea is that an Instruction Set Architecture (ISA) and Processor Architecture (PA) are both optimized and custom designed for the target application. In this way, the hardware can be extremely efficient and higher performance as compared to a general purpose processor, while offering a bit more flexibility than an ASIC. Application profiling is very important in designing an ASIP, since the goal is to perform the operations of a specific algorithm efficiently.

To understand more fully how an ASIP can provide more efficiency than the GPP end of the spectrum, consider an application that requires programmability and a processor core as the base hardware. If we start with a RISC-type processor like

MIPS [29], there might be 20 or more instructions available. Consider that a certain application really only makes use 5 of those instructions. In that case, the overhead of being able to fetch, decode, and process all the unused instructions is waste, and could be eliminated.

By profiling the application we can determine which instructions are used, and used most frequently. Suppose further that through profiling we find that there is a set of two operations that always execute in pairs, and very frequently. For example, suppose adding two 8-bit integers is always followed by multiplying two different numbers. These two operations could be combined into one instruction, and a Custom Functional Unit (CFU) could be designed to perform the operation in parallel instead of in series. Custom instructions and CFUs are common characteristics of an ASIP design, although there are many more techniques that exist [30][31]. TMS-320C2x processors made by Texas Instruments include special addressing modes designed for FFT applications. This is an example where a DSP (explained in the next section) was designed to include application specific features. In this way the DSP chip is more like an ASIP since the custom addressing mode is not generally useful, it is useful only for a specific application. There are countless studies on all parts of the design space for ASIPs, including those on instruction set selection for ASIPs [32]-[34], ASIP designs for low power , ASIPs for specific applications, and automatic ASIP generation techniques.

It should be noted that a considerable amount of work has been done over the last decade concerning all parts of the design space for application specific instruction processors. As described in [17], an application-specific instruction processor yields a balance between the optimal design of a wholly hardware specific implementation and a general-purpose processor. Typically the design processes involves multiple phases. The first involves obtaining detailed profile information [17][20][21][35] on the optimized algorithm. The application profile information is used to inform design decisions, and identify where enhancements can be made for the application. Given that profile, an existing architecture can be customized to more efficiently meet the specific workload of the application domain. Instruction-set extensions (ISEs) and CFUs can further increase the performance, and work on hardware-software co-design [17][20][21] addresses the issue of producing new software tools for new instruction sets and otherwise modified instruction set architectures. Application-specific functional units (AFUs) can be added to implement ISEs.

Work has also been done on application-specific registers and register structures, and application-specific floating-point units (FPUs) for custom architectures. In addition the pipeline, datapath and control components are also a necessary part of ASIP design. If an existing core is being used, much of this work is already done, although typically modifications are made. The storage and memory networks can also be customized for a specific application, and again profile information is used to direct design decisions. Work has been done on automatically generating the various topologies and configurations mentioned above. In particular, automatic instruction set extension (ISE) identification and generation programs have been researched and have been identified as being a crucial part in the early stage of the pre-architecture design phase. Tools even exist that can go directly from application source code to automatically generate an ASIP HDL implementation, although these methods rely somewhat on pre-defined component libraries, and other fixed parameters. Finally, low-power VLSI and HDL design techniques [17][36] are used typically in order to reduce the system frequency, capacitance and voltage requirements. It is also noted that typically the ASIP design process is iterative, in that design changes often require another pass at earlier stages in the design before finding the optimal configuration.

It should be noted that the term ASIP does not indicate any specific processor architecture. In fact, an ASIP can be designed to use existing ISAs such as RISC, CISC, or VLIW, or it can be designed with a totally new architecture. ASIP simply refers to the fact that

Domain Specific and Digital Signal Processors (DSSPs, DSPs)

Domain Specific Signal Processors and Digital Signal Processors are another class of processor hardware more toward the GPP end of the spectrum. These processors are not made specifically for one application, rather, they are optimized for a certain domain of applications. These processors are typically programmable, and contain specialized instructions specifically useful for the application domain.

Examples include the C5000 family from Texas Instruments and the Sharc family from Analog Devices.

General Purpose Processors (GPPs)

General purpose processors are the most widely used, and most generally applicable processors available. GPPs are designed for a variety of applications, and not optimized for any one specific application, or even application domain. They offer the highest flexibility, since they can be programmed to do almost any task, given that the specific device has enough memory and processing capability for the specific task. GPPs also exhibit the quickest development time and lowest cost. There is generally no hardware customization needed, and because they sell in high volumes, they are relatively inexpensive. From a practical point of view, they also benefit from having very well documented uses. There are literally millions of on-line resources providing example code, application notes, example system designs, and other useful information.

Examples of GPP devices include the TI MSP430, Atmel's AVR, Microchip's PIC, NXP's ARM-based microprocessors, Freescale's HC11, and the famous Intel 4004, 8080, and 8086 processors. Still more exist from Fujitsu, Samsung, Epson, STMicroelectronics, Renesas, NEC, and others. Processors from these companies are in everything from refrigerators to space shuttles, and number in the trillions sold worldwide. They generally come in 8-bit, 16-bit, 32-bit, and 64-bit configurations, and come with a variety of peripheral modules. Peripherals typically include communication interfaces like UART, SPI, and I^{2}C, timing modules, analog-to-digital (A/D) and digital-to-analog (D/A) interfaces, general-purpose i/o ports, and more.

Using a GPP for a GFM-based dive computer is not a bad idea, provided that there exist microprocessors that can satisfy all the constraints of the system. A study conducted by this paper's author suggests that there are not many, if any, microprocessors that can process the large amount of information GFM requires, within the real-time bounds required, and at a sufficiently low-power as to satisfy the power and area constraints for a dive computer. Even if one exists that can just satisfy the requirements, still more benefit can be achieved by using a more customized hardware that offers higher performance and more energy efficiency.

The time-saving aspect of developing with GPPs cannot be emphasized enough, and how the advent of high-level computer languages has played a big role. High-level languages like the C Programming Language allow programmers to develop algorithms and code modules that, if designed to be portable, can be used on a variety of microprocessor with little re-design or effort porting the code. Since high-level languages can abstract the details of hardware, the code can be used on virtually any GPP as long as the low-level hardware-specific routines are ported. An I^{2}C routine is a good example of this. I^{2}C is a communication interface that uses two lines, one for data, and for a clock. The interface uses addressing to select different IC's on the bus, and communication is bi-directional. A high-level I^{2}C routine could be written entirely without any details on the specific hardware being used, and the user would need only to specify which i/o ports are used for the two lines, and to configure the low-level routines to access those ports. I^2C is only one example and in fact there are thousands of portable routines available on-line.

Combining the ease-of-use of portable high-level routines, and the general-purpose nature of GPPs, designs can be completed in relatively little time or cost, although while sacrificing efficiency. For many applications efficiency is not an issue, and a GPP is a great fit. For example, if you were to design a vending machine program that processed button inputs from the user, and then sent a signal to output the product purchased, energy efficiency or real-time performance may not be a high priority. If the user gets their product a few seconds later than they expected, for example, it will most likely not be an issue. We can also say that whether a vending machine processor uses 1mA or 2mA while processing user input may not be a high priority. In recent years, energy efficiency for all electronics has become a general priority, but we can say the importance is much higher in a battery-operated safety-critical medical device, for example.

Processor Architectures

A lot has been said in this chapter about different architectures at a high-level, but there is one thing that has been only mentioned in passing. This is the idea that we need to use some sort of processor architecture for the GFM algorithm. If we are going to use a processor to implement GFM in the low-power dive computer application, we need to understand a bit about processor architectures. Whether we use an ASIP, DSP, or GPP, we need to have an understanding about different types of processor topologies, and why they are used. In this section we describe some of the most commonly used processor architectures and concepts.

Processor Architectures - RISC and Pipelining

We can begin with what is called a Reduced Instruction Set Computer (RISC). A RISC architecture is one that, as the name suggests, is based on a simple instruction set. The concept goes all the way back to the 1970's and is still used today. The idea is to use a very simple instruction set including simple arithmetic and logical operations that operated on registers, and to use special load and store instructions to operate on memory. Since memory accesses are slow compared to register accesses, this allows most operations to be performed on registers rather than memory. Using very simple, 1-cycle instructions allows the processor to run at higher frequencies, and make it easier to design *pipelines*, explained in the next paragraph. Examples of RISC assembly instructions include: load, store, add, subtract, multiple, divide, rotate bits left, rotate bits right, and many more.

A pipelined processor is one that runs instructions in parallel (as much as possible) rather than in series, making use of hardware components that would otherwise be unused. Pipelining is a technique to achieve high Instruction Level Parallelism (ILP), which can increase performance of an application by a factor greater than or equal to the number of *stages* in the pipeline [29]. An analogy that is very intuitive to understand is that of washing clothes. If you break up the task is stages, you can say that first you need to wash, then dry, then fold the clothes. While you are drying, you can put another load in the washer, and while you are folding, you can put the wet clothes in the dryer. If each operation takes 10 minutes to complete, and you perform them in series, it would take 3000 minutes for 100 loads, for example. On the other if you pipeline the tasks, as described above, then you will complete 100 loads in roughly 1000 minutes, since the pipeline effectively allows you to wash, dry, and fold each load in an average of 10 minutes. The serial washing time of 3000 minutes is roughly divided by 3 (the number of stages in the pipeline), for a pipelined total of about 1000 minutes.

Unfortunately in practice, although pipelining makes operations much more efficient, there a whole host of issues that need to be handled in real designs. In real designs, there can be dependencies on one stage to the next, and so some operations cannot be done in parallel, unless some sort of modification is made. These issues are called *hazards* [29], and related hardware is called *hazard detection* and *elimination* hardware. There are various types of hazards including *data hazards*, *control hazards*, and *structural hazards*, and each has own consequences and solutions.
A good pipeline design takes into account all the hazards that need to be accounted for. The pipelined MIPS type processors are a good example of pipeline design, and are featured in various computer architecture textbooks. A modern example of a

popular pipelined RISC type processor is the ARM7 ARM9, and ARM11 family of processors, used in everything from the latest hand-held gaming systems to the latest touch-screen smart-phones. RISC designs emphasize simplicity in instruction set and hardware, many registers, few addressing modes, and high performance.

Processor Architectures - CISC

CISC architectures refer to a Complex Instruction Set Computer (CISC), where the instructions handle multiple operations, as opposed to one simple operation. Because each instruction performs more operations, they generally contain more information, and are more complex to implement than simpler RISC instructions. As example might be for one instruction to perform an addition, as well as a logical bit shift operation. At the time before advanced compiler design, it was thought that hardware was easier to design than compilers, so CISC instructions were meant to perform as many operations as possible in fewer instructions. It was also expensive to have large program code space in processors, however today it is not as much of a problem. Later, CISC was adapted with the goal of making it easier to adapt high level languages like C, and to make hardware analogs to the high level constructs like procedure calls, and loop control. The Motorola 68K series microprocessors are an example of CISC architecture. CISC architectures emphasize small code space and speed, extensive addressing modes, few registers, simple compiler design, with more emphasis on hardware.

<u>Processor Architectures – VLIW</u>

VLIW refers to computers designed with Very Long Instruction Word (VLIW) architectures. This refers to the idea of creating very long instructions with multiple operations per instruction, to achieve higher ILP. This relies on the underling architecture having multiple functional units available for use, and on efficient *multiple-issue static scheduling* of instructions at compile time. Compilers for VLIW architectures are generally very complex, while the hardware is relatively simple.

For example, suppose a processor core has 4 arithmetic and logic units (ALUs), and 1 memory. A VLIW instruction might include 1 add, 1 subtract, 1 bit shift, 1 bit negation and one load operation, all in one instruction. By scheduling operations effectively, a VLIW compiler can produce code that makes fullest advantage of the VLIW hardware for a high level of ILP and efficiency. As with pipeline designs, the challenge is to make sure the greatest amount of hardware is being utilized on each clock, so that the desired efficiency is achieved. An example of a modern VLIW architecture is the IA-64 architecture used in the Intel Itanium processor. VLIW architectures emphasize simpler hardware and high parallelism, but with very complex compiler design required.

<u>Processor Architectures – DSP</u>

Digital Signal Processors (DSPs) are a class of processors that are optimized for high throughput computations on large amounts of data. Many DSPs operate on *streams* of data, and are optimized for arithmetic operations on those streams. DSP does not specifically associate itself with any specific architecture, and so a DSP is not, for example, always RISC, or CISC, or VLIW, although in fact DSPs exist that can be categorized as such. DSPs are processors, in that they do process instructions and operate on data, but compared to a GPP, a DSP is typically not designed to handle peripheral operations, or control flow, rather, it is mainly for processing arithmetic operations. DSP instructions are typically simpler than GPP instructions, and typically use fixed-point formats as opposed to floating-point. DSPs use integer, fractional, and custom numeric formats whereas GPPs typically use standard integer and floating-point formats. Most instructions are based on multiply-accumulate (MAC) operations and are scheduled by the compiler, while GPPs many times use more complex, dynamically scheduled instructions.

All this being said, there are a few types of DSP architectures that are more common than others. DSP designs based on dual MACs, SIMD, VLIW, and superscalar DSPs are common. A dual MAC design uses two multiply-accumulate units run by one instruction, in order to process incoming data in parallel. SIMD refers to running one single instruction on multiple data values, and superscalar and VLIW were discussed previously. In general, DSP architectures emphasize simplicity in programming, complexity in hardware, real-time results and high throughput on large amounts of input data. Various custom DSP designs have been developed, in order to very closely match the needs of an algorithm domain with more customized hardware than if a GPP were used. Examples include the TICC55x SIMD machine, the TigerShark VLIW, and ZSP superscalar processors.

Fetch Order, Cache, and Order of Execution

The earliest processors ran all instructions in order, as written by the assembly or machine code programmers. Many modern processors have out-of-order execution, and a host of newer architectures that correspond to out-of-order processing. Instructions can be fetched in order, possibly more than one at a time, then entered into scheduler hardware, where the decision to execute the instruction is made based on the availability of hardware modules, and based on dependencies that might exist between instructions. Out-of-order processors are typically superscalar, and have multiple ALUs, memories, etc. These processor also employ cache, either for instructions, for memory accesses or for both. Cache is used as a small, fast memory used to speed up operations that would otherwise have to access larger slower memory. For example a processor might be designed to access a hard-disk, but for faster operation and less slow memory accesses to the hard-disk, a small static RAM can be used as a cache, and can increase efficiency. Memory hierarchy design is a topic in itself, and will not be covered in depth here, except to say that the memory system design can be just as important as any other decision in processor system design. For small ultra-low-power embedded processors, typically only some form of EPROM, EEPROM and RAM are used, and larger memory structures utilizing multiple levels of memory are not an issue. On the other hand though, silicon is becoming increasingly smaller, and even low-power processors such as some ARM core based devices include pipelined designs with cache memory. The textbooks on computer architecture [29] by Hennessey and Patterson are great resources to learn more about memory design and are offered here as reference. As we will see in Chapter 5, in this study cache memory is not needed in order to design an efficient processor for GFM.

Floating-Point to Fixed-Point Conversion

This sub-section is about a very important software implementation technique. If done correctly, the technique of converting floating-point algorithm code to fixed-point can yield a design with higher performance and less area and power. In fact, although this technique is technically a software technique, and can be done on its own without hardware customization, it can also be a part of a hardware/software co-design flow (to be discussed in the next chapter), where hardware is customized with the resulting fixed-point algorithm in mind. It is certainly the case that converting the GFM algorithm to fixed-point is a good choice, and the technique will be detailed here.

Numeric Representations

The term *fixed-point* [20][21] refers to a numerical representation where the position of the decimal point is fixed relative to the bits in the number. For example, the binary number 0101 can be used to represent the decimal value 5, if we assume the decimal point is fixed to the right of the right-most (least significant) bit. The same value 0101 can also be used to represent 2.5, if we assume the decimal point is fixed to the left of the same bit. Fixed-point numbers can be signed or unsigned, but the main idea is that the decimal point is fixed in place. A *fractional* form of a fixed-point number simply refers to a fixed-point number where all bits are understood to represent bits in behind the decimal point, so that there is no integer part.

Fixed-point format is widely used across the board in terms of hardware implementation options, including GPPs, DSPs, and even ASIPs and ASICs. It allows for simple hardware to perform arithmetic operations on these values in a way that takes less area, time, and energy than floating-point formats. The reason is that fixed-point hardware does not need to know or even care about the location of the decimal point. For example, if you want to add binary numbers 0100 and 0110, you can assume the value is an integer (non-fractional) and add them for a result of 1010. The corresponding hardware works the same way. Using a basic 4-bit adder digital logic circuit, you can input both numbers and get that result. Now if in fact the decimal point was understood to be in front of the first bit (from the right), that does not change the answer.

To see this more clearly, note that adding binary integers 0100 and 0110 amounts to adding decimal 4 and 6 for a result of 10. This matches the binary value of 1010 we got from adding the binary numbers. Now suppose there is a decimal point such that we are adding values 010.0 and 011.0. In decimal, we are adding the numbers 2 and 3, and we should get 5. If we take the same result 1010 and place the decimal point in the same position in the result, we see that in fact 101.0 is still correct, having the value of 5. The point is that on a basic level, fixed-point representations require only simple binary arithmetic hardware, and as we will see that is much simpler than floating-point hardware. As we will see, there are enhancements that can be added to fixed-point representations and to the corresponding hardware, but on a basic level the hardware is as simple as described here. For example, in fixed-point implementations, *guarding*, *rounding*, *truncation* and *saturation* play an important part in defining the underlying ALU operations. Fixed-point formats are denoted as $Q_D(m.f)$, where D refers to the number base (2 for binary), m is the number of integer magnitude bits, and f is the number of fractional bits. Using this notation, $Q_2(16.4)$ refers to base 2 binary data with 16 bits integer part and 4 bits fractional.

The term *floating-point* (f.p.) refers to a fractional numerical representation where the decimal is not in a fixed position. In other words, the decimal point "floats" within the value, and the meaning of a floating-point formatted number depends on knowing its location. Floating-point representation reserves some of the bits in the value for a fractional integer part called the *mantissa*, and some for an *exponent*. A popular format is the standardized IEEE 754 Single-Precision Floating-Point Format [37]. This format uses 1 bit for the sign, 8 bits for the exponent and 23 bits for the fractional part. There is also a bias on the exponent, a value of 127, and special encodings for zero and infinity. Note that performing arithmetic on floating-point numbers means handling the various parts of the value separately, and floating-point hardware is relatively complex.

Although floating-point hardware requires more complex operations than fixed-point, it can be efficiently parallelized so that multiple f.p. operations can take place at once. Pipelined floating-point ALU designs exist that break up the various operations in floating arithmetic into pipeline stages. For example, in [36], a FPU for add and subtract operations breaks up the operation into 5 different stages including sub-normal detection, pre-normalization, add/subtract, post-normalization, and exception handling. This sort of pipeline allows otherwise cycle intensive f.p. operations to be completed on each clock, when many such independent operations need to be performed and can be correspondingly started on each clock. Pipelined floating-point multiplication and division can be much more complex than addition and subtraction, and pipelined designs exist for those operations as well.

When speaking of floating- and fixed-point numerical formats, we need to also understand a bit about *noise*, *distortion*, *dynamic range*, and *precision*. The dynamic range of the numerical format refers to the maximum value - the minimum value. For example, for 8-bit data, the dynamic range is 255. IEEE 754 Single-Precision Floating-Point Format has a range of 3.4e^38 – 1.18e^-38. Range for fixed-point numbers depends on the number of bits, and how many bits are for integer and for fraction. The precision is the smallest value that can be represented, for example the number 001 for integer systems. Distortion is when the input to output function is nonlinear. For example, when data overflows from arithmetic operations, truncation can introduce distortion since the true answer should have been different.

When using floating point formats, you get a very high dynamic range and precision, but also slower and more expensive and complicated hardware. With fixed-point, you get low range and precision, but very fast and cheap hardware. It depends on what the application requires as to whether to use one or the other. For example, if a system is to be built that accepts integer data from 0 to 100 as input and always produces integer data from 0 to 10 as output, there would be no reason for using floating-point hardware. On the other hand, some applications require many high dynamic range, high precision calculations to be performed quickly, and floating-point might be the only possible solution.

Conversion

We have seen how different numeric formats give different dynamic range, precision, and require more or less complex hardware. There is another point then that needs to be discussed, which is whether or not to convert an algorithm from floating-point to fixed point. Of course, the decision is based on whether or not the increased range and precision is required by the application, and whether or not the benefit of the more complex hardware outweighs its cost. Converting from floating-point to fixed-point is complicated, but if done well, can produce a much more energy efficient implementation with smaller and faster hardware, with only modest reduction in accuracy. As part of a custom hardware design flow, converting to fixed-point involves some work on the algorithm up-front, but with the benefit of reducing design time in lieu of having to design complex floating-point hardware.

If we were able to use infinitely many integer and fractional bits, we could easily convert from floating- to fixed-point, but the issue lies in the fact that we cannot. We have to choose how many bits to use for the integer part, and how many for the fractional part. In other words, we need to set the inherent range and precision in the system when converting to floating-point. This can be daunting at first, since, when working with floating-point in a high-level language, most of the underlying details of the numeric format are not even paid attention to. In a fixed-point system, the choice of format determines the accuracy of the system outputs. Given a finite number of bits, either we choose a high range, or high precision, or some trade-off point, but in any case we usually end up with less accuracy as compared to the floating-point version. How much of a difference is noticeable and acceptable is up the designer.

One part of this conversion process is called *scaling*. Scaling refers to the idea that once you constrain all your values to a fixed-point format, you now need to make sure results of intermediate computation do not exceed the maximum or minimum values of the format. For example, if the fixed point format is 4 bits integer, and 4 bits fraction, and you want to multiply 5 times 5, the result is too large to fit in the format. The problem can be fixed in various ways, one being to make make sure all the values input to that particular multiplication never result in an overflow. If certain values or intermediate values become too large in the process of performing arithmetic operations, then those values can be scaled such that they do not overflow the bounds of the numeric format.

As an example, consider that in a block of code from a program, you want to multiply three numbers A*B*C. Suppose the numeric format is $Q_2(8.0)$, 8-bit integer data. Now, if the values of A, B, and C all range from 0 to 2, the largest result is 8, which is well within the range of an 8-bit integer. The problem arises when you have three values that can each range from 0 to 125, where the largest result of A*B*C cannot fit within the numeric format. What you can do then, is to scale one or more of the variables A,B,and C, so that given the typical range of values for the three variables, the result of this operation never overflows. For example, if all values were scaled so that they were all 6 or less, 6*6*6 would fit within the format.

Of course with fractional data, scaling becomes more difficult since you are dealing with possibly very small data as well as large data, and scaling can effect the accuracy and precision of results. Although you might scale values initially so that arithmetic results do not overflow, you eventually may need to scale values back to put the number back in the correct format. If you have three numbers A, B, C all in $Q_2(18.14)$ format (32 bits with 18 bits integer), and you scale them all by dividing by 100, then you multiply all of them together, you need to multiply the result by 100 to get the true answer back. By doing this, you can allow the operations to take place without overflowing or otherwise violating the numeric format during intermediate operations, and by choosing the scale factors carefully, you can optimize how much range and precision you get out of the final answer. Note that this is all in an attempt to get as much range and precision as possible out of a fixed-point format using simpler and cheaper hardware.

To do this, theoretically you could monitor (by profiling) the value of each variable and each intermediate result, while iterating through different scale factors for each variable and arithmetic result, to achieve the highest possible optimization. It is very important to not that each time to introduce a scale factor operation, you introduce more cycles in the operation of the processor running the algorithm. This means to optimize scaling you need to do it as little as possible, and in just the right places in the algorithm. In addition, the input vectors used to choose the scaling factors have to be robust enough to allow you to choose the right values and the right variables to scale. The method used by the author in this study was to run the GFM algorithm and output variable and intermediate result values to a file while running the program. Values that had very large (or small) values or intermediate operations that resulted in very large (or small) values were scaled appropriately. At each iteration, the scale factors were refined so that eventually all variables and intermediate values took full advantage of the numeric format, never overflow or violate the format given valid input data, and so that results are as accurate as possible. The GFM algorithm was in fact converted from floating-point to fixed-point in this study. The resulting format used was $Q_2(18.14)$ with various scale factors introduced into the code on certain variables and after certain intermediate results of arithmetic operations. The efficacy of this transformation will be discussed more in Chapter 5.

Chapter 4: Hardware/Software Co-Design and Design Flows

In the previous chapter, it was hinted at that an ASIP solution best fits the GFM algorithm and dive computer application. It will become more clear in the next chapter why that is, and for now we turn our attention to a related topic – hardware/software co-design. Hardware/Software Co-design refers to the fact that when a customized processor is designed, it requires not only hardware design, but also software design, and in fact they need to occur concurrently particularly within the design flow of ASIP, or DSSP/DSP. This chapter will explain a typical design flow for ASIP type hardware, while clarifying what is meant by hw/sw co-design and why it is important.

ASIP Design Flow

There are many ways to describe the design flow of an ASIP, but the one chosen here and used in this study is as described in [17]. The design process is one that includes starting with design inputs, partitioning the design between HW and SW, following parallel paths between HW and SW design, and ending with integration and testing. We can visualize this as given in the diagram in Figure 14. Note that both the HW ans SW design flows start with a specification, then either modeling or programming, simulation, and eventually implementation. The arrows in between the HW and SW design flows illustrate the point that HW and SW need to be designed concurrently, and in fact both sides influence and dictate the design of the other side. The most clear example of this is that of when an ISA and firmware is defined for the hardware, the programming usually has to change to make full use of that choice. If, for example, in hardware a module is designed to perform division as on operation, software needs to be re-written so that division is not done in software, but rather by invoking the new hardware module. In the case of an ASIP, where custom modules, memory structures, and instructions are designed for a specific application, it is required that for the most efficient design that a HW/SW co-design process is followed.

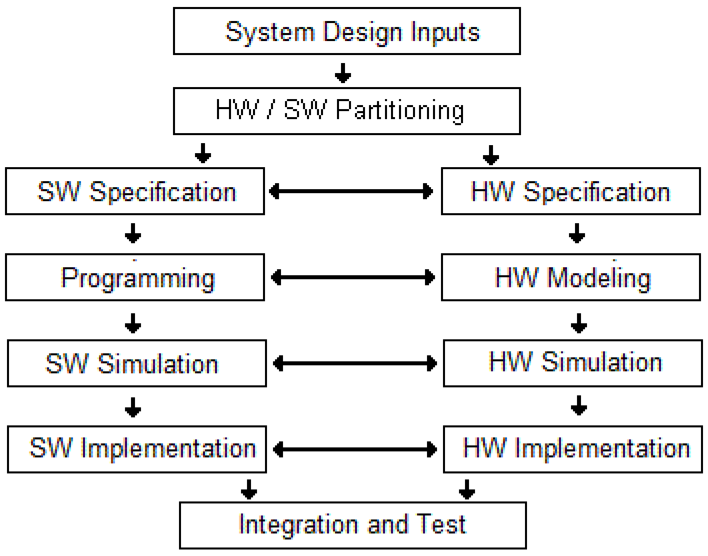


Figure 14: HW/SW Co-Design Flow

Under this umbrella of HW/SW Co-Design are the topics of firmware design, benchmarking, application and instruction set profiling, requirements specification, and of course all the topics falling under the processor core design. An ASIP design typically involves multiple teams, each of them containing experts in each sub-topic, utilized in order to arrive at the most efficient design possible. Experts from both ends of the spectrum including software design and programming, processor design, application profiling, firmware design, integrated design environment (IDE) designers, and even VLSI designers all can have inputs into an ASIP design. The goal is to accurately define and design hardware for the specific application such that no other implementation, except possibly a full ASIC design, can compete in terms of performance and energy efficiency.

In recent years a good deal of effort has gone into designing automated IDEs that can efficiently profile an application, define and implement custom hardware and firmware systems, and produce an ASIP design automatically. Some tools available use so-called architecture description languages (ADLs) for use in describing higher-level architectures, so that different options can be iterated through and to eventually arrive at the most optimal architecture for a given application. These tools are extremely expensive, but can reduce the design time considerably compared to a totally manual design.

For example, one such product quoted by the author of this paper was on the order of $50,000 for a single license. This cost comes along with a high learning curve to learn the tools and make efficient use of them. This means such tools can in fact be out of reach for many designers, either because of the time-line needed to learn the tools, or because of the associated cost. Another point is that these tools rely on somewhat pre-defined architecture options, and so can actually be limiting in terms of their capability to output efficient designs for specific applications. For example, the Processor Designer tool [38] from CoWare allows a designer to specify the number of stages and other options within a RISC-type pipeline design, in order to find the most efficient implementation. Another tool performs analysis on high-level C-code in order to automatically decide what custom instructions can be implemented to increase efficiency.

Chapter 5: Optimizing Hardware for GFM

Summary

In this chapter a specific design for optimizing hardware for the Gas Formation Model will be presented. The chapter will be loosely organized following the ASIP design flow described in Chapter 4 and illustrated in Figure 14. Because the design flow is iterative, and moves back and forth between hardware and software design, the information here will not be entirely linear. Information will be presented on each relevant section, and then we will revisit various sections as required. This chapter starts with information on high level application profiling, then hardware/software partitioning and system design, followed by low level profiling. Note that all that was done using the algorithm in its original form as a high precision floating-point routine. Later the results of floating-point to fixed-point conversion are given, followed by fixed-point profiling results. From the large amount of profile information, later sections discuss design choices made at an architectural level, in terms of instruction set and firmware design, processor core architecture, memory and i/o system design, and toolset design. All this is part of the hardware/software co-design flow described in Chapter 4.

High-Level Application Profiling (Floating-Point)

In Chapter 1, we explored the application itself, and defined performance, power, and area constraints. Although we know the application is relatively demanding, we need a more concrete measure of how demanding it is, in order to be able to select a design path for customizing hardware. In addition, we need to make a reasonable decision in terms of which portions of the program to accelerate with custom hardware. These choices are made by doing thorough application profiling, and this section will describe results of such profiling, and decision made based on those results.

Application profiling has multiple uses. At a high level, profiling of the high level language version of the algorithm can help roughly identify which instructions are executed most often, and correspondingly where hardware acceleration may help optimize the system. There is a rule of thumb mentioned in [17], termed the 10/90 rule. What this refers to is that for application that can benefit from acceleration by hardware, many times 10% of the code is executed 90% of the time, and so in that case the code that represents the most executed 10% is the portion you want to focus on accelerating. Of course whether there is a block of code that fits those numbers exactly depends on the application, and the specific ratio is not important, but the idea is that is that you want to identify which code needs to be optimized or not. In fact the GFM algorithm code running inside a dive computer falls into this category and so is a good candidate for acceleration.

We can begin by looking at profiling results on the high-level C-code in full precision floating-point format. Using this program implementation, each line of C code can be given a count each time it executes during the dive sets 1 through 4 given in Chapter 1. Recall that each dive includes the real-time state computation that gives the free gas related to the specific dive being simulated, as well as all the prediction calculations run at each and every time-step as explained in Chapter 1. This typically amounts to millions of time-steps being run over the course of each dive. By looking at line counts for each line of C-code, we can see if there are blocks of code that are good candidates for acceleration.

At this point it is important to take a look again at the dive computer as a system, at a higher level. We will be deciding here that it is the entire base GFM algorithm routine that needs to be accelerated, but we need to take a step back to understand why. Recall that dive computer performs many functions, including storing dive data in non-volatile memory for later retrieval, controlling and displaying information on the output display, and processing user input. Because the author works for a dive computer manufacturer, it can be reported here that for a top-of-the-line product like the OC1 made by Oceanic [39], there are over 50,000 lines of compiled and hand-written assembly code. The code for the GFM routine is less than 100 lines of C-code. When compiled, the actual count of assembly lines of code of course depends on the compiler and processor used. If we say that each line of C-code converts to 50 lines of assembly (to be extremely conservative), still we can say that the GFM algorithm makes up only 10% of the code. From compilation results using an IAR C-compiler for an ARM7TDI 32-bit target device and for a 16-bit MSP430 target, with full optimizations on, the floating-point version program code compiles to under 4kb in both cases, whereas the entire dive computer product compiles to well over 100kb of code.

Dive sets 1 through 4 from the standard dive profiles represent 96 different dives, spanning thousands of minutes and literally trillions of lines of executed c-code. The dive descriptions were given in Chapter 1, and here we will analyze the instruction profile. In a custom, automated program designed for profiling, each line in the basic GFM routine's main loop (after initialization), was given a line number. Each time the line was executed a count was incremented corresponding to that line. The results from a selection of the 96 dives are given in Figures 15 - Figure 24. Note that each figure includes 4 dives, representing dives out to the NO-D limit, out to ½ the NO-D limit, and with and without a safety stop of 3 minutes at 15 ft. Analysis of the figures follows.

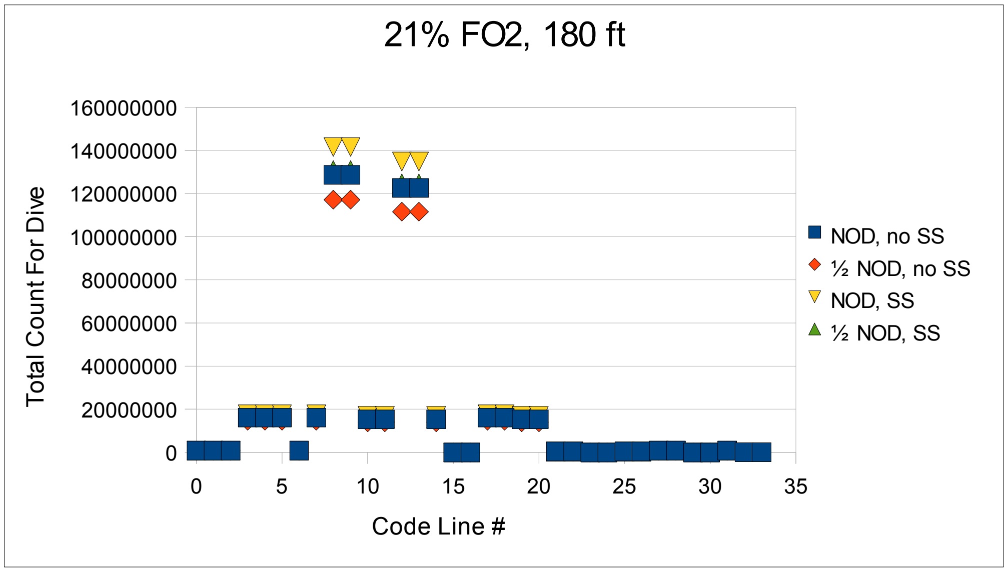


Figure 15: Instruction Profiling Results, 21% FO2, 180ft

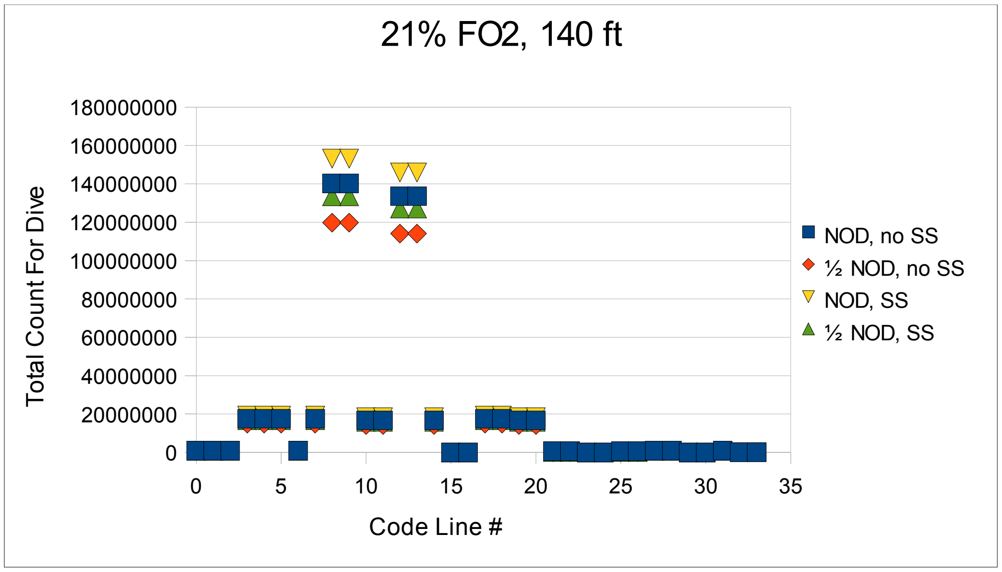


Figure 16: Instruction Profiling Results, 21% FO2, 140ft

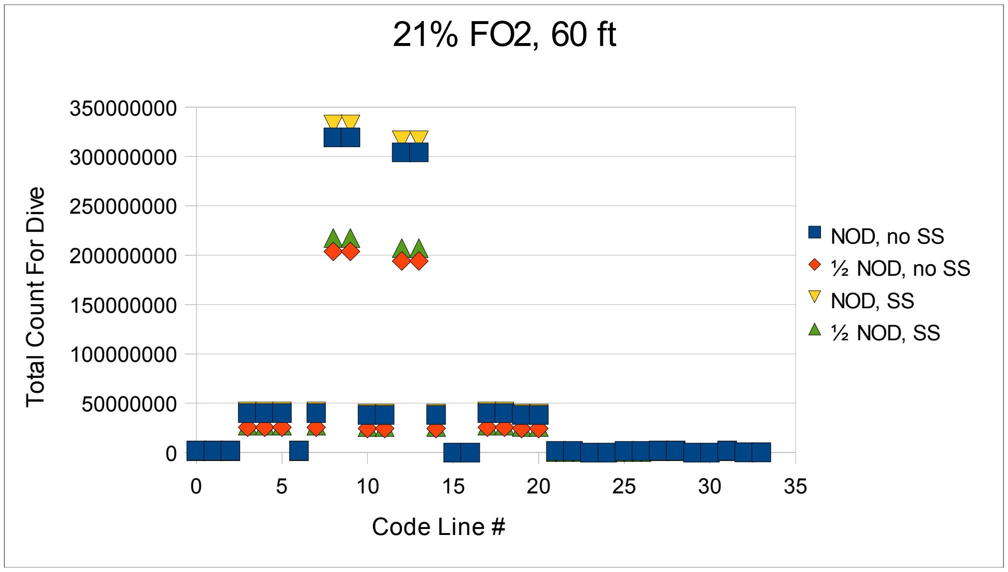


Figure 18: Instruction Profiling Results, 21% FO2, 60ft

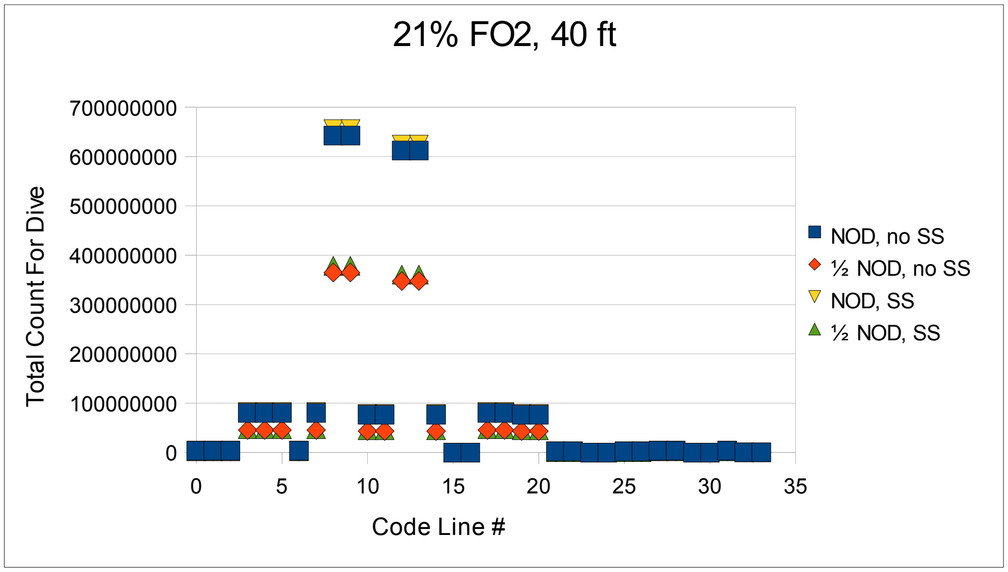


Figure 19: Instruction Profiling Results, 21% FO2, 40ft

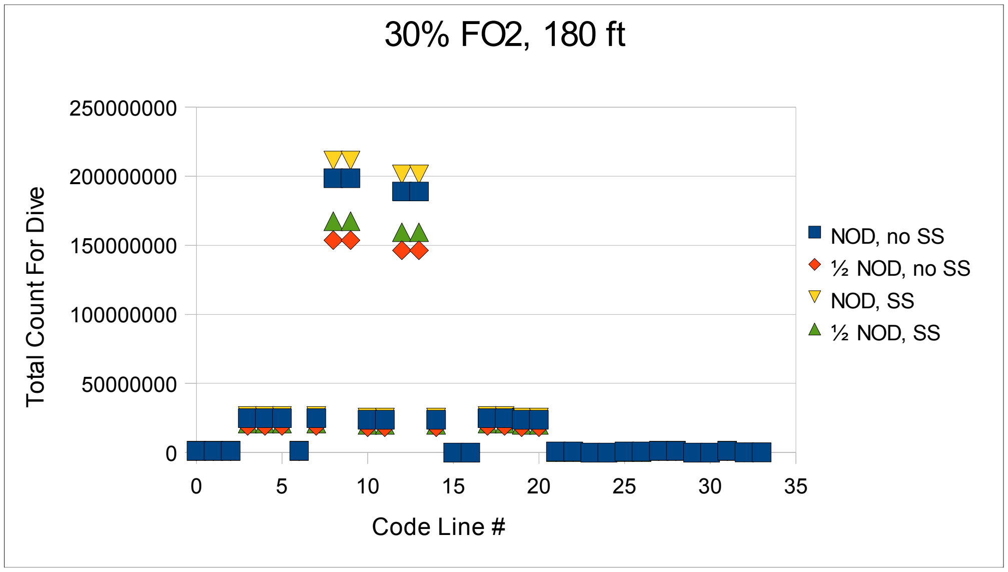


Figure 20: Instruction Profiling Results, 20% FO2, 180ft

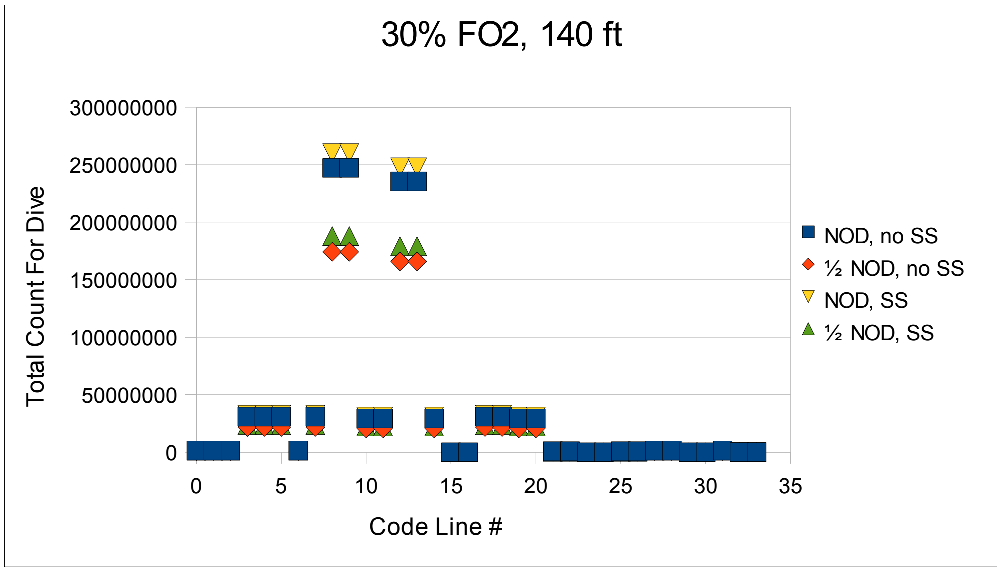


Figure 21: Instruction Profiling Results, 30% FO2, 140ft

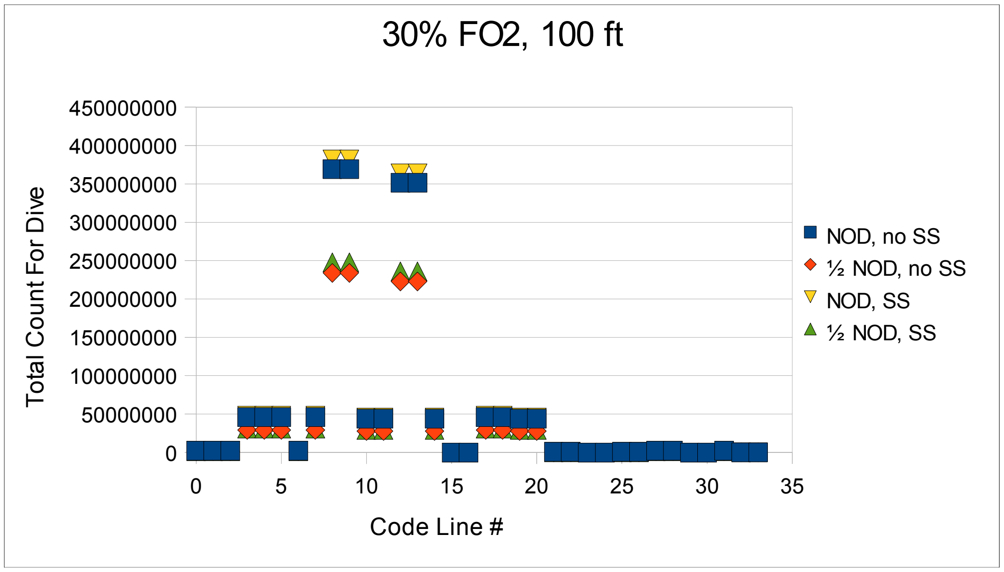


Figure 22: Instruction Profiling Results, 30% FO2, 100ft

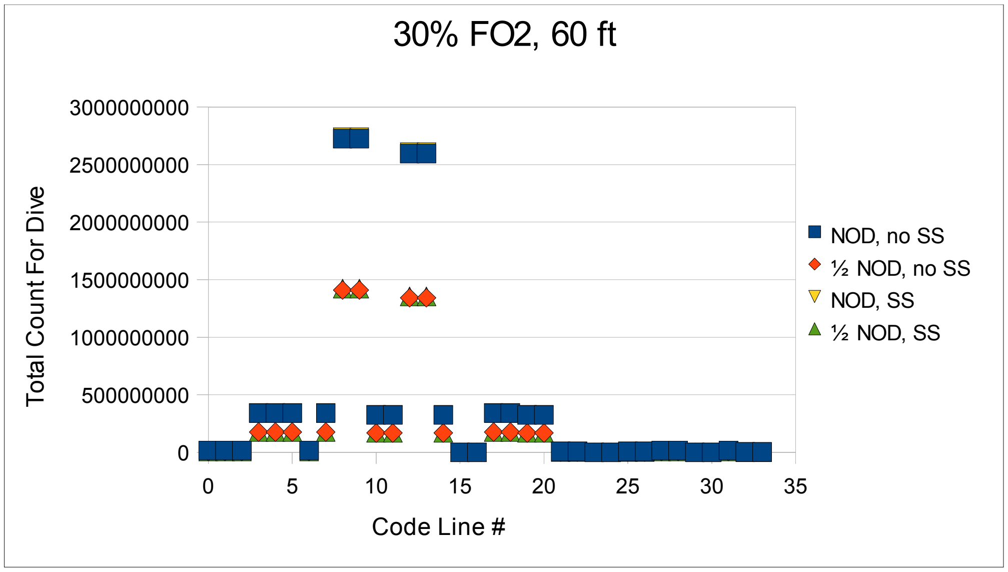


Figure 23: Instruction Profiling Results, 30% FO2, 60ft

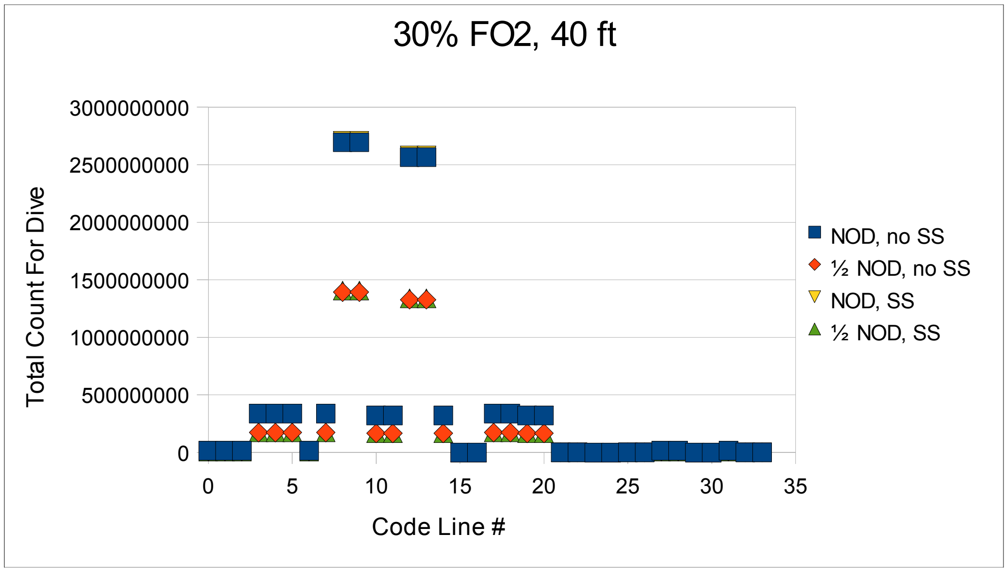


Figure 24: Instruction Profiling Results, 30% FO2, 40ft

In the profiling results above, we see that over the course of the dives, the GFM routine (including prediction calculations at each time-step as defined in Chapter 1) amounts to hundreds of millions of executions of particular lines of code within the GFM time-step routine. For example in Figure 24, lines 8 and 9 were executed between 1.4 and 2.6 billion times over the course of the length of the dives. We do see some differences with the different dives within each figure, but those differences are expected since dives without a safety stop are inherently shorter, and dives that go out to ½ of the NO-D time are shorter as well. Shorter dives always have fewer lines executed, since the number of times the time-step is run is directly related to the length of the dive.

Within each time-step there are some variations as well, and this is what is called program *phase* information. Phase information is related to branches in the program, and whether or not branches are taken or not taken given different inputs. Fortunately for GFM, the only time the time-step routine branches is where free gas either forming or not forming. For example, between lines 30 and 35 in the reference program code of the GFM patent, there is a condition that is tested such that if variable p(nx) is greater than pN2, then variable GF(nx) is calculated. This value for GF represents gas formation, and this condition being tested depends on the dive profile being run. The point is that different dives will either satisfy or not satisfy this condition, and it depends on what portion of the dive is being run at any given time. In the beginning of a dive, where the diver has just descended to the bottom depth, for example, gas has not yet formed, and so the condition will not be satisfied. What this means is that this condition contributes to the program's phase, and so accurate cycle counts and profiling results depend very much on what type of dive and in what phase of the dive you are analyzing.

It turns out, however, that in fact as you can see from the aforementioned figures, this phase information is hardly noticeable. For example, if we look at Figure 15, line 20 represents the "if" statement condition, and line 21 represents the condition being satisfied. If you take the counts for each line, and normalize over the number of times the routine is called, you will see that in fact line 20 is called exactly 20 times for each dive, as expected. Line 21 however, is called a different number of times for each dive. For the first dive (NO-D, no ss) it is called 478,392 times (normalized as 0.624 times per time-step), for the second dive it is 0.371 times per time-step, 0.557 per time-step for the third dive, and so on. In all cases, given these 96 dive types, line 21 is called less than once per time-step, while others are called as much as 168 times per time-step, and in fact the average normalized value over all dives is 0.43 times per time-step.

Beside interesting observations about particular lines of code, what is more interesting is that throughout all the dives the overall characteristic is the same. As was mentioned previously, where there is more or less dive time there are expected differences, but on average we can say that the instruction profile is more or less constant when normalized to count per time-step. One important observation we want to make is about the high counts within the routine. Since these lines of C-code correspond to multiple instructions in assembly, and multiple execution cycles, we can say that the GFM routine requires literally trillions of cycles in execution during typical operation for a typical dive scenario. Even if we consider a processor architecture that has a deep pipeline that parallelizes operations, unless we use multiple processors in addition, cycle counts over a dive and over the life of a product are staggering. On the other hand, this observation that it is in fact the GFM routine that needs to be optimized offers an opportunity where customized hardware can bring the design concept from the realm of the theoretical to that of practicality.

Although we cannot easily provide here cycle counts for an entire dive computer system including all the management, storage, user input, and output display activities, we can in fact say that compared to GFM, those operations are entirely negligible. One way to see this roughly is simply by timing the other peripheral activities in the dive computer, and comparing those to that of running GFM within the same system. The OC1 product uses a 1/8 second interrupt system where all peripheral activities performed have to fully complete within that time. Most activities are complete well within 50ms, although some display routines require up to 110ms.

If the required GFM computation was run within the same sort of system like the OC1 on a 16-bit MSP430 device, it would require roughly 227000 cycles per time-step. Running at a relatively high frequency of 8 Mhz, and assuming a single processor system, 227000 cycles would take 28ms per step. Considering that Constraint 2a stipulates that we need to perform 2711 time-steps every 6s, and that 2711 time-steps would take 76 seconds, it is clear that the constraint cannot be met, at least using a 16-bit microprocessor like the MSP430. We can consider a much more powerful and efficient processor like an ARM7, but even then at 45007 cycles per step, the 2711 time-steps would still take over 15 seconds to complete, thus still not meeting the constraint.

Note that given a specific frequency we can in fact calculate how many cycles an implementation cannot exceed in order to just meet Constraint 2a. At 8 MHz for example, an implementation would need to process time-steps within 17705 cycles in order to meet Constraint 2a. In fact, a better design should meet Constraint 2b, and so we can say that a processor should be able to process each time-step in half that time (8852 cycles) while the other half of the time it is held in a low-power state. Again it should be noted that this is in reference to a one-processor system running at 8MHz. Dive computer products like the OC1 rarely operate at frequencies above 8MHz due to their low-power nature. This goal is aggressive, but can be done, and the design choices made in this study will show how this goal can be met.

From examining the profile information at a high-level, we have learned a couple things, and make a couple design decisions. First, we know that the computation of the GFM algorithm is in fact the most important portion of the program to focus on in terms of customizing hardware. Within a dive computer system, and over the life of typical operation and even the life of the product, the computation of system activities is totally negligible when compared to computing GFM. That means any hardware acceleration should be focused on accelerating GFM itself, and not other important dive computer code. In fact, even higher-level routines related to GFM (like simulating dive profiles, retrieving and storing FGV values and prediction results, interpolating or calculating on prediction results, etc) are also negligible when compared to the immense cycle counts required for the basic GFM routine to run while meeting performance constraints.

Secondly, we know that we should accelerate the entire base GFM routine, as opposed to finding portions within the routine to accelerate. Because of the incredibly high numbers of calculations needed, even lines of code that are only executed once per step are still in very high numbers over the course of a dive, or the the life of the product. Some lines are executed more than others, but on a whole, hardware needs to be customized so that the entire routine can be run in a number of cycles that amount to orders of magnitude less than the current implementation.

HW/SW Partitioning

For the reasons explained in the previous section, we can now present a block diagram of what a customized GFM dive computer system should look like at a high-level. The concept is in fact intuitive, but the analysis was required in order to make sure intuition was not hiding unseen factors. In fact intuition is correct, and so the concept for the most efficient GFM-based dive computer is that of a GFM co-processor. The concept of a GFM co-processor is that a customized processor should be designed that primarily computes the base GFM routine itself, much more efficiently than a GPP can. The rest of the dive computer system would be handled as is done today, using a low-power general-purpose processor, and should have an interface to input and output information to an from the GFM co-processor. The concept of a co-processor is descriptive, but in terms of terminology we will simply refer to it as a custom processor for GFM.

Figure 25 shows such a processor interfaced with a main dive computer system processor. Note that this partitioning of the dive computer system also takes into account the i/o requirements of communicating with the GFM processor. Although the GFM routine requires a great deal of computation and efficiency, it needs to communicate with the main microprocessor only every 6s. Communication to and from the main microprocessor is needed in order to input the current depth, and to output the current computation results of either the real-time state or prediction results. In fact the data can even be processed on the main micro, and only the most basic GFM routine can be handled in the custom processor.

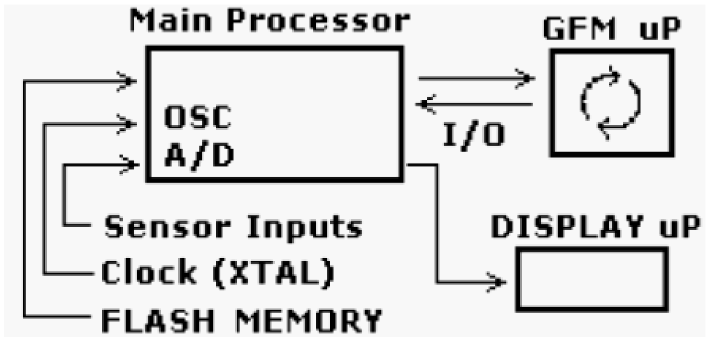


Figure 25: GFM-based dive computer system concept

This type of system design would also allow for GFM to be added to any product with minimal design time. Given a well designed, efficient and high performance custom GFM (co-)processor, the dive computer system designer needs only to drop in the IC, interface to it, and process the data according to the specific product design for which GFM is being used. Because GFM models physiological phenomena in the human body, there are various other applications where it can be used, for example medical visualization software, where GFM and the custom co-processor could play an important part in optimizing computation.

Low-Level Application Profiling (Floating-Point)

To design custom hardware, as we saw in the last chapter, part of the design flow is also to profile the application at a low level while concurrently making changes and redesigning both hardware and software. To do this, you need to have an initial hardware implementation for which to code the algorithm. When the goal is to design custom hardware, what you can do is start with a reference processor core and implementation. The first phase of this project involved using the full-accuracy floating-point algorithm code version unchanged, and to profile and customize an out-of-order RISC-based processor core using the Simplescalar tool set. SimpleScalar is a freely available simulator that can be used to evaluate different architecture options for various targets, specifically including the PISA, MIPS-IV-based instruction set architecture (ISA). It can simulate and characterize a 32-bit out-of-order pipelined processor architecture, using specific hardware options selected by the user. It is also extensible, with the ability of adding custom instructions and custom hardware units. This was done as a reference point, as a source of profiling information on a RISC-type processor core, and to investigate how aggressively a modern out-of-order processor could be customized to reduce the cycle count given the GFM algorithm as the benchmark. The gcc cross-compiler version 2.7.2.3 ported for the PISA architecture was used with high optimization option -O3. Application profile information was obtained using sim-profile from the tool set, and the out of order simulator sim-outorder was used to obtain cycle count information.

It was desired to find a correlation between the hardware architecture options (cache levels/sizes, number of arithmetic units, etc) and the cycle count output. To take interactions between parameters into account, and for a more robust evaluation, a Placket and Burmann Statistical Fractional-Factorial Design of Experiment (DOE) methodology was used, and the methodology described in [40] was followed almost exactly. Area estimation was performed using the SimpleScalar Estimator tool [41]. Simulations were performed and design changes were iterated systematically to reduce cycle count. The purpose of the P&B DOE is to isolate those design parameters (in this case Simplescalar architecture options) that contribute most to cycle count. By doing this, you can then set those parameters with highest influence on cycle count to a relatively high value, and those with lesser impact to lesser values, in order to not waste resources and power. For example, if the DOE showed that the number of integer ALUs was the most influential, while cache size is not important at all, then the number of integer ALUs might be set to 4 while cache might be eliminated. In this way, the DOE results can be used to customize the Simplescalar core such that the design specifically caters to the needs of the GF algorithm, and then by profiling GFM on that configuration we can gain more insight into how a totally custom core can be designed for more efficiency.

TABLE I: Simplescalar Configuration

Sim-outorder Parameter	Value
RUU Capacity (# instr.)	64
# Integer ALUs	3
Branch Mis-pred. Latency (cycles)	1
Issue Width (instr. / cycle)	4
Load/Store Queue Capacity (# instr.)	32
Decode Width (instr. / cycle)	4
Instruction Fetch Size (# instr.)	8
Bimodal Branch Predictor Table Size	256
# Floating-Point Multipliers/Dividers	1
Width of Memory Bus (# bytes)	4
# Floating-Point ALUs	1
# Integer Multipliers/Dividers	1
Data Cache	None
Instruction Cache	None
Fetch Speed	1
Memory Latency	18,2
Cache:icompress	true
bpred	bimod

Table I: Simplescalar configuration for customized PISA core.

The DOE led to a high performance design with a cycle count of 2418 cycles, a cycle per instruction (CPI) of 0.384, and with an estimated area of 2858 millions of square lambda and 1.5 million transistors. The customized PISA core design would meet the real-time performance constraint with a modest frequency of just over 1 MHz, and the corresponding Simplescalar parameters are given in Table I. Those parameters not listed in the table were given default values. Note that this aggressive design customization for performance amounts to a 95% performance speedup (a factor of 18) as compared to the cycle count on ARM7TDMI for the software floating-point version. This design configuration would be great except that in the DOE the only consideration made was for performance, and so the design is surely not as efficient as it could be.

We can learn a good deal from looking at the profile information from the GFM code run using this Simplescalar configuration. The figures that follow show the results of running the standard dive sets using the customized Simplescalar processor core configuration. We will analyze the Instruction Profile, Instruction Class Profile, Branch Instruction Profile, Addressing Mode Profile, and Load/Store Address Segment Profile. Even though these profile data are specific to the RISC hardware simulated by Simplescalar, they can still give important insight into what the algorithm requires, and this information will be used to make decisions on what characteristics the customized hardware should have.

Figure 26 shows the distribution of the instruction classes used in the program. We see that only 23% of the instructions are actually implemented with floating-point instructions, while 38% of instructions are integer operations. Integer operations are so numerous because they include address computations and other overhead in loops, branches, comparisons, and intermediate operations. The next largest components of the distribution are load, then unconditional branches, and then stores, although they only contribute less than 38% of the instructions. Interesting to note is that on a processor like the 16-bit MSP430, since there is no floating-point hardware, a similar evaluation yields around 100,000 integer operations per loop. Since floating-point instructions take many cycles and use multiple instructions, this reduced instruction count here is expected. For example consider that you have 1475 floating-point multiplication operations. Assume each operation takes 87 cycles, including overhead of 8 cycles for calling and returning from the multiplication routine. That amounts to over 11,000 cycles simply in overhead of calling the f.p. routine. The total cycle count is over 120000, and roughly matches the type of numbers you would see on a 16-bit platform like the MSP430. If we suppose we are running at 8MHz, then 11,000 cycles takes roughly 1.4ms. That means in a given GFM calculation taking 5000 time-steps, it would take the MSP430 75 seconds to complete the calculation, and almost 7 seconds of that would be overhead in calling floating-point routines. With the compilation statistics given here, a running frequency of 8MHz, and an average (ideal maximum) cycles per instruction (CPI) of 1.0 in a 3-stage pipelined architecture, the 75 second calculation would reduce to around 23 seconds. In fact these numbers are in the correct ballpark for actual measurements made on the MSP430 vs. an ARM7-based product that uses a 3-stage pipelined architecture.

Another interesting set of data is that of the most frequent instructions used in the compiled program. Figure 28 shows the 8 most frequently used instructions, comprising 87% of the total instructions in the program. The break down of instructions is as follows: the largest percentage of instructions are floating-point single-precision loads, immediate unsigned integer addition, single-precision floating-point multiplication, unsigned integer addition, single-precision floating-point subtraction, store single-precision floating-point, shift left, and finally single-precision floating-point addition. It is important to note here that instruction counts do not inherently indicate execute time, since some instructions can take longer than others, and memory access times are not included in such a comparison. Other statistics include the branch instruction profile, which indicates that 87% of branches used conditional direct type branches, and the addressing mode profile indicated that 97% of instructions used register + constant addressing.

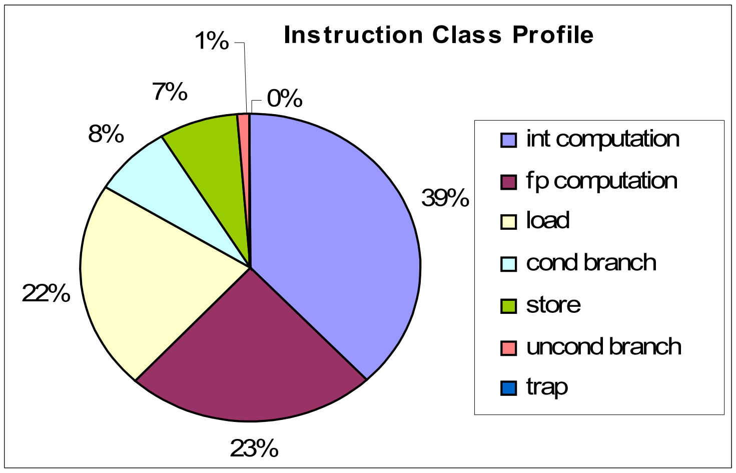


Figure 26: Instruction class profile from Simplescalar profiling results

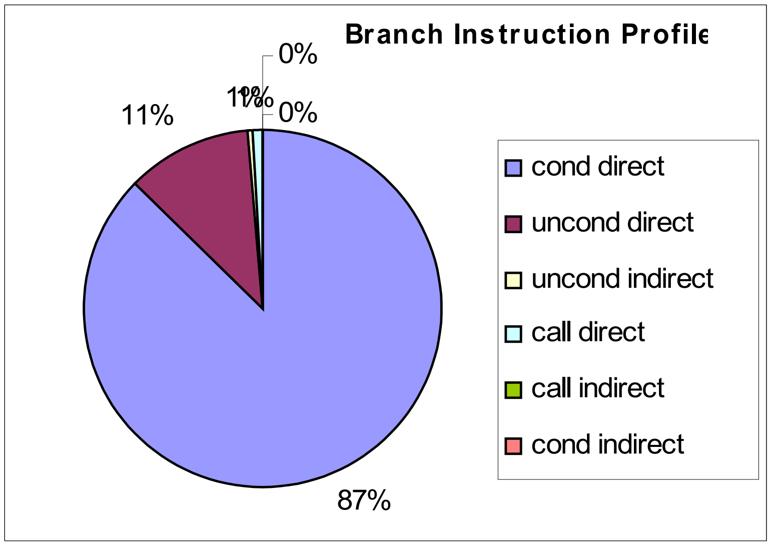


Figure 27: Branch Instruction profile from Simplescalar profiling results

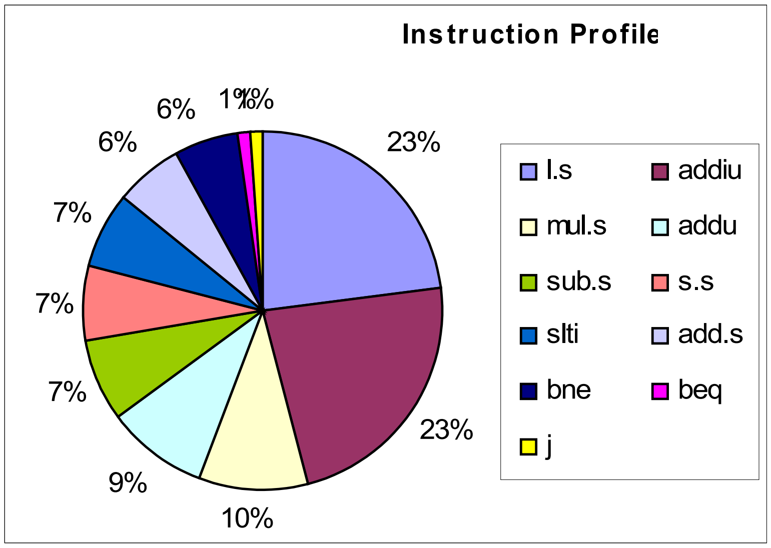


Figure 28: Instruction profile from Simplescalar profiling results

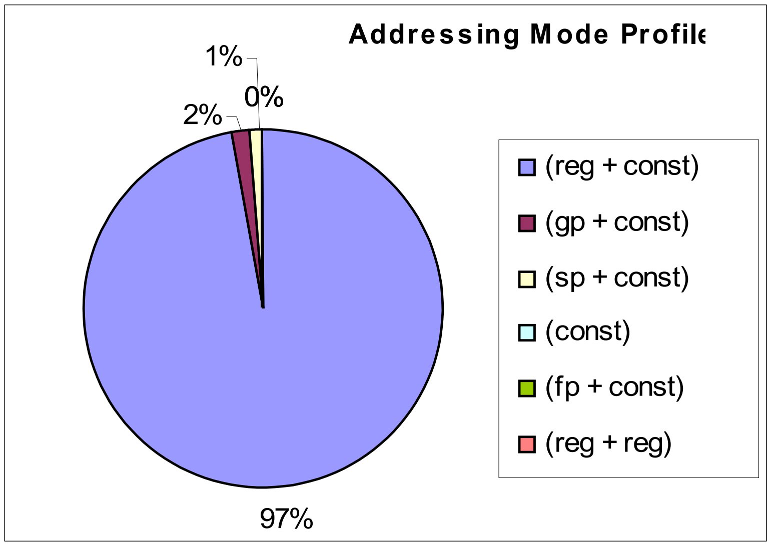


Figure 29: Addressing mode profile from Simplescalar profiling results

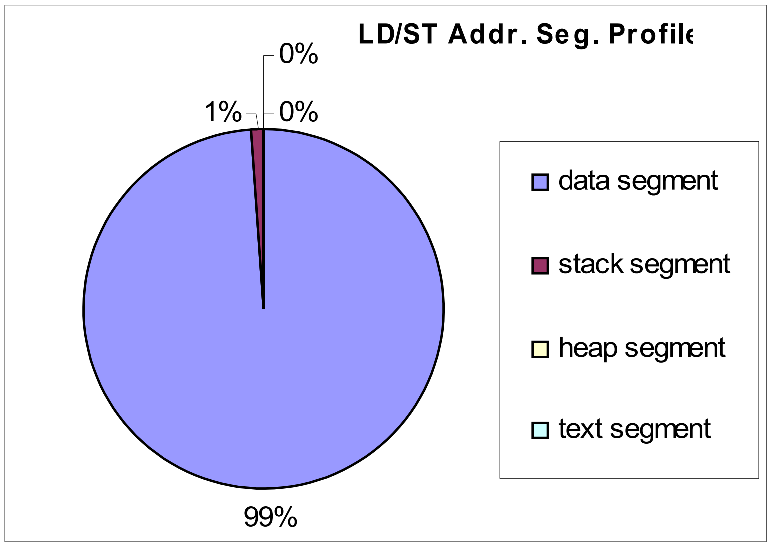


Figure 30: LD/ST address segment profile profiling results from Simplescalar

From the addressing mode profile information we can see that there is no need for so many addressing modes, since 97% of operations use one type of addressing. When designing a custom processor core, a good deal of complexity can be removed by only including the required addressing modes for the application, since the processor would no longer need to have hardware capable of decoding the other unneeded modes. From the load/store address segment profile, we see that virtually all instruction operate on data memory. This is as opposed to an algorithm that makes a lot of subroutine calls (using more stack memory), or that allocates a lot of memory at runtime (using more heap memory). As we could intuitively see from high-level code, the GFM algorithm routine is simplistic, does not contain a lot of sub-routine calls or memory allocation, and the profile results confirm this.

The branch instruction profile further confirms the intuition that because of the large amount of time spent in loops, there is a high percentage of conditional branches executed. This observation allows us to notice that providing loop support in hardware could offer more efficient operation. Providing hardware loop support would not only reduce the number of conditional branches needed in software, but also the corresponding address calculation, thereby reducing integer computation as well.

Another very important observation made from profiling results is that the Simplescalar simulator contains a much higher number of instructions than are listed in the instruction profile. What that means is that all the complexity within the processor core design is simply wasted space, and possibly wasted energy if those circuits are not fully off or in a low-power state. An application-specific core design should provide only instructions explicitly needed for the GFM algorithm to run, in order to reduce energy and improve performance. In general, performance can be increased when complexity is removed because delay time within the logic circuits can possibly be decreased. In other words, in general, the less complex the processor core is, the faster it can operate, and so a customized core design should be a simple as possible, including only those instructions, addressing modes, branch types, and memory structures needed for the application. This information was later used to define a new instruction set and processor core architecture for the GFM algorithm.

Floating-Point to Fixed-Point Conversion Results

As was mentioned previously, conversion of an algorithm from floating-point to fixed point can allow the algorithm to be implemented using simpler, lower power and lower cost hardware, and to run more efficiently and in less time. If the conversion to fixed-point can be done such that the resulting (usually reduced) accuracy is sufficient, this technique can offer a huge increase in performance and efficiency simply by redesigning the software implementation. Of course, when converting to fixed-point, there is an inherent change being made to the target hardware design, but as part of a hw/sw co-design flow, hardware re-design is expected anyway.

TABLE II: Fixed- vs. Floating-Point Speedup

Processor	Floating-Point Cycles/Step	Fixed-Point Cycles / Step	Speedup %
MSP430	227000	1871129	--
Simplescalar	--	19711	
ARM7TDMI	45007	15272	66
ARM9TDMI	43071	13133	70
ARM10E	41271	13133	68
XScale	41271	13133	68
CortexM1	80496	33856	58
ARM1136J	45100	13133	71

Table II: Speedup resulting from floating-point to fixed-point conversion. ARM targets profiled on IAR compiler with full optimizations for speed.

This section presents the results from converting the GFM algorithm code to fixed-point. The results in Table II are in the form of cycle counts. As was explained in Chapter 1, the fixed-point format used in this study was $Q_2(18.14)$. The table shows the cycles counts per step for the algorithm run in its original floating-point implementation, and in the new fixed-point implementation. This was done on 7 different architectures, and the corresponding speedup resulting from the conversion is given in the last column.

The results are quite impressive, with speedup ranging from 58% to 71%. Note that the MSP430 is presented there only for reference. Since it is a 16-bit machine, in fact the 32-bit fixed-point version is worse because it cannot do 32-bit integer operations at the hardware level. On the other hand, it is interesting to see how high a cycle count is exhibited on that device for the floating-point version compared to other architectures. Note that none of the devices presented here have floating-point hardware like Simplescalar, instead they use emulated floating point software routines to compute floating-point results. The fixed-point implementation cycle count is listed for Simplescalar, but not along with the corresponding floating-point count. This is because although Simplescalar can simulate floating-point hardware, the compiler ported for it cannot compile using emulated (software) floating point. Therefore, the Simplescalar floating-point cycle count cannot be compared to the Simplescalar fixed-point cycle count since they represent different hardware. All ARM cores are 32-bit, fixed-point hardware machines.

The results are impressive, but it also has to be shown then that the fixed-point version of the algorithm is in fact valid. To show that the fixed-point version performs accurately enough for the application, and that the conversion was a success, presented in Table III are results from running the standard dive set. The results are given in terms of the main GFM output, the free gas volume. The FGV values for a specific dive profile are the most important outputs, and FGVs obtained using a fixed-point version of the algorithm need to match results from the floating-point version. Table III shows the results for dives with a bottom depth of 180ft up to 60 ft, with and without safety stops (SS), and out to the NO-D limit, and ½ the NO-D limit. You can see which dives were run out to the NO-D limit by noticing which dives reached the critical FGV, 40ml, and those that were considerably less. You can also see the effect of taking a safety stop - as expected the free gas is less when the stop is taken.

TABLE III: Fixed-Point Accuracy, FO2=21%

		Floating-Point			Fixed-Point		
Depth	**SS**	**Time (min)**	**# Steps**	**Max FGV**	**Time (min)**	**# Steps**	**Max FGV**
180	N	6.6	766129	40.2	6.7	765940	40.3
180	N	4.0	697352	23.4	4.0	697112	22.8
180	Y	6.6	842989	9.2	6.7	842590	9.0
180	Y	4.0	779294	7.8	4.0	778854	7.2
160	N	7.9	789807	40.1	8.1	798137	40.3
160	N	4.7	705146	23.3	4.7	704990	22.8
160	Y	7.9	866457	9.2	8.1	874997	9.4
160	Y	4.7	787088	6.8	4.7	786732	6.6
140	N	9.8	834890	40.1	10.3	845394	40.3
140	N	5.3	712945	21.7	5.3	712807	21.3
140	Y	9.8	911750	9.6	10.3	922044	9.7
140	Y	5.3	794687	5.4	5.3	794349	4.9
120	N	13.5	921089	40.2	13.8	925764	40.0
120	N	7.0	749552	22.3	7.0	749132	22.1
120	Y	13.5	997949	10.6	13.8	1002414	10.4
120	Y	7.0	831494	7.4	7.0	830874	7.3
100	N	19.5	1066685	40.1	20.0	1083677	40.1
100	N	9.7	809116	21.0	10.7	835550	22.1
100	Y	19.5	1143335	12.0	20.0	1160537	12.0
100	Y	9.7	891058	5.8	10.7	917292	6.4
80	N	30.3	1343695	40.1	30.9	1355036	40.0
80	N	15.3	946925	19.8	15.3	946847	18.7
80	Y	30.3	1420555	14.9	30.9	1431686	14.4
80	Y	15.3	1028667	5.2	15.3	1028389	4.8
60	N	52.4	1901589	40.0	53.3	1919344	40.0
60	N	26.0	1212749	15.8	27.0	1239632	15.9
60	Y	52.4	1978449	19.4	53.3	1995994	18.7
60	Y	26.0	1294491	3.9	27.0	1321374	5.9

Table III: Standard dive set run in both floating-point and fixed-point algorithm implementation for FO2=21%

Note that as was explained in Chapter 1, all FGV values are maximum peak values found after the simulated diver surfaces. It is that peak FGV value that is used to determine probability of DCS, and so it is vital for that value to be correct. All dives in Table III were run using air (FO2=21%), but that the results also are representative of results for other mixtures as well.

It is also important to point out that the dives that include safety stops were taken to the NO-D limit or ½ the NO-D limit as calculated without considering the safety stop, then the safety stop was taken subsequently. This is as opposed to calculating new NO-D limits that consider the safety stops in the calculation, which would result in longer times. For example, for the 100ft dive without a safety stop, the NO-D limit is 19.5, and in the table another dive is shown with a safety stop for the same bottom time. This would be confusing to those who understand the concepts of free gas and how GFM works, except that you need to notice that the resulting FGV is much lower. If the dive with a safety stop were taken out to a new NO-D limit that was calculated in such a way that included the safety stop, the resulting FGV would of course also be roughly 40 ml, and the NO-D time would be longer.

Comparing the results between fixed- and floating-point, we start with the dive times. Finding the NO-D limit the case of simulating these standard sets amounted to staying at the current depth until the immediate ascent prediction gave a FGV value above Vcrit (40ml). For example, For the 100ft dive without a safety stop, the NO-D limit was 19.5 minutes for the floating-point version, and 20.0 minutes for the fixed-point version. As we know, some difference is expected due to the nature of how the fixed-point implementation approximates a much higher resolution numeric format. Still the difference is 1 minute or less, and no more than 1.1 ml in all cases except the last dive, where the difference was 1 minute and 2ml. Additional accuracy can be found by adjusting the scale factors in the fixed-point program code, but for a typical dive computer application, these results are acceptable.

The number of steps required is large because it includes all the hundreds and even thousands of prediction calculations (as described in Chapter 1) performed at each time-step. When the dives times between floating- and fixed-point versions differ by even a small amount of time, there are more or less predictions introduced, and therefore more or less time-steps needed. Even so, over all dives in Table III, the difference in the number of time-steps run is less than 4%. Again, this is entirely acceptable since times and FGVs are what need to be accurate, not necessarily the number of time-steps.

The summary for this section is that tables II and III show that the fixed-point algorithm implementation is valid and accurate, and as we know - much more energy efficient than the floating-point version. If a product designer, for example, is willing to live with the 1ml difference in FGV from the fixed-point version algorithm in order to get 66% more performance on an ARM7TDMI target for example, then this fixed-point implementation is a great success. If a designer needs a much more accurate result, there are many options. The fixed-point version could use a longer format, for example 40 bits or more instead of the 32 bit format used here. One could also scale values much more aggressively to squeeze out as much accuracy as possible. The issue with more aggressive scaling is that as mentioned previously, each time you introduce a new scaling operation you introduce more operations, and so as always there must exist a trade-off between performance and accuracy.

In this study, we will accept the results as shown in Table II and Table III, and continue forward to further customize by designing now a customized processor core for this new fixed-point GFM implementation. By doing this, the goal is to combine the efficiency found from the fixed-point conversion with more efficiency introduced by customizing hardware, for a much higher overall efficiency for the system. In the next section we begin the process of designing a custom core by using the information from this and previous sections to define a design a new instruction set architecture.

Low-Level Application Profiling (Fixed-Point)

Since we now want to use a fixed-point implementation of the algorithm, we need to again profile that implementation. This was done again on the Simplescalar simulator and results are presented in figures 31-35. This configuration yielded a CPI of 0.54 and a cycle count of 19711 cycles per time-step. Note that we expect the time-step to be more than the first Simplescalar configuration because in this configuration we are no longer using floating-point hardware.

It is interesting to note that as shown in Figure 31, the instruction class profile is completely different from the floating-point hardware version. Since there are no floating-point instructions used, we now see 62% of instructions are integer computations (compared to 39%), about 13% are branches (compared to 9%), and load and store operations make up about 25% of instructions (compared to 29%). More integer computation is of course expected, and increased numbers in other categories is as well, since the floating-point instructions are removed.

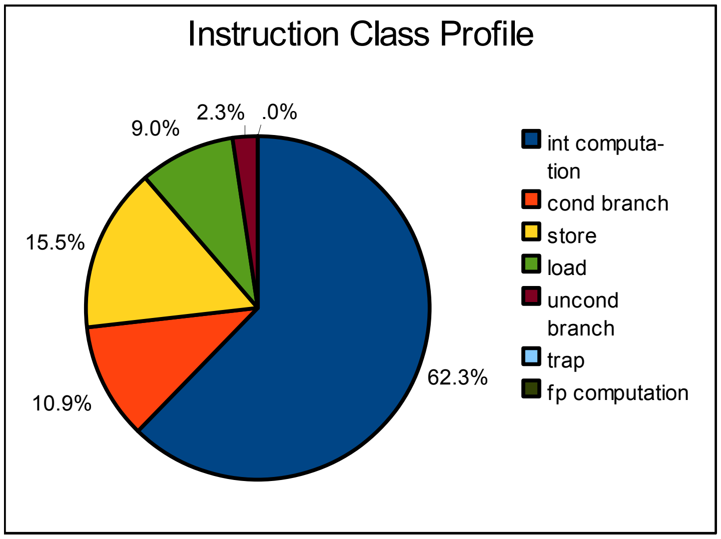


Figure 31: Instruction class profile from Simplescalar profiling results

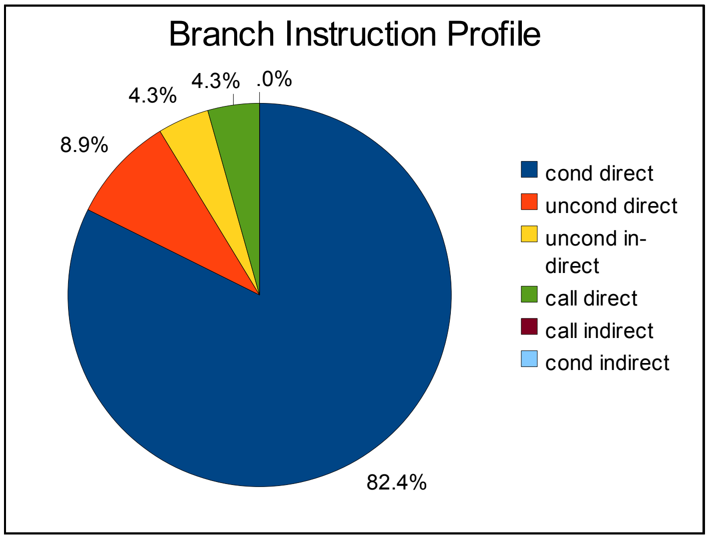


Figure 32: Branch Instruction profile from Simplescalar profiling results

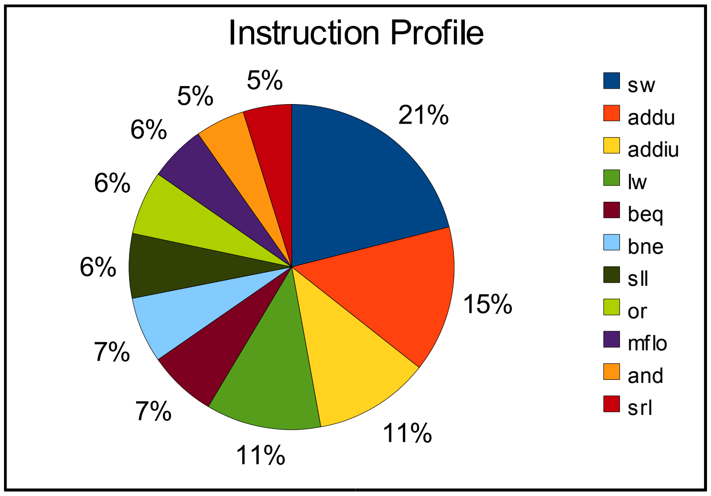


Figure 33: Instruction profile from Simplescalar profiling results

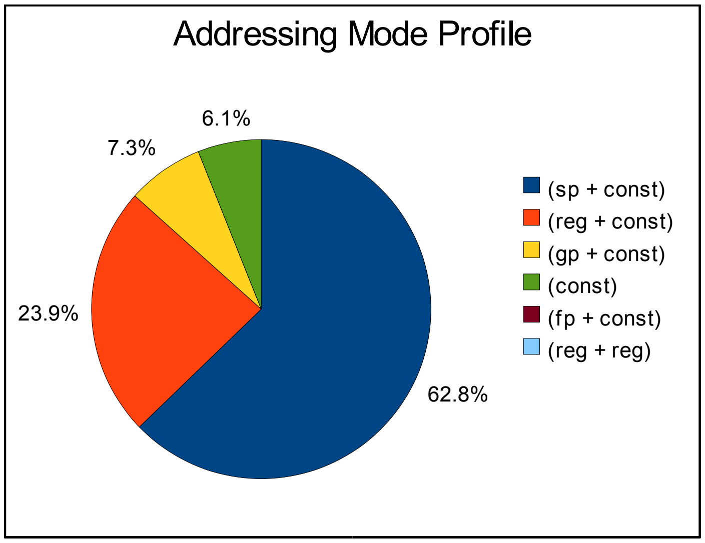


Figure 34: Addressing mode profile from Simplescalar profiling results

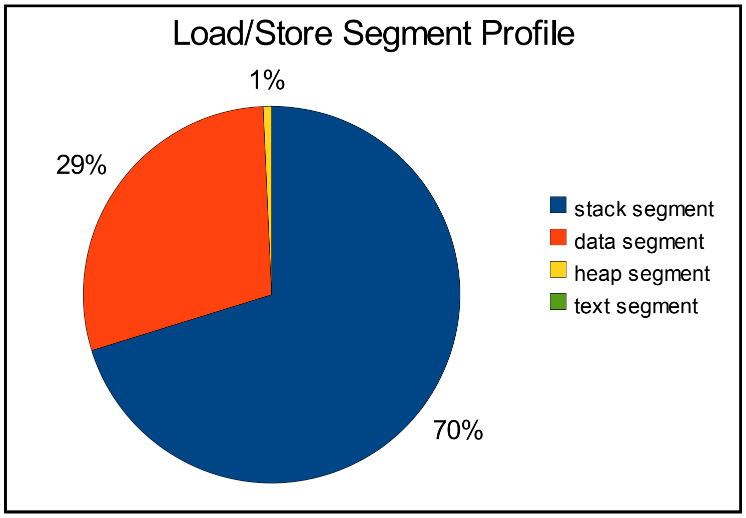


Figure 35: LD/ST address segment profile profiling results from Simplescalar

The branch instruction profile is very similar, however, the addressing mode, instruction, and load/store instruction profiles are different. If we compare the instruction profile in Figure 28 to Figure 33, we see that in the fixed-point version, of course all the single-precision floating-point instructions are missing (such as mul.s, sub.s, add.s), and the percentage of integer addition instructions is higher. The fixed point version code also uses many more addressing modes (compare Figures 29 and 34). It is not exactly clear why this is, but we can say that because the implementation is different, there may have been optimizations that could be performed more easily by the compiler using more addressing modes with the fixed-point implementation.

Finally, we see that the load/store profile is different as well (Figures 30 and 35). In the floating-point version there are almost 100% of memory accesses use the data segment, and virtually no stack usage, whereas in the fixed-point version, almost 70% of memory usage utilizes the stack and the remaining usage is mostly from the data segment. This most likely has to do with the fact that the fixed-point version code needs to call fixed point routines to do arithmetic, whereas the floating-point version code does not. This is expected and gives insight into the memory structure design that will be explained in a following section.

What is also interesting is the specific instruction frequency plot given in Figure 36. This is a plot of which instructions executed how many times per step, averaged over many time-steps. Very interesting is that for the most part, instructions are evenly distributed, but at one point there is a spike. This is an indicator that some instructions execute much more often than others, and that there may be an opportunity for acceleration in hardware. In fact upon inspection at the corresponding addresses in the Simplescalar PISA assembly code, it was found that the spike in Figure 36 in fact corresponds to the integer division routine, and that these 49 instructions represented less than 2% of the code, but the program spent 27% of the time executing those instructions (as a group). In the next section we will see how implementing a hardware integer division module further accelerated the code.

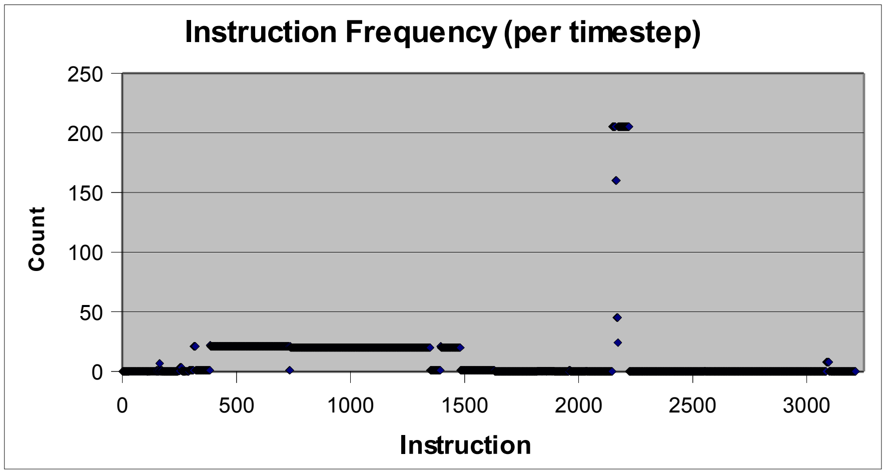


Figure 36: Specific instruction frequency per time-step

Instruction Set Architecture Design

When the GFM algorithm is run on a 16-bit, non-pipelined, in-order, mostly RISC instruction set microprocessor, it requires 227,000 cycles per step when running the floating-point version of GFM. With am ARM7 core that has a 3-stage pipeline, we expect at least a reduction to at least 1/3 of that number, and in fact it is less, at about 45,000 cycles per step. When converted to a fixed-point implementation, we see additional efficiency where the cycle count is about 20,000 cycles per step for Simplescalar, and roughly 15,000 cycles for an ARM7 core. Using this information, and the profiling information from the previous sections, we now can make some decisions on what a custom architecture should include, with the first step being defining the instruction set architecture.

We know that a pipelined architecture is favorable, since even with only a 3-stage architecture in the ARM7 we see a factor of 5 decrease in the number of cycles. Of course we can also reach this conclusion simply by referring back to Chapter 1 and taking note of how computationally intensive and repetitive the algorithm is. What that points to is the need for some sort of mechanism to provide more parallelism, and a pipelined architecture is a good choice. Now, given that we are going to choose a pipelined processor, we then would want to use a RISC-like ISA, since RISC type ISAs lend themselves well to pipelining given the simple nature of the instructions. How many stages in the pipeline are needed will become clear in the next section, but for now we will focus on instructions.

Basic Instructions

We will be using a fixed-point implementation based on the results presented in the previous sections. On a basic level we will certainly need arithmetic operations like add, subtract, multiply and divide. These will be based on 32-bit instructions, 32-bit data, and 32, 32-bit registers. Arithmetic operations will only be able to operate on register data, so data need to be loaded into general-purpose registers first. In fact the arithmetic operations in this design were combined into one instruction called ALU, and the specific operation will be designated by an operation code.

We will use a load-store architecture, so we will need load and store operations. Because we know the GFM algorithm runs a lot of operations from within loops, and operates on array data, we will include indexed load and store instructions, as well as explicitly addressed. In fact, since there is a high amount of load and store operations, we will use load and store operations that can load and store 2 data values at a time. These operations are called LOAD2 and STORE2, and LDIX2 and STRX2 for indexed addressing versions.

Loop Instructions

Given the large amount of processing done in loops, the design presented here includes hardware loop support. This means that within the instruction set, there are instructions to set a loop counter, and perform a combined test and branch operation. These instructions are called LDINDX and WHILE. LDINDX (“load index”) is used to load the start and end index of a loop counter, and “while” is used to test the loop counter, increment it, and branch if needed. The LDLPCTR instruction is used to load the loop counter into a register, for use in the program. The WHILE instruction is used to convert C-code for loops to a “do-while” type structure in assembly. All loops in the implementation are written in effect as do-while loops. As we will see in the next section, these instructions correspond to custom hardware loop registers, but these are unaccessible otherwise except via these instructions.

Control Instructions

In addition to hardware loop support, a special c-code like if statement is implemented. This instruction is called IF_CMP, and works by specifying two registers to compare, the operation used to compare, and an offset used for calculating the branch target if the condition is true. The condition is tested in hardware automatically, without explicit operations such as separate compare and then branch-if-equal (or similar) instructions. Although the basic instructions are RISC-based, these control instructions, along with the loop instructions make the design more like a CISC machine. In this way this design is more of a hybrid architecture than purely RISC or CISC. There is also a JUMP instruction to explicitly jump to some address, for example for use in an if-else type structure.

Special Instructions

In addition to the above instruction types, there are a few special instructions. One called RTorPRED, tells the processor which memory space will be operated on. What this refers to is that when we need to run predictions, as was explained in Chapter 1, memory from the real-time state variables needs to be copied, and the time-steps then need to be run on the copied values, so as not to disrupt the real-time state. What the instruction actually does will be explained in the next section, but for now we can say that it tells the memory system which memory space is being used. In addition we have a related instruction called MEM_COPY. This instruction tells the hardware to start copying real-time data to the prediction memory space.

Instruction Set Details

Table IV gives details of the custom instructions created for the custom GFM processor. Note that there are 4 different instruction formats. The number of instruction formats corresponds to the complexity of the instruction decoding scheme, so in general it is desired to reduce the number of type of instructions. The first type (Type 1) used by ALU, and RTorPRED, uses an opcode, then 9 bits unused, followed by 2 5-bit source register addresses in the case of ALU (or unused in the other instructions), a 5-bit destination register address (or unused), followed by 4 its used differently for each instruction.

TABLE IV: Custom 32-Bit Instruction Set

OpCode (4)			**Detail (28)**		
			Type 1		
ALU	N/U(9)	RS1(5)	RS2(5)	RD(5)	OP(3),N/U(1)
RTorPRED	N/U(9)	N/U(5)	N/U(5)	N/U(5)	N/U(2), RTorPRED(2)
			Type 2		
LOAD2	ADDR1(9)	RD1(5)	ADDR2(9)	RD2(5)	
STORE2	ADDR1(9)	RD1(5)	ADDR2(9)	RD2(5)	
			Type 3		
JUMP	OFF(15)	N/U(5)	N/U(5)	N/U(3)	
IF_CMP	OFF(15)	R1(5)	R2(5)	OP(3)	
LDLPCTR	N/U(15)	RD(5)	N/U(1),N/U(4)	N/U(3)	
LDINDX	LPSTRT(8), N/U(7)	LPMAX(8)	N/U(1)	N/U(4)	
WHILE	OFF(15)	LPMAX(8)	N/U(1)	N/U(4)	
			Type 4		
LDIX2(4)	ADDR1(9)	ADDR2(9)	RD1(5)	N/U(2)	ONEORTWO(3)
STRX2(4)	ADDR1(9)	ADDR2(9)	RD1(5)	N/U(2)	ONEORTWO(3)
LDIX1m1(4)	ADDR1(9)	N/U(9)	RD1(5)	N/U(2)	N/U(3)

Table IV: custom 32-bit instruction set for customized processor core

ALU uses the last 4 bits to include the specific ALU operation code (indicating either add, sub, mul, or div), and RTorPRED uses the bits to indicate which memory space to use (real-time or prediction).

The second category (Type 2) is the load/store group including instructions LOAD2 and STORE2. The format is simple and includes two sets of addresses/register pairs. For LOAD2, the addressed are the addressed in memory from where data is read, and the register addresses are where they should be written. For STORE2, the addresses are where the data read from the registers should be written to. Note that the addresses are 9-bits, so this limits the memory space to 512

32-bit values, and the registers are limited to 32 addresses. Considering the prediction memory space as well, this amounts to a maximum of 4KB of memory that can be addresses using these instruction formats.

Type 3 instructions are control and loop instructions. JUMP uses a 15-bit offset where the first bit is used to indicate whether the jump is forward or backward. This amounts to allowing up to a jump of 16KB in either positive or negative direction. Considering that we have 32-bit instructions, this allows for jumps to branch forward or backward no more than 4096 instructions. In fact the total number of custom assembly code instructions for the GFM routine is only about 320 so this is more than enough. The IF_CMP instruction also uses this same type of jumping mechanism, except that it also includes the addresses of two registers to compare, and an operation code for what type of comparison is to be performed. There are 4 types implemented, if-less-than, if-greater-than, if-equal-to, and if-not-equal-to. If the register values being compared satisfy the condition indicated, the jump is taken corresponding to the 14-bit offset specified.

LDINDX is used at the start of a for-loop type c-code structure, in order to load the start and end loop indices. LPSTRT and LPMAX are the starting and ending indices, respectively. Most of the overhead associated with loops is removed due to the internal loop registers and automatic loop increment and test. As was mentioned previously, in this way these instructions allow a c-code style for-loop to be de-constructed and re-written as a do-while structure. For example, the GFM algorithm contains a few loops in the form below:

```
for(int nx = 0; nx < NUM_CELLS; nx++)
{
     // ...
}
```

This structure is re-written in the custom assembly as follows:

```
# prepare index for loop
OP_LDINDX 0 21 LOOP1
LP1: OP_NOP
OP_WHILE LP1 21 LOOP1
```

Note the use of the “NOP” instruction. The “NOP” is a commonly used instruction to indicate “no operation”, and the hardware simply allows the processor to run but without doing any specific operation. These can be used to insert so-called stalls in the pipeline where necessary. In the case above, it is simply used as a placeholder for the loop. Note that the label “LP1” is the branch target for the WHILE instruction. The LDINDX instruction loads the loop index once, and the WHILE instruction loops back to the label LP1 on each iteration. When the loop is complete, the condition tested by WHILE does not result in a branch, and the next instruction after WHILE is loaded. Typically the for-loop structure written in assembly includes first loading the initial loop counter value, then a comparison operation is done after loading the current loop counter value and the reference value, then a branch is either taken or not taken. This customization allows for hardware to perform the comparison immediately when the WHILE instruction is loaded, and the result is processed in the first stage, thereby reducing the number of operations needed for the loop comparison to 1 , as well as the number of stages needed to 1. The result is that the loop condition is tested and processed within 1 cycle, as opposed to multiple cycles for a standard loop compare and branch operation. Hardware for 2 levels of loops was initially supported, although in this custom implementation of GFM only 1 is used, since inner loops are optimized by a compilation technique called unrolling, used to increase efficiency in the pipeline, and to more aggressively parallelize operations.

Type 4 instructions include indexed instructions. The index used to load or store values is in fact the loop counter, and this optimization has to do with the fact that GFM processes a lot of values from within c-code arrays. Arrays inherently use indexed values since arrays are simply a collection of data values in memory that are contiguous. For LDIX2 and STRX2, the addresses needed to load or store 2 values are specified, as well as the loop number (referring to the inner or outer loop), and the number of values to load or store (one or two). If one value is specified, the second set of data is simply ignored. One optimization made here is that only one register is specified, and the second register address required (for a double load/store operation) is assumed to be the next register address. This is useful for example when loading or storing two consecutive array values. This also depends on how the assembly code is written, but happens enough in GFM that this optimization is useful. A specific example is given in Figure 37, where the original c-code is given first, and the resulting assembly code follows. Note that registers R4 and the inferred register R5 are used to load and store values. The structure shown allows the loop to load, store, then compare and branch in only 3 cycles, although when pipelined the operations take an equivalent of less than 1 cycle per loop because of the way the pipeline overlaps instructions between loop iterations and pipeline stages. The LDIXm1 instruction was an optimization added that allows you to load a value not base on the loop counter value, but based on that value minus 1. This is present in the c-code implementation of GFM, and because the loop counter is held in a special register, it was simple enough to add another register that always holds 1 less than the loop counter for the special load instruction. This reduces the number of cycles needed to load and then decrement the loop counter value before using it.

```
// c-code
for(int nx = 0; nx < NUM_CELLS; nx++)
{
     poldold[nx] = pold[nx];
     pold[nx] = p[nx];
}

# custom assembly code
OP_LDINDX 0 21 LOOP1
LP1: OP_LDIX2 pold p R4 [i] 2
OP_STRX2 poldold pold R4 [i] 2
OP_WHILE LP1 21 LOOP1
```

Figure 37: Conversion from c-code for-loop to custom assembly

Processor Core Architecture Design

The ISA design given in the last section of course assumes or infers something about the physical architecture, and the architecture has to assume something about the ISA. The concept for the physical core was to provide a 5-stage pipeline designed for instruction level parallelism (ILP), but by reducing the number of instructions and providing customized instructions for one specific application to reduce complexity in hardware. If we simply used a 5-stage pipeline machine like MIPS, there would be a lot of overhead designed into the system that is unneeded when running the GFM algorithm. Choices to reduce complexity are mainly to reduce area and power in the resulting design. Having more instructions, more addressing modes, and more complicated structures built in to the system means more circuits are drawing power from the supply (unless specifically turned off), and possibly longer transport times as signals go through the system from input to output.

The basic pipeline described in this section is based on the basic MIPS design, where the first stage is instruction fetch, followed by instruction decode, followed by execution and address calculation, followed by memory access, and finally write back. The difference of course, between this design and MIPS, is that instructions have been customized, load and store operations can load and store 2 values at a time, and where some instructions are more CISC like than RISC. The customized loop and control instructions are not present in MIPS, and RTorPRED is customized for this design. In addition, to achieve a higher performance, the integer division operation is handled in hardware rather than in software.

Architecture

Starting with the top-most level view of the device, the GFM custom processor looks like as shown in Figure 38. There are 9 pins to interface to the device. Because the device was designed and simulated using Xilinx FPGA tools, and because the device chosen requires a clock input of greater than 2 MHz, the signal CLOCK is used to input a clock signal to the device. The ENABLE signal is a global enable, used to enable or disable major blocks within the device, in order to conserve power. The signals SS, MOSI, MISO, and SCLK are used to communicate with the device using an SPI module interface, designed into the device. DCMLOCK is a signal used to indicate when the internal clock is stabilized. The internal digital clock module (DCM) takes the input clock signal CLOCK and divides it for a faster clock. The input signal STEP is used to tell the device to start another GFM time-step computation, and the RDY signal indicates that the computation is complete and ready to compute again.

Figure 39 shows the next inner level device schematic. Note the external signals connecting to 3 main components. The top-most component is the DCM. The bottom left is the SPI and the bottom right is called the datapath. The datapath includes all the memory, ALU, and control components and interface signals that make up most of the system. The SPI has a few lines connecting to the datapath because the SPI interfaces with system memory. The SPI is very simple and basically contains one command, which allows the outside world to input pressure and time data to the GFM algorithm for processing. The datapath is much more complex, and will explained further in the following section. Note that figure 38 is analogous to how the custom GFM processor would look as a physical custom IC.

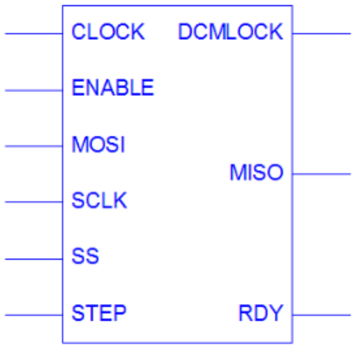


Figure 38: Top level device schematic

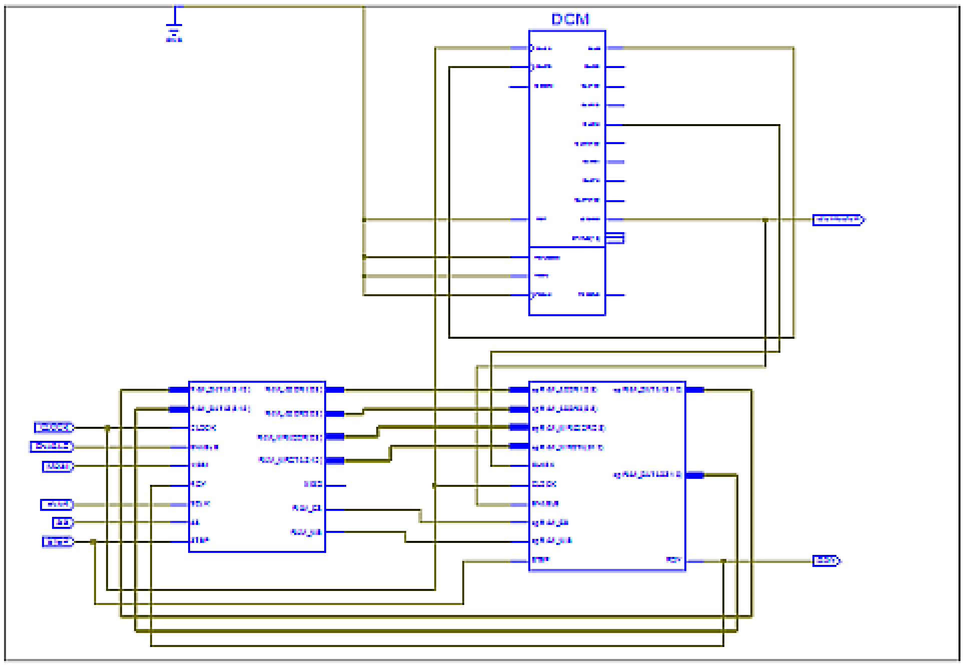


Figure 39: Second level device schematic

The Datapath

Figure 4 shows a conceptual rendering of the custom processor datapath. Notice the pipeline stages and related pipeline registers S2, S3, S4, and S5. In the bottom left corner you will notice the hardware loop support registers and related adders and subtractors. In the upper left corner you see the program counter (PC), and related MUX and adders. Note how most data flows from left to right, but that some is forwarded back through the pipeline from stages 4-5 to stages 3, 4, and 5. As was mentioned previously, this is a technique used to avoid data and control hazards.

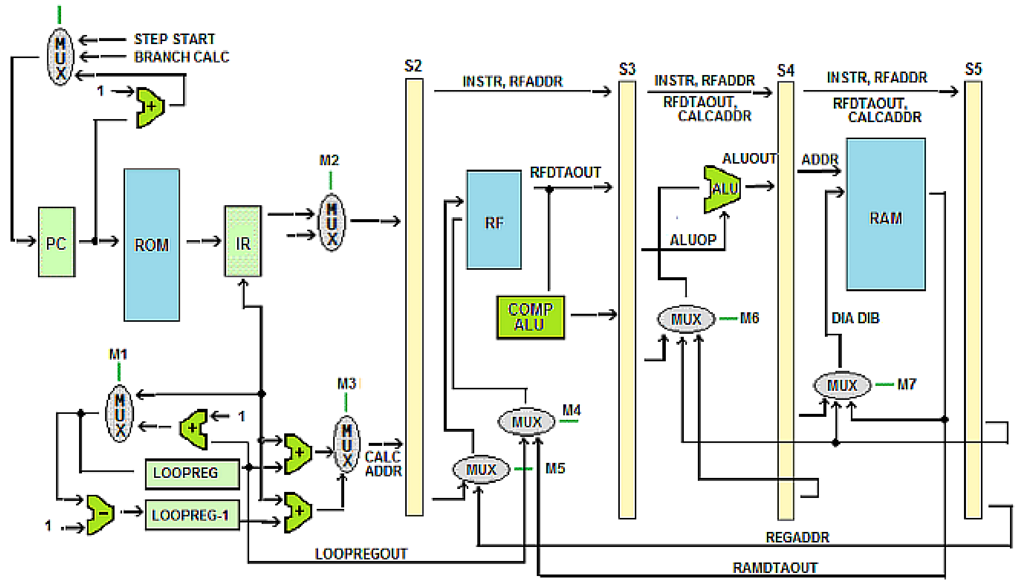


Figure 40: Custom GFM Processor Datapath

The ALU in the 3rd stage is shown as a simple ALU, but note that it includes a hardware division module that can stall the entire pipeline while in progress. MUX M2 is used to select between sending the next instruction, or a NOP instruction to stall the pipeline. Notice that the ALU, RAM, and register file (RF) all require MUXes to select between input data that come from either pipeline registers (originally from data read or calculated in earlier stages), forwarded from later pipeline stages, or sent from special calculation ALUs (like loop register calculations). In this pipeline design, the entire instruction is forwarded through each stage, starting from stage 1 when it is read from ROM. Note that the instruction registers (IR) acts as the stage 1 pipeline register.

Note that there are some components not shown in the figure. For example, the controller that handles all the MUXes, memory enable and read/write signals, and other logic associated with the datapath. The controller can access all components including the pipeline registers, ALU, memory, and MUX select signals. The controller is a very important portion of the design to point out because it implements the control corresponding to the custom ISA. As was mentioned previously, there is some logic associated with the division module and stalling the pipeline. In this case, there is a signal called FREEZE, as opposed to stall, named so because when a division operation is detected and begins, the entire pipeline is frozen until the operation is complete.

Memory

The memory system is relatively simple. There is one memory used for program code (ROM), and one for random access memory (RAM). The RAM memory is special in that it is split between real-time and prediction memory from with the same block memory. The real-time memory is the memory used in calculation of the real-time state of the algorithm, and the prediction memory is used to perform the same operations, but on a copy of the real-time state variables in order to predict future values of the real-time variables. Since the design is pipelined, the memory copy can be done in software efficiently by loading and storing pairs of data.

The ALU

In implementations like those using ARM processors, MSP430, and many other fixed-point hardware processors, division is handled in software. Whereas multiplication, addition, and subtraction are handled with simple fixed-point hardware, division is not as simple, and requires software routines to carry out the operation. For example, given the fixed-point c-code implementation of GFM, when compiling the program using an IAR commercial strength compiler using an ARM7TDMI as the target, the fixed-point division routine requires roughly 80 cycles to complete each division operation, averaged over 100 calls to the division routine. This is with some optimizations turned off so that the compiler does not remove the arbitrary operation done simply to profile the operation. For this reason, in this design the ALU was designed so that division is handled in hardware, with a minimal number of cycles. There exist many binary division algorithms, and the Wikipedia article on different methods is a good place to start. The implementation here uses the simplistic and intuitive, iterative, shift-and-subtract method as given in Figure 32 as a c-code implementation, taken from [42].

Note that this implementation is given only as a conceptual implementation, since, this design implements this algorithm in hardware. The assembly programmer using this design needs only to specify a division ALU operation, and the hardware with iterate through the steps shown to arrive at the result. Note further that the method requires 32 iterations of the loop, as well as some additional operations to deal with the results. Extra operations are needed since this routine handles unsigned division, but we use it for signed, fixed-point division for $Q_2(18.14)$ formatted numbers. What happens is, when a division ALU operation is detected, the operands are loaded into the appropriate ALU registers, and the processor ticks along until the operation completes, halting the pipeline during the computation, and outputting the result using the standard ALU output upon completion.

Handling the division operation in hardware reduces the reported near 320 to 385 basic RISC instructions, to one, and with a 5-stage pipeline, that amounts to roughly 60 to 80 effective cycles reduced down to around 35. This is a huge reduction considering how many division operations are performed over the course of a standard dive, and over the life of a dive computer product. Energy is also reduced of course, since less cycles means less time the processor is running and thus less energy dissipated.

```
unsigned divlu(unsigned x, unsigned y, unsigned z)
{
     // divides unsigned (x || y) by z
     int I;
     unsigned t;
     for(i = 1; i <= 32; i++)
     {
          t = (int)x >> 31;           // all 1's if x(31) = 1
          x = (x << 1) | (y >> 31);  // shift (x || y) left
          y = y << 1;                 // one bit
          if((x | t) >= z)
          {
               x = x - z;
               y = y + 1;
          }
     }
     return y;                        // remainder is x
}
```

Figure 41: Conceptual division algorithm written in c-code

<u>The Serial Peripheral Interface</u>

The SPI is a simple interface designed to allow data to be input and output from the custom processor from the main system processor. The data format is most significant bit first, where the first byte is the number of bytes to be sent. Data is clocked into the SPI receive buffer on each rising clock edge, when the slave-select pin in low (SS=0). When SS=1, the byte is complete, and this is repeated until all bytes are received (when the number of bytes received equals the value of the data in the first byte). There is one command implemented currently which is the write data command. The master processor uses this command to send the pressure and time data to the slave GFM processor. The number of bytes is sent first (8), then the command (0x00), then 4 bytes for the 32-bit value to be written and finally 2 bytes for the address to write to. After the data is written to the slave processor, the master raises the STEP line to start a GFM time-step calculation.

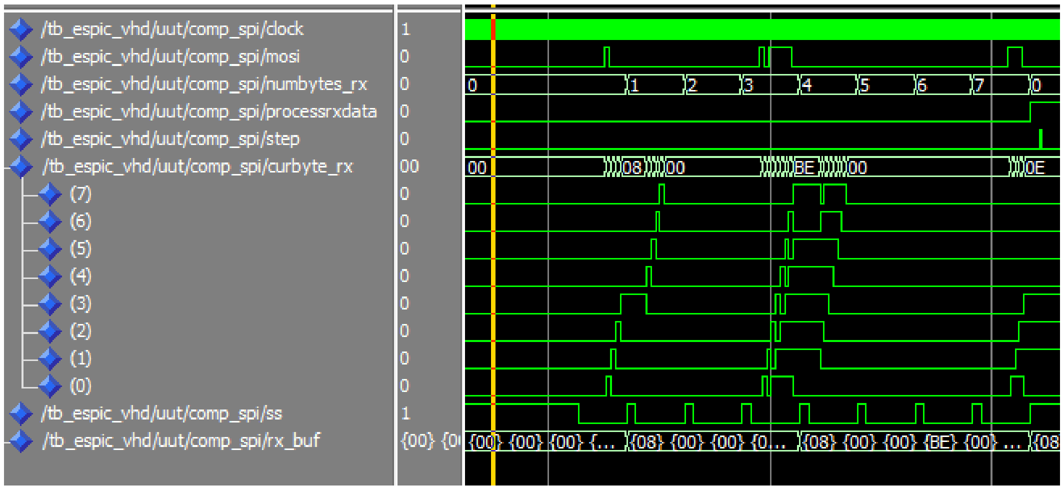

/tb_espic_vhd/uut/comp_spi/clock
/tb_espic_vhd/uut/comp_spi/mosi
/tb_espic_vhd/uut/comp_spi/numbytes_rx
/tb_espic_vhd/uut/comp_spi/processrxdata
/tb_espic_vhd/uut/comp_spi/step
/tb_espic_vhd/uut/comp_spi/curbyte_rx
(7)
(6)
(5)
(4)
(3)
(2)
(1)
(0)
/tb_espic_vhd/uut/comp_spi/ss
/tb_espic_vhd/uut/comp_spi/rx_buf
08
00
BE
0E

Figure 42: SPI Data Transfer

Figure 42 shows 8 bytes being received by the SPI module. Notice how the data is shifted into the “curbyte_rx” variable, representing the current byte being received. Also notice the variable “numbytes_rx” as it counts up on each byte received. Lastly, notice how the STEP signal is raised after the data is sent. This starts the GFM time-step routine immediately after receiving the new pressure data. Currently you cannot read out from the device to the master, but the implementation would be similar.

Implementation Characteristics

The custom processor architecture described in previous sections was implemented in VHDL for a Xilinx FPGA target. The specific device used was the 3s200vq100-4, chosen for its gate capacity and built-in block memory. The implementation synthesis results are given in Figure 43, and the maximum frequency allowed was reported as 22.5MHz. The cycle count per step for the custom processor ranges from 4949 cycles for the non-gas forming portion of a dive to 6421 for the gas forming portion of a dive. On a particular dive to 180ft for 3 minutes, the average cycle count was 5175. Recall that the exact cycle count depends on what phase the program is in and so this has to be taken into consideration. Also note that synthesis results depend on the specific design goals chosen in the synthesis tool, and it is noted here that the design was simulated with a design goal of optimizing for area and minimum power rather than for speed. In fact a maximum frequency of 22.5MHz is perfectly suitable for a low-power application such as a dive computer, as long as the performance, power, and area constraints are met.

```
Selected Device : 3s200vq100-4

 Number of Slices:                      1059  out of   1920    55%
 Number of Slice Flip Flops:            1277  out of   3840    33%
 Number of 4 input LUTs:                1846  out of   3840    48%
 Number used as logic:                  1756
 Number used as Shift registers:          90
 Number of IOs:                            9
 Number of bonded IOBs:                    6  out of     63     9%
 IOB Flip Flops:                           1
 Number of BRAMs:                          4  out of     12    33%
 Number of MULT18X18s:                     4  out of     12    33%
 Number of GCLKs:                          5  out of      8    62%
 Number of DCMs:                           1  out of      4    25%
```

Figure 43: Synthesis Results for Custom Processor on Xilinx 3s200vq100-4 Device

Custom Tools (The Assembler, and Manual Compilation)

It should be mentioned that throughout this project as part of the hardware/software design flow, custom tools needed to be developed in order to complete the design. This section will describe the custom assembler used to implement the custom processor program code. The assembler is used to convert assembly code to machine code. The machine code output is in the form of 32-bit words used to initialize the ROM of the FPGA target for program code. Since there is no compiler for the custom processor, all high-level c-code for the custom implementation was hand-assembled into custom assembly, and the assembler was used to convert to machine code for use by the processor.

The assembler is a simple program, written in Java, that essentially parses the assembly program and converts the instructions into machine code. There are a few components needed by the assembler. First, there is an input file called a symbol table, that contains the names of all variables in memory, along with their corresponding number of 32-bit words each variable takes up in memory. The order of the entries in the symbol table is important because they are assigned consecutive addresses by the assembler, starting with address zero. The assembler first reads the symbol table, and stores the names and addresses in a table. It then reads all labels found in the program, and stores the names and program line addresses corresponding to each label. The assembler also contains values of each machine code instruction and contains code that can calculate addresses used in jumps, and branches.

To complete the assembly process, the program reads each line of the assembly program code, and parses out each value. For each instruction, the values found on each line of code are parsed and addresses are calculated, variables are found in the symbol table, and all values are converted to binary numbers. All values are then concatenated into one 32-bit instruction of machine code, and output to a new file. There is also an output log that shows the conversion results for each line, along with all variables found in the symbol table, and any errors that occurred. The output machine code is simply a file of 32-bit instructions, one per line, and that data is used to initialize the ROM in the VHDL design of the custom processor.

As an example of program code conversion from high level c-code, to assembly, and finally to machine code, in Figure 44 is a simple selection of code from the GFM program. Note that since the if-compare instruction was implemented in the

ISA, the comparison and if-statement from c-code is implemented with only 2 instructions, plus the branch target with the label "IF1". The value of p[nx] in this case is assumed to be in register R24. The entire fixed-point c-code program was converted in this way, and the entire resulting machine-code program was then written into the ROM initialization in the VHDL code for the custom processor in order to run the program.

```
// C-code
if(p[nx] < fpZERO)
{
     p[nx] = fpZERO;
}

// assembly code
OP_IF_CMP IF1 R24 RZERO LT
OP_STRX2 p NOADDR RZERO 1
IF1: OP_NOP

// machine code
11010000000000000101100011110000
01110000011111111111111111001001
00000000000000000000000000000000
```

Figure 44: High-level c-code conversion to assembly and custom machine code

One more note should made here on the topic of compilation techniques. Since the assembler for the custom architecture was created with the use of a compiler, it is noted that compilation techniques to produce efficient assembly code were done manually. This is important because any efficient use of a processor that runs from program code needs to have the code produced efficiently and correctly, and some techniques can improve performance drastically when correctly implemented. For example, the IAR compiler tools that are used with TI MSP430 Family processors, ARM processors, and others, uses several common techniques when, for example, optimizing for speed. One such technique is called loop unrolling, and is particularly useful with pipelined designs. An illustration of a few compilation techniques can be aided with the following code segment, from the original floating-point version of the algorithm code:

```
for(int nx = 0; nx < 21; nx++) {
     for(int ns = 0; ns < 8; ns++) {
               q[ns][nx] = (q[ns][nx] + pold[nx] - poldold[nx]) -
                         (beta[ns] * delt * q[ns][nx]);
     }
}
```

The code involves setting the 168 array variable **q** values given the current value of variables q, pold, poldold, beta, and delt. There is one outer loop that executes 21 times and one inner loop that executes 8 times. Now, the first optimization one can notice is that the values for pold and poldold are independent of the inner loop, that is, they are constant throughout each iteration of the inner loop, and they can therefore be taken outside of the inner loop. In fact, in this program beta, and delt, are constants, and so each beta[ns]*delt can be pre-calculated and taken out of the loop as well (this is also called "common subexpression elimination" because the calculation is unnecessarily repeated if left unoptimized).

Another observation is made that for the inner loop, there is no inter-loop dependence. In other words, regarding loop variable ns, the calculation for q[1][nx] does not depend in any way on the calculation of q[0][nx], and so goes for all values of ns. This means the inner loop can be in fact “unrolled”, or in other words, the iteration of the loop calculation is unnecessary, and any convenient or efficient ordering of calculating each individual value q[ns][--] can be done within each outer loop iteration. To illustrate this, suppose we pre-calculate all values of pold[nx]-poldold[nx] as the first operation on each “nx” loop. We then calculate all values of (beta[ns] * delt * q[ns][nx]) for all values of ns, and unroll the final calculation for q. The code might look as below:

```
for(int nx = 0; nx < 21; nx++) {
     tmp1 = pold[nx] - poldold[nx];
     tmp2 = (beta0delt * q[0][nx]);
     tmp3 = (beta1delt * q[1][nx]);
      ...
     tmp9 = (beta7delt * q[7][nx]);
     // --------------------------------
     q[0][nx] = (q[0][nx] + tmp1 - tmp2;
     q[1][nx] = (q[1][nx] + tmp1 - tmp3;
      ...
     q[7][nx] = (q[7][nx] + tmp1 - tmp9;
}
```

Now, in assembly, there is even one more optimization that can take place, which is that if the common operations are interleaved across inner loop iterations, this can lend itself well to making full use of the pipeline. For example, because we know the inner loop iterations are independent, and we plan to unroll them, we can also split up their respective operations and run them in any order we choose. In other words, it may be smarter to code all the `q[0][nx] + tmp1` operations together in order of ns, as opposed to completely finishing the calculation for each ns first. This can be said for all the operations needed so that instead of ordering all ns=0 operations first, then ns=1, and so on, instead, all `q+tmp1` operations are run, then all (`q+tmp1-tmp<ns+2>`), and so on. For a pipelined design, this grouping of similar operations allows the pipeline to process each operation (for example addition) in a minimum average number of cycles, because of the way the pipeline works. For example, in the custom assembly there is a section of code that corresponds to this loop described here that reads:

```
OP_ALU NOADDR R13 R3 R21 ADD          # R21=q[0][nx] + (tmp1)
OP_ALU NOADDR R14 R3 R22 ADD          # R22=q[1][nx] + (tmp1)
OP_ALU NOADDR R15 R3 R23 ADD          # R23=q[2][nx] + (tmp1)
OP_ALU NOADDR R16 R3 R24 ADD          # R24=q[3][nx] + (tmp1)
OP_ALU NOADDR R17 R3 R25 ADD          # R25=q[4][nx] + (tmp1)
OP_ALU NOADDR R18 R3 R26 ADD          # R26=q[5][nx] + (tmp1)
OP_ALU NOADDR R19 R3 R27 ADD          # R27=q[6][nx] + (tmp1)
OP_ALU NOADDR R20 R3 R28 ADD          # R28=q[7][nx] + (tmp1)
```

The example code above shows additional operations that correspond to interleaved operations from the unrolled inner loop, and each is processed in 1 effective cycle because they can be easily pipelined. It is only be careful inspection and manual compilation that resulted in the code being programmed this way, taking advantage of knowledge of the specific hardware. This and similar techniques is normally the function of the compiler, but in this case this was done manually.

Low-Level Application Profiling (Fixed-Point, Custom Hardware)

At this point we want to see the final profiling results from the custom processor core run using the GFM time-step routine as the benchmark. We have already seen the results on various targets including the MSP430 16-bit processor, various ARM core processors, and Simplescalar. We expect the design here to allow for a relatively low cycle count per step meeting the design constraints, but also with a smaller area and gate count than Simplescalar and possible ARM as well. A lower cycle count and smaller area design would point to a more energy efficient design, since energy is a function of power and time. Reducing the cycle count amounts to reducing the amount of time the processor spends processing actively on, and reducing the gate count and area amounts to reducing the number of transistors on, and so the current and power is reduced as well.

TABLE V: Custom Core Design Speedup

Processor	Floating-Point Cycles / Step	Fixed-Point Cycles / Step	Custom Processor Speedup % versus Float. Pt. / Fixed Pt.	
Custom	--	5175	--	--
MSP430 (16-bit)	227000	1871129	97.7	99.7
Simplescalar	--	19711	--	73.7
ARM7TDMI	45007	15272	88.5	66.1
ARM9TDMI	43071	13133	88.0	60.6
ARM10E	41271	13133	87.5	60.6
XScale	41271	13133	87.5	60.6
CortexM1	80496	33856	93.6	84.7
ARM1136J	45100	13133	88.5	60.6

Table V: Custom processor core speedup compared to various other architectures.

Table V shows the cycle count from the custom processor core versus other processor targets, along with the speedup %. Note that each processor corresponds to a cycle count for the the floating-point version program code, and fixed-point version program. The speedup percentage compares each program version on the various targets, versus the custom implementation. Remember that the Simplescalar simulator simulates floating-point hardware, whereas all other targets listed are fixed-point machines. The point is that we can compare the fixed-point implementation from Simplescalar but we cannot compare the floating-point version, since the compiler ported for Simplescalar cannot simulate floating point in software, only in hardware. Under speedup %, the left column represents the speedup considering the conversion from floating- to fixed-point, as well as the custom hardware design. In other words, the left-column under speedup compares the floating-point program running on various targets compared to the fixed-point version running on the custom processor. The right column compares only the fixed-point implementation running on various targets compared to the same running on the custom processor. Considering the custom hardware only, we see a speedup of between 60 and 84% (not including the MSP430). Considering both hardware and software design optimizations, we see speedup of between 88 and 93% (again not including the MSP430).

It is important to look also at the instruction profile for the custom architecture.

Figure 45 shows the percentage each instruction is called per time-step, while running a 180ft dive for 3 minutes. Because the custom assembly code includes a lot of NOP instructions, in fact the largest percentage of executed instructions are NOPs. After that, we see multiplication and subtraction, followed by LOAD2 and then addition, making up 50% of instructions. Indexed load and store take up 11%, and WHILE instructions make up 2%. Division is now only 1% of instructions, but recall that division is the only multi-cycle instruction and so it takes more time than others and is not pipelined.

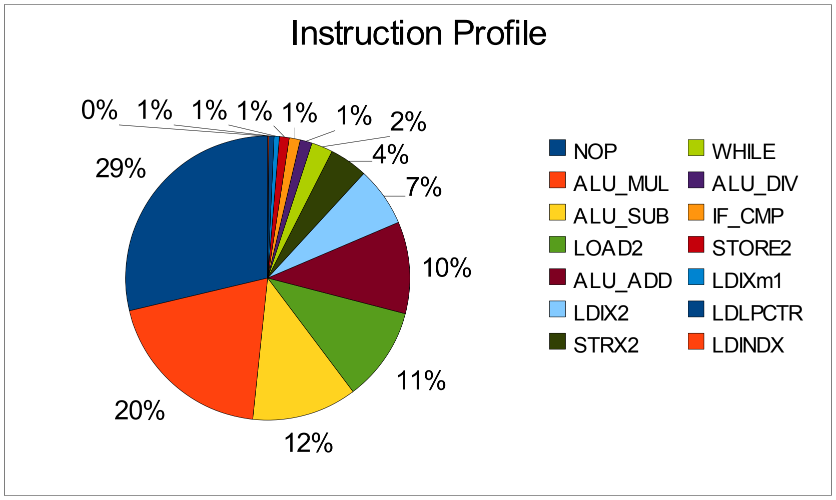


Figure 45: Instruction Profile (Custom Hardware)

Note that in earlier plots like in Figure 32, the instruction profile in fact closely matches the amount of time or cycles spent for each instruction, since all instructions are pipelined and each is fetched roughly 1 cycle after the previous instruction. In this case though, the division instruction breaks up this homogeneity. Figure 46 is a more representative plot where the multi-cycle division instruction is taken into account. The plot more accurately shows how much time is spent executing each instruction per time-step. In Figure 46, the most time is spent in the division instruction, and the NOPs are a less significant percentage. The rest of the percentages follow similarly as in the previous figure.

It is interesting to note also that the total combination of load/store instructions makes up only 15% of cycles. Arithmetic instructions in total make up 63% of cycles. Control instructions like WHILE, IF_CMP, and LDLPCTR and LDINDX makeup 3%, and the remaining instructions are NOPs. This points to the custom CISC-based hybrid load/store design being very suitable for the application, since a small percentage of time is spent on high energy load/store instructions, and most instructions are pipelined arithmetic instructions for which the design is well-suited. The NOPs are a side-effect of making the pipeline design less complicated, since the need for them is a result of not having appropriate mechanisms in hardware to deal with certain dependencies. On the other hand, a compiler or even assembler design can include the needed NOPs easily, without requiring that they be explicitly written into the program by the programmer. For this study, the NOPs required were written into assembly as needed. One such example is before and after a division instruction, NOPs are needed to be sure the pipeline is flushed before the division operation starts. This causes division to require 40 cycles – an acceptable number considering it is handled in hardware, but it is noted here that this could be improved.

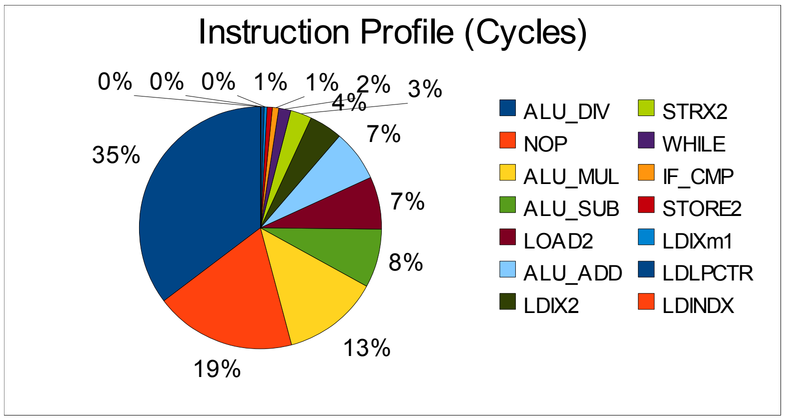


Figure 46: Instruction profile (Based on cycles; Custom Hardware)

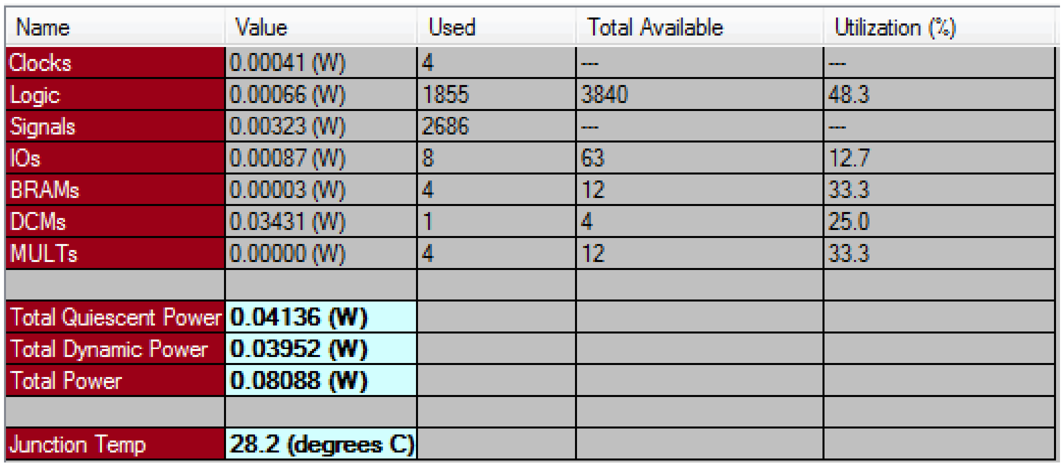
Chapter 6: Results

To begin to look at the results of this study, we need to summarize what was done, what was expected, and finally what resulted from the customized processor core design and software optimizations. This section will cover performance, power and area constraints, and how the custom design addressed these needs. The last section will deal with real examples of how this new design could be used in a practical product design, and what it would mean for the implementation.

Results: Power Constraints

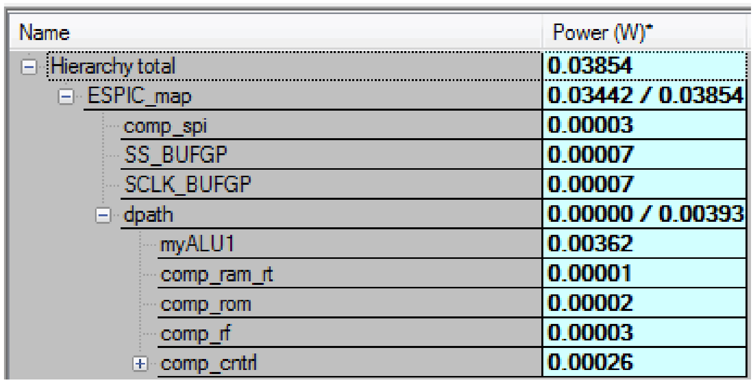
Recall that in Chapter 1 we defined the needs of the application, and in particular design constraints 3 and 4. Constraint 3 specifies that the instantaneous current should be such that a coin-cell battery such as the Sony CR2430 could be used to power the device. Constraint 4 specifies that the life of the product must be a minimum of 1 year considering a certain usage profile of one 30-minute dive per day. To arrive at specific power consumption for the custom device, however, we have to keep in mind that the design was completed on an FPGA using FPGA design tools. FPGA tools can estimate power but in a dive computer system, FPGAs typically use too much power to be considered for in-application use. On the other hand, there are some low-power flash-based FPGA devices that, while expensive, could be used for a real design. That being said, we will report here the power estimation results as given in the Xilinx Power Analyzer software.

The first set of results is the power analysis organized by type, shown in Figure 47. What is shown are power figures in Watts (W), for the various component types in the design. Also shown is the power broken up into quiescent power and dynamic power. For this design, we are more concerned with dynamic power since the quiescent power of an FPGA is much higher than an ASIC, and the two cannot be compared accurately. Notice that the total dynamic power is 40mW, but the DCM is the main contributor at 34mW. The logic, signals, BRAMs, and MULTs are the most important component types here.



Name	Value	Used	Total Available	Utilization (%)
Clocks	0.00041 (W)	4	---	---
Logic	0.00066 (W)	1855	3840	48.3
Signals	0.00323 (W)	2686	---	---
IOs	0.00087 (W)	8	63	12.7
BRAMs	0.00003 (W)	4	12	33.3
DCMs	0.03431 (W)	1	4	25.0
MULTs	0.00000 (W)	4	12	33.3
Total Quiescent Power	**0.04136 (W)**			
Total Dynamic Power	**0.03952 (W)**			
Total Power	**0.08088 (W)**			
Junction Temp	**28.2 (degrees C)**			

Figure 47: Xilinx Power Analysis by Type; Run @ 6MHz



Name	Power (W)*
Hierarchy total	0.03854
ESPIC_map	0.03442 / 0.03854
comp_spi	0.00003
SS_BUFGP	0.00007
SCLK_BUFGP	0.00007
dpath	0.00000 / 0.00393
myALU1	0.00362
comp_ram_rt	0.00001
comp_rom	0.00002
comp_rf	0.00003
comp_cntrl	0.00026

Figure 48: Xilinx Power Analysis by Hierarchy; Run @ 6MHz

In figure 48 we see how the entire datapath component (which includes ram, rom, register file, ALU, controller, and virtually everything in the design besides the SPI and clock manager) amounts to only 4mW of dynamic power. The main contributor here is the ALU, which makes sense given our understanding of the architecture, the instruction profile, and the device usage. Data is transferred only every 6 seconds, and in between transfers, the device is computing thousands of values using the ALU. At a voltage of 1.2V, the total current draw is 32.5mA, and the 4mW coming from the datapath (not including th clock manager) amounts to 3.3mA. For constraint 3, although we are estimating what the result would be for an ASIC, even with the power draw given for the FPGA the constraint could be met.

For constraint 4, we note that using equation 4 with TSPM=2611*10 (1 real-time step and 2611 prediction steps per 6s), 30 minutes of dive time corresponds to 783,300 time-steps. The 783,300 time-steps at 6MHz amounts to 11.23 minutes of calculation (see Table VI). Given a battery with a 220mAh capacity, and considering the 4mA from the datapath, the battery would last 293 days. This is not considering the clock manager, but the analysis is still worthwhile. If the device is fabricated using a technology such that it matches the 0.21mW/MHz at 1.8V that the ARM7TDMI exhibits, then the device would draw 0.47mA on average at 6 MHz and would last over 3.6 years given one 30 minute dive per day. Of course whether or not the design can achieve the same power rating as another existing device depends on many factors, but the calculations are given here for reference.

Results: Area Constraints

Converting from gate counts on an FPGA implementation to equivalent gate counts in an ASIC is not simple, and in fact many say that there is no real way to do it accurately. One reason is that FPGAs contain routing and other logic not present in ASIC designs, and any conversion is not equivalent inherently. Also, ASIC design depend on specific cell libraries and libraries for memories and other components that cannot be accurately estimated from FPGA results. Since the custom GFM processor design was synthesized using FPGA tools, the best we can do is a very rough estimate for gate count, for use comparing to other fabricated IC designs. Making this more complicated is the fact the FPGA design tools have various so-called design goals that allow the designer to specify whether the design should be optimized for power, area, or speed. Xilinx used to output a value of “equivalent logic gates” after synthesis of a design, but they have since removed this feature in latest versions of the Xilinx ISE software. The only way to get a true gate count as would be found in an ASIC, is to use an ASIC synthesis tool such as those made by Synopsis. That being said, we wish to arrive at least at a rough estimate of transistor count in order to compare to other designs.

As was reported in Figure 43, the design uses roughly 55% of the available slices in the 200k gate 3s200 device. As mentioned in the Xilinx app note [43], the designer of an FPGA needs to realize that gate counts or estimates for designs made using FPGA tools depend very much on how well the design matches the architecture of the FPGA. In Xilinx terminology and specifically for the Spartan 3 architecture, one CLB equals 4 slices. Figure 49 gives a representation of the architecture of a Spartan 3 FPGA [44]. The architecture is relatively simple, consisting of "configurable logic blocks" (CLBs), Digital Clock Managers (DCMs), I/O Blocks (IOBs), some Block Ram, and a few multipliers.

From the Spartan 3 datasheet [44], the total "Equivalent logic cells" equals "total CLBs" x (8 logic cells / CLB) x 1.125 effectiveness. One "logic cell" equals one 4-input lookup table plus a "D" flip-flop. Also one CLB equals 4 "slices". Calculating 55% used out of the total number of slices gives 1056 slices used, which means 264 CLBs. The 264 CLBs equates to 2376 logic cells. On the other hand, since numbers are given in Figure 43 for the number of flops and LUTs used from within the slices, a more accurate number may be to use those numbers, reported as

1277 flops and 1846 4-input LUTs used.

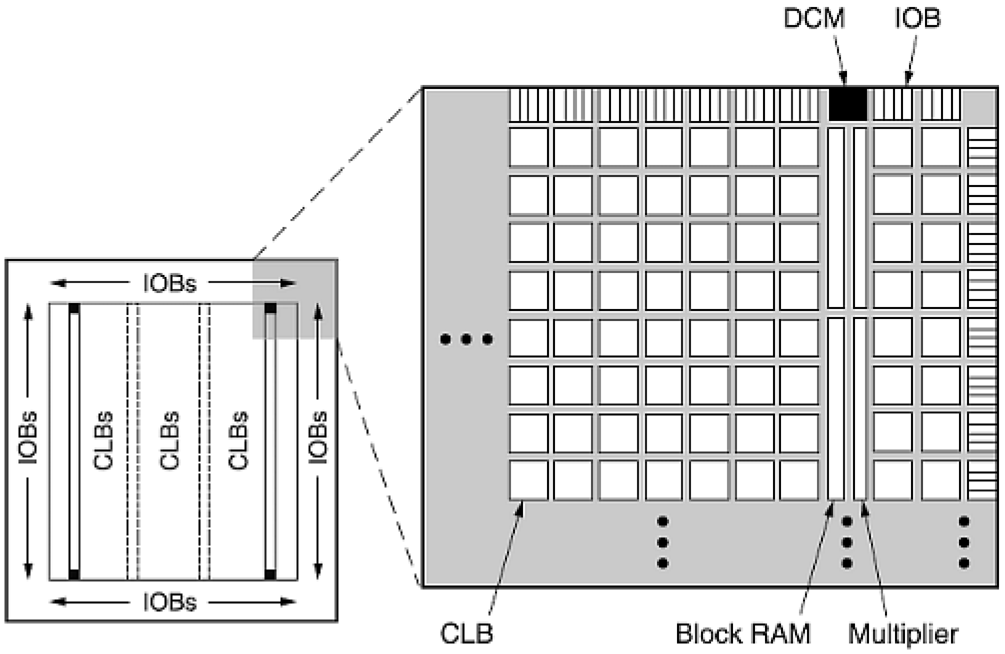


Figure 49: Xilinx Spartan 3 Architecture

To convert from number of "D" flops to number of transistors, we can use the number 26 as given in [45]. The transistor count then is roughly 33k considering flops only. The app note [43] explains that converting LUTs to number of gates is difficult since LUTs can be used for different purposes. The gate count per LUT is listed as between 1 and 9. For the sake of this study we will use 6 gates (roughly 24 transistors) per LUT, and so the 1846 LUTs amount to roughly 44k transistors, bringing the total now to 77k. Transistor counts for memory are typically not included in processor core comparisons, so the 4kb of RAM used in this design is noted, but not included in area estimation, especially since it is a relatively small memory size that surely will not be a factor even in watch-style design. The design does use 4 of the dedicated multipliers available, and so these need to be accounted for. The author is using an estimate of 2200 gates required for each 18x18 signed multiplier, taken from an article on EE Times. Using 4 transistors per gate the 4 multipliers make up 35k transistors, making out total 112k transistors, not including memory.

Now, considering that an ARM7 is given [46] as using 2.2 sq. mm, and 74k transistors, and an ARM9 is 4.15 sq. mm and 112k transistors (using 0.35um technology), this design using 112k transistors (estimated) is quite a good result, and in fact we can say here that this would meet the constraint in terms of area. Related to area is power and energy, and so we want to expand on how area affects energy. As we saw in Table V, this design amounts to 66% and 60% speedup when compared to ARM7 and ARM9 respectively. The estimated transistor count for this custom design amounts to a 51% increase in area for the ARM7 and 0% increase for the ARM9. Table VI shows how the ARM7, ARM9, and the custom design compare in terms of transistor count, cycle count, and energy. Energy here is an estimated relative measure that uses cycles as runtime, and transistor count as a rough measure of power; given as the multiplication of the two values and then normalized for the ARM9 architecture. The last column gives percent change in energy for the custom design versus the other two.

TABLE VI: Transistor Counts and Energy, Custom vs. ARM7/9

Processor	Cycles / Step	# Transistors (core only)	Energy	Normalized Energy	% Change Energy
Custom	5175	112k	580	0.39	--
ARM7TDMI	15272	74k	1130	0.77	-48.67
ARM9TDMI	13133	112k	1470	1.00	-60.54

Table VI: Transistor Counts and Speedup, Custom vs. ARM7/9

It is noted that this method used in this section for estimating gate counts and energy is not very accurate, but does give a rough sense of where the custom design falls in comparison to the other architectures. The area constraint for this design was given by Constraint 5, and it stated that the design should be able to fit within a 12 sq. mm area. Since the ARM9 is quoted as requiring only 4 sq. mm, and because our transistor count estimate is exactly the same as for the ARM9, we can conclude that we should be able to meet this constraint with the same or more advanced technology node. Another rough validation (without requiring complex area estimation techniques) is to note that since the design already meets the constraint as designed (and using only a fraction of the device resources) in the 100-pin VQ100 package Xilinx device used in the study, it would surely meet the constraint if fabricated into an ASIC.

Results: Performance Constraints

Recall that in Chapter 1 we defined the needs of the application, and in particular design constraint 2b. Constraint 2b requires that the time per GFM time-step be less than 1.149ms in duration. Recall that this was designed such that the 7 predictions and 1 real-time time-step as specified in Chapter 1 could be completed within 6 seconds, at 50% duty. This is a hard real-time bound that takes into account a real-time operating system that takes advantage of low-power processor states. Note also that this is essentially equivalent to a processor being able to complete 2611 time-steps within 3s at 100% duty (active on 100% of the time). It was also noted that in order to calculate the amount of time required by the implementation to complete 1 time-step, the operating frequency, and cycle count per step needs to be known.

TABLE VII: Time Per Step vs. Frequency

Processor	**Cycles / Step**	**TPS @ 2MHz (ms)**	**TPS @ 4MHz (ms)**	**TPS @ 6MHz (ms)**	**TPS @ 8MHz (ms)**
Custom	5175	2.59	1.29	0.86	0.65
ARM7TDMI	15272	7.64	3.82	2.55	1.91
ARM9TDMI	13133	6.57	3.28	2.19	1.64

Table VII: Time per step at various system frequencies. Fixed-point implementation.

To be able to determine whether this design meets Constraint 2b, Table VII gives the time per step at various frequencies, for the custom design as well as ARM7 and ARM9 target architectures. It is important to note that although Table V gives cycle counts and % speedup, that is not enough to determine whether a hard real-time bound can be met. Only by calculating at specific frequencies can we understand the practical impact of the custom design, or other designs for that matter. What we see is that for frequencies of 8MHz and below, the only architecture that can meet the performance constraint is the custom architecture. In fact, the custom design meets the constraint at any frequency above 4.5MHz. Table VII is important because the higher the system frequency the higher the current, and in an ultra-low power design such as a dive computer the coin-cell batteries used may not be able to provide enough battery life for the application. The challenge is to implement the design in a low-power technology so that its full potential as a performance efficient processor for GFM can be realized.

As an example battery life calculation, suppose a diver makes one, 60-minute dive per day. As is given for the ARM7TDMI architecture performance characteristics [46], suppose the processor consumes 1.5mW/MHz at 3V for a 0.35um process. Now suppose a coin-cell battery with a 220mAh rating. Note that the 60-minute dive requires (60min)*(2611*10 steps/min), or 1,566,600 GFM time-steps. These time-steps would require 66.58 minutes to complete. Of course this cannot be done since the dive is inly 60 minutes long, as this is expected since we know the ARM7 cannot meet the constraint. On the other hand, if we can assume that given some process technology that the custom design has the same power rating, then the GFM calculations would take 22.45 minutes to complete, and at 6MHz, drawing an average of 1.12mA at 3V, the device would last about 6.5 months - very reasonable considering that the advanced GFM algorithm would be running in real-time.

Results: Summary Example

Suppose we now have constructed a watch-style high-end dive computer (similar to the Oceanic OC1), and that the design includes a custom GFM processor. The main system program has been modified to use the GFM algorithm, and as explained in Chapter 1, sends and receives data to and from the GFM chip every 6 seconds. In between those 6 seconds, the GFM chip runs the real-time calculation, as well as the 7 predictions as described in Chapter 1, for use in calculating the NO-D time and the result of a safety stop.

Because the processor was custom designed, its performance is very high. It runs GFM time-steps in roughly 5175 cycles, and assuming we run the system at 8MHz, each time-step only takes 650us. As reported in Table V, this amounts to 66%, 61%, and 74% speedup as compared to the same fixed-point algorithm code version running on an ARM7, ARM9, and Simplescalar implementation, respectively. Since we want to run most of the time in a low-power state, we run at 50% duty, and so we run only as many time-steps per second as will take ½ second to complete. In this case, we run 769 steps per half-second, and we are in a low-power state the other half-second. Every 6 seconds then, we have the ability to run 4615 GFM time-steps, while only active at high frequency half the time. Recall that to be able to run all 7 predictions including the ascent with a safety stop every 6 seconds, it was required that 2611 time-steps were completed within 6 seconds at 50% duty, and so this is easily accomplished. In fact, if we only want to run 2611 steps, we need only to be running calculations 28.2% of the time.

It is important to note also that the 7 predictions outlined in Chapter 1 are not mandatory. These predictions were chosen as a very aggressive goal, and one that would allow the product to calculate in real-time the effectiveness of a safety stop as well as perform enough calculations needed to find the NO-D time. As was mentioned in a previous section, if the product designer wished to do say only 3 predictions for use in finding the NO-D time, and instead using more cycles to do other predictions or not use them at all, that is up to the designer. In fact for that matter, the 50% duty cycle is not a strict requirement either, and a designer can choose to set the amount of “active/on” time at 10% for longer battery life. This is entirely up to the designer and the custom architecture gives the product designer the freedom to choose, because of the increased efficiency the custom architecture provides. The point is that because the architecture was designed with such aggressive goals, the performance is so high that many more options exist in terms of real practical product implementations.

Because of the low transistor count as reported in the second section of this chapter, and given the simulation power analysis and interpretation in the first section of this chapter, we can say that the device would in fact fit within a low-profile, battery operated watch-style design and form factor, and the design is estimated to be able to meet or exceeds the 1.5mW/MHz @ 3V rating given for the ARM7TDMI device fabricated using 0.35um technology. The efficiency in terms of cycle count has implications related to power. Given 1.5mW/MHz @ 3V, we can say we use 12mW @ 8MHz @ 3V, or 4mA @ 8 MHz @ 3V. Running 28.2% of the time at 8MHz and at 3V means that the average current would be 1.13mA. Given a 300mAh battery such as the CR2430, the device would last 265 hours running continuously. Considering typical usage of 1, 30-minute dive per day, the device would last 531 days or 1.5 years.

To summarize this example, a a high-end watch style dive computer device could be designed using this custom processor design, using a coin-cell CR2430 battery, that could last over 1 year. This device would be running the GFM algorithm in real-time, used to calculate the no-decompression time via 6 prediction calculations, as well as 1 prediction made to show the efficacy of a standard safety stop, within the real-time bounds of 6s indicated by the GFM inventors. Running at only 28% duty, there is room for additional or supplemental predictions, and room for other options in terms of calculating GFM-based outputs. For example, a PDC designer might choose to use only 3 predictions for calculating the NO-D time, and to use the remaining cycles available to calculate the effect of three different safety stops, or maybe reduced or increased ascent/descent rates. This additional capability afforded by the custom design allows the designer freedom to utilize GFM in a dive computer in a way that would not be possible otherwise, using off-the-shelf components. Of course there are devices that are powerful enough to compute GFM time-steps in a small number of cycles, but to compete with this custom design they would need to be able to do it at a low frequency and at low power as well.

On the other end of the spectrum, this time considering a less demanding application, the custom design could be used to run calculations for a very simple device as well, with equally useful benefits. Suppose a dive computer manufacturer wishes to build a low-end computer. The low-end computer might use a larger battery (since the overall shape is larger and possibly even hand-held), and might have less demanding need for GFM predictions. In this case, the custom processor still affords the same efficiency running GFM calculations, and the designer has the freedom to choose either a lower frequency, or change the duty cycle while still meeting the long battery life requirements of the product. Since the number of time-steps is reduced for the less demanding application, there is room to reduce the system frequency while still meeting the real-time requirements.

In terms of manufacturing, as in the past, fabricating your own custom IC is an expensive endeavor. Typical costs can be in the millions of dollars, and for a small company like a dive computer manufacturer this can be prohibitive. On the other hand, there are several alternatives. One is to have a so-called “Structured ASIC” manufactured. This essentially amounts to programming the top layer of an otherwise pre-determined silicon die with a custom design, such that the cost of creating this custom IC is much lower than a full custom IC. For example, one company called eASIC quoted (on 3/28/08) their software at $30k (NRE), and the cost for 1,000 prototypes at $14,950 ($14.95 each). This is relatively expensive compared to GPP tools, but magnitudes less than a full ASIC design cost. The benefit is that you get custom IC's at costs much lower than FPGAs, and almost comparable to GPPs, but with the high efficiency and low area and power that the custom design provides.

Another option for implementation is a low-power flash-based FPGA. These are quite expensive devices compared to GPPs, but require no fabrication, and have very good power and area characteristics. For example, the Actel Igloo Nano series AGLN250 FPGAs have up to 250k system gates, 36k bits of RAM, and exhibit a typical 24uW power draw. The quiescent current is 34uA at 1.5V, and 1.8uA in sleep mode. The methodology is quite complicated to calculate dynamic power, but essentially amounts to figuring out what resources are being used, and summing up all the power from each. Of course the power used depends on device utilization and the frequency and amount of switching. The device comes in various packages including a 100-pin QFP package that would easily fit inside even the most demanding watch-style designs. These devices sell for under $20 in quantities of 100+, and design software is relatively inexpensive, even coming with a free 1 year license.

Results: Overall Summary

The Gas Formation Model scuba diving algorithm and its related target application of a low-power, small footprint, battery-operated and safety critical scuba dive computer was detailed in Chapter 1. Constraints for such a design were detailed in terms of performance, area, and power. In Chapter 2, sources of CMOS power were discussed and common low energy hardware design techniques were presented. Chapter 3 presented characteristics and trade-offs for existing implementation options for implementing the GFM algorithm in a dive computer system, and Chapter 4 discussed the concept of a hardware/software design flow, in particular, required to design a custom processor for implementing GFM.

Chapter 5 detailed how an in-order, pipelined, 32-bit, custom hybrid architecture processor was designed for implementing GFM inside dive computer. Important steps such as hardware/software partitioning, high- and low-level application profiling, floating-point to fixed-point conversion, instruction set architecture (ISA) design, processor core architecture design, custom functional unit (CFU) design and toolset design were detailed as they related to the design flow. The design included a floating-point to fixed-point algorithm conversion, followed by a manual compilation into assembly code for the custom architecture. Chapter 6 presents the results of the design, again in terms of performance, area, and power. The efficient design exhibits an instruction profile that uses only 15% of cycles for load/store operations, and the majority (63%) of operations performed are pipelined arithmetic instructions. The custom design meets the real-time performance requirements at any frequency above 4.5MHz, and outperforms the popular ARM7 architecture by a factor of 3, exhibiting a speedup of 66%, and has been estimated to be able to meet the power and area requirements of the application as well.

Given a practical design example, the conceptual device was shown to be able to last over 1.5 years on a 3V coin-cell battery with 300mAh capacity, given a particular usage profile. Additional practical design options were given in order to allow the designer to take full advantage of the GFM algorithm, while still meeting various design constraints for more or less demanding applications. Finally, three real implementation options for the custom design were discussed. This custom design was shown to have promise as a real product in industry, and work is ongoing to create its first prototype working within a real dive computer system.

References

[1] S. Crow; J. Lewis, "Dive computer and method for determining gas formation," US Patent 7,313,483 B2. Dec.,25,2007.

[2] S. Crow; J. Lewis. (2008,Third Quarter). Introducing the Gas Formation Model. *The Undersea Journal from Professional Association of Diving Instructors* [Print]. pp 57-59.

[3] B. Wienke, "," in *Technical Diving in Depth*, Flagstaff, AZ: Best Publishing Company, 2001, pp. 3-5, 67-110, 131-160, 257-305.

[4] J. Lewis; K. Shreeves, "," in *Decompression Theory, Dive Tables, and Dive Computers*, 2nd Ed. Santa Ana, CA: International PADI Inc, 1993.

[5] PADI. *About Padi* [Online]. Available: http://www.padi.com/scuba/about-padi/

[6] Scuba Diving Magazine. (2009, Aug. 7) *Digging Deep on 2009's New Dive Computers* [Online]. Available: http://www.scubadiving.com/gear/dive-computers/2009/08/digging-deep-on-2009s-new-dive-computers/

[7] C. Passerone, "Real time operating system modeling in a system level design environment," in *IEEE International Symposium on Circuits and Systems*., 2006, pp. 2549-2552.

[8] Z. Murtaza; S.A. Khan; A. Rafique; K.B. Bajwa; U. Zaman;, "Silicon real time operating system for embedded DSPs," in *International Conference on Emerging Technologies*., 2006, pp. 188-191.

[9] F. Colaco; F. Cardoso, "Flying Objects: a modular real time object operating system," in *11th IEEE NPSS Real Time Conference*., Santa Fe, NM, 1999, pp. 543-546.

[10] NOAA, *Dive Planning Forms, Tables & Formulas* [Online]. Available: http://www.ndc.noaa.gov/dp_forms.html

[11] D. Chen; S. Dwarkadas; M.C. Huang; K. Shen; J.B. Carter, "Program phase detection and exploitation," in *20th International Parallel and Distributed Processing Symposium.,* RI, 2006, 8 pp.

[12] A.S. Dhodapkar; J.E. Smith, "Comparing program phase detection techniques," in *36th Annual IEEE/ACM International Symposium on Microarchitecture*., 2003, pp. 217-227.

[13] Sony Micro Battery. Lithium Manganese Dioxide Batteries [Online]. http://www.sony.net/Products/MicroBattery/

[14] Apple. Lithium Ion Batteries [Online]. http://www.apple.com/batteries/

[15] Apple. Batteries - Iphone [Online]. http://www.apple.com/batteries/iphone.html

[16] Texas Instruments. Wireless Pelagic Dive Computer Extends Battery Life Using TI MCUs [Online]. http://www.dsp-fpga.com/news/db/?4001

[17] T. Glokler; H. Meyer, *Design of Energy-Efficient Application-Specific Instruction Set Processors (ASIPs)*, 1st Ed. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2004, pp. 1-116.

[18] L. Yann-Hang; K.P. Reddy; C.M. Krishna, "Scheduling techniques for reducing leakage power in hard real-time systems," in *15th Euromicro Conference on Real-Time Systems*., 2003, pp. 105-112.

[19] Z. Xiaoying; Y. Jiangfang; D. Dong; X. Cheng, "Leakage Power Reduction for CMOS Combinational Circuits," in *8th International Conference on Solid-State and Integrated Circuit Technology*., 2006, pp. 1621-1623.

[20] L. Dake, *Embedded DSP Processor Design*, 1st Ed., Burlington, MA: Morgan Kaufmann Publishers, 2008, pp 24-43, 87-152, 159-183, 217-237, 239-308, 369-396, 475-510, 597-616.

[21] P. Ienne; R. Leupers, *Customizable Embedded Processors - Design Technologies and Applications*, 1st Ed. San Francisco, CA: Morgan Kaufmann, Morgan Kaufmann, 2007, pp 117-145,209-233,381-394.

[22] L. Zhang; Y. Zhang; W. Zhou, "Floating-point to Fixed-point Transformation Using Extreme Value Theory," in *Eighth IEEE/ACIS International Conference on Computer and Information Science*., Shanghai, China, 2009, pp. 271-276.

[23] C. Shi; R.W. Brodersen, "Floating-point to fixed-point conversion with decision errors due to quantization," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*., 2004, pp. V41-V44.

[24] H.S. Stoker, "Gasses, Liquids, and Solids," in *General,Organic,and Biological Chemistry*, 5th Ed. Belmont, CA: Brooks/Cole, 2008, ch. 7, pp. 172-173.

[25] Z. Bo; D. Blaauw; D. Sylvester; K. Flautner, "Theoretical and practical limits of dynamic voltage scaling," in *41st Design Automation Conference*., Ann Arbor, Ml, 2004, pp. 868-873.

[26] A. Forestier; M.R. Stan, "Limits to voltage scaling from the low power perspective," in *13th Symposium on Integrated Circuits and Systems Design*., Manaus, 2000, pp. 365-370.

[27] SimpleScalar LLC [Online]. Available: http://www.simplescalar.com/

[28] C. Weaver; R. Krishna; L. Wu; T. Austin, "Application Specific Architectures: A Recipe for Fast, Flexible and Power Efficient Designs",in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2001, pp. 181-185.

[29] J.L. Hennessy; D.A. Patterson, *Computer Architecture - A Quantitative Approach*, 4th Ed. San Francisco, CA: Morgan Kaufmann, 2007, Ch. 2-3, pp. 66-142,154-185.

[30] P. Biswas; V. Choudhary; K. Atasu; L. Pozzi; P. Ienne; N. Dutt, "Introduction of local memory elements in instruction set extensions," in *41st Design Automation Conference*., 2004, pp. 729-734.

[31] P. Biswas; N. Dutt; P. Ienne; L. Pozzi, "Automatic Identification of Application-Specific Functional Units with Architecturally Visible Storage," in *Proceedings Design, Automation and Test in Europe*., Munich, 2006, pp. 1-6.

[32] J. Lee; K. Choi; N. Dutt, "Efficient instruction encoding for automatic instruction set design of configurable ASIPs," in *IEEE/ACM International Conference on Computer Aided Design*., 2002, pp. 649-654.

[33] Gschwind, M, "Instruction set selection for ASIP design," in *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*., Rome, 1999, pp. 7-11.

[34] N. Clark; H. Zhong; S. Mahlke, "Processor acceleration through automated instruction set customization," in *36th Annual IEEE/ACM International Symposium on Microarchitecture*., 2003, pp. 129-140.

[35] S. Kim; M.H. Sunwoo, "Low Power ASIP Architecture Optimization based on Target Application Profiling," in *IEEE International Symposium on Circuits and Systems*., New Orleans, LA, 2007, pp. 3764-3767.

[36] S. Kilts, "," in *Advanced FPGA Design: Architecture, Implementation, and Organization*, Hoboken, NJ: John Wiley & Sons, 2007, ch. 3, pp. 37-47.

[37] A.M. Nielsen; D.W. Matula; C.N. Lyu; G. Even (2000, Jan). An IEEE compliant floating-point adder that conforms with the pipeline packet-forwarding paradigm. *IEEE Transactions on Computers* [Print]. volume 49 issue 1, pp. 33-47.

[38] Coware. *Processor Designer* [Online]. Available: http://www.coware.com/products/processordesigner.php

[39] Oceanic. *Personal Dive Computers* [Online]. Available: http://www.oceanicworldwide.com/p_computers.html

[40] J. J. Yi; D. J. Lilja; D. M. Hawkins, "A Statistically Rigorous Approach for Improving Simulation Methodology," in *IEEE International Symposium on High-Performance Computer Architecture*, 2003, pp. 281-291.

[41] M. Steinhaus; R. Kolla; J. L. Larriba-Pey; T. Ungerer; M. Valero, "Transistor Count and Chip-Space Estimation of SimpleScalar-based Microprocessor Models," in *Proceedings of the Workshop on Complexity-Effective Design*, 2001, pp. 1-15.

[42] H. Warren, "Unsigned Long Division," in *Hacker's Delight*, Boston, MA: Pearson Education Inc, 2003, pp. 149.

[43] Xilinx. *Gate Count Capacity Metrics for FPGAs* [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp059.pdf

[44] Xilinx. *Datasheet Spartan 3* [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf

[45] Xilinx. *Using Configurable Logic Blocks* [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug331.pdf

[46] S. Segars, "The ARM9 Family - Higher Performance Processors for Embedded Applcations," in International Conference on Computer Design: VLSI in Computers and Processors, 1998, pp. 230-235.