GreenArrays, Incorporated

WHITE PAPER

How the GreenArrays™ Architecture Naturally Conserves Energy

Written by: Greg Bailey

GreenArray chips enable unprecedented control over power consumption in both quantity (many small computers, each consuming power only when necessary) and in time (ability to stop and start consuming power in picoseconds at the level of a single core). The basic mechanism for communication within and outside the GreenArray chips naturally takes advantage of these capabilities to minimize not only instantaneous power, but its integral over time, which is energy.

Since it is energy that we wish to conserve – both on a planetary scale, and on a very small scale such as devices powered by batteries or other emerging methods of gathering and storing small amounts of energy on the human body and elsewhere – energy per unit of necessary work accomplished is the appropriate perspective from which to characterize and compare alternative ways of solving problems.

In this paper we will examine the ways in which the GreenArray chips minimize energy consumption for tasks ranging from high speed parallel processing to execution of large bodies of code written in high level languages such as C, using examples of the manifold execution models that may be supported by the GreenArray chips.

Energy per Unit of Useful Work

Most computers consume power continually even when there is no useful work to do. Those which have low power shutdown modes, these days, generally shut down all or most of the machine rather than only small selective part, and in general the transition to and from low power modes takes too much time to cycle rapidly. Thus it has been traditional to simply specify the current and thus power consumed by the machine while running. It is also traditional to specify the speed of the machine in millions of instructions per second (MIPS). The speed in MIPS gives one an idea of the maximum amount of work a machine can do in a unit of time, and of the ballpark latency of the machine in reacting to external events. The static power level indicates what the energy cost will be to perform that maximum amount of activity over a given interval of time. These numbers are useful for understanding the maximum output of a given computer but are of little use at all in evaluating the economics of using that computer for a given application in an energy conscious world. In real world applications, the maximum possible output of the computer is seldom necessary all the time, or even most of the time.

If energy is a consideration, or, as is the case in many applications, a hard limit, it is necessary to know something quite different. Specifically, how much energy is required for the computer to perform a defined amount of useful work, which is required by the application. For example consider a hearing aid. Let us say that the problem definition is as follows: At 40 KHz the device must sample an input from a microphone, perform a series of

defined calculations, and update an output to a speaker. We measure the power required by the computer to do this necessary unit of work – processing one sample – integrated over the time interval of 1/40,000 second during which it must be accomplished. If the machine must run full power while doing nothing part of that time as is the case with convential computers, that waste must be included in the integral. This integral will be expressed in energy units such as Joules (1 Joule = 1 Watt for 1 Second). We multiply this amount of energy by 40,000 to get energy over one second; we multiply it again by 86,400 to get energy per day. We then divide the capacity of the battery by the daily energy requirement and this gives us the number of days the battery will last.

Anything that can be done, at the hardware or software level, to conserve energy – in large or small amounts – will make that battery last longer. If the energy source is human body heat or kinetics, or light energy, captured and stored in some device such as a battery or even just a capacitor, the available supply of energy may be considerably smaller than with a conventional pre-charged battery and the aggressiveness of energy saving measures will be commensurately greater.

Fine Grained Control of Energy Consumption

Fine grained conservation of energy is a discipline in which the GreenArray chips are uniquely qualified to excel.

GreenArray chips consist of arrays of computers. For example, the GA144 chip contains 144 of our F18A computers. Each F18 is a self-contained computing engine with its own private RAM, ROM, stacks, ALU and registers. Each F18 is capable of communicating directly in four directions; to other F18 nodes, or, if a node is on one of the four edges of the rectangular array, to the external world using one or more pins with various peripheral circuitry appropriate to the nature of the interface. When an F18 is actively executing instructions, its power consumption is typically about 4.5 milliwatts when running. When an F18 is waiting for an event, its leakage power consumption is instead on the order of 100 nanowatts.

Between nodes there are 18 bit, bidirectional parallel buses called Comm Ports. The handshaking on these ports is trivially simple and the latency and jitter are low, on the order of gate delays, tens of picoseconds. When a node is reading or writing a Comm Port, it proceeds with no delay if its partner is ready to transfer data. If the partner is not yet ready, the node waits until it is. While waiting, a node's power consumption drops, within picoseconds, to nothing but leakage. When the partner becomes ready the idle node resumes its work within picoseconds.

These properties enable power control at an unprecedentedly fine scale both in small quantity – the miniscule 4.5 mW typical core power to nearly zero when the core has no useful work to do – and in time intervals, since it is practical for a node to start and stop consuming power at cycle rates as high as 100 MHz. When any of our many nodes has nothing to do, it is waiting and we save 4.5 mW until it needs to be active again. Whenever a node has nothing to do for several nanoseconds its power consumption drops by a factor of 45,000. When one node waits for an event for even as little as 1 nanosecond, we have saved 4.5 picojoules (pJ) of energy. Integrating many of these small savings in power over time, substantial savings in energy occur, automatically, as a natural consequence of the architecture.

External communications are similar. Power consumption varies from peripheral to peripheral, depending on the nature of the interface, the duty

cycle of its activity, and the software in use. For example, an A/D converter will use several mA when it is tracking the input voltage, but under program control the converter may be stopped and its power consumption reduced from mA to nanoamperes. External pins may be used as event sources, so that a node may wait, consuming only leakage, until the external pin has changed state.

An entire GA144 can be waiting for an external event, with one node watching, for example, a pin. Even if that pin is a serial input source and the event is a pattern rather than a simple state change, this may require only a very small duty cycle from the one node that is watching. So, the core power required may be only 14.4 microwatts for the idle chip, plus 450 microwatts for 1/10 duty cycle of the watching node, or 4.5 microwatts if only 1/1000 duty cycle is required.

Just as we can save a great deal of energy in a single chip by using energy only when actually needed, an application in which a massive number of GreenArray chips are deployed can conserve a substantial amount of energy on a global scale. Every picoJoule conserved by a GreenArray chip is a picoJoule that may be used for some more productive purpose by our civilization, such as prolonging its life on Earth.

Multilevel Programming

The GreenArray architecture is suitable for use a wide variety of programming models, depending on the nature and demands of the application. While it is not suitable for every application – no computer is – it is suitable for many, especially when low power is a controlling concern. For example, a single GA144 can execute an absolute maximum on the order of 100 billion instructions per second. Its internal memory RAM is absolutely limited at 9216 18-bit words. If an application requires a teraflop of computing performance, or if it requires access to more than 2560 words of RAM at the highest, internal speeds, then a single GA144 will not meet the requirements of that application. Most likely it will require a computer that consumes considerably more power than does a GA144. If the application needs to be employed in a low energy environment, it will most likely need to be rethought and recast as a different solution before any low energy computer could be built to run it.

However, if the application is within the performance and capacity limits of a GreenArray chip, there exists one or another model whereby it may be implemented.

If CPU performance requirements are high, the fastest parallel programming methods must be used. As performance requirements diminish, it becomes feasible to use abstractions on the hardware down to and including conventional high level programming languages.

If large memory is required, and the access speed necessary is within the capabilities of external memory, the GreenArray chips can be connected directly to SRAM or SDRAM devices.

Resident Machine Code

RAM or ROM resident code and data yield the highest performance possible in the GreenArray chips. Within an instruction word, opcodes can execute at speeds on the order of 700 MIPS, and by using the micronext instruction up to four instructions may be executed in a single word loop at sustained rates this high. Parallel data or signal flow blocks may be allocated to nodes such that groups of nodes, whose members are all doing simultaneous useful work each cycle of operation, implement the equivalent of continuous process models such as control or signal processing algorithms. Energy need be consumed only when there is useful work to be done; nodes use nanowatts

when waiting for data or events. The energy required to perform a quantum of work is also small; for example, an F18 node requires less than10 pJ (picojoules) of energy to add two numbers, based on a time of 1.4 nS at 4.5 mW to perform the operation. By comparison, a TI MSP430F at minimum supply voltage consumes ~300 pJ to add two numbers, based on time of 125 nS at 2.37 mW (minimum power 8 MHz, 165 µA/MHz, 1.8v from manual). We hope to document empirical energy costs of other fundamental algorithms run in this 100 to 700 MIPS environment soon.

We provide a simple Forth language and tools for writing, simulating and debugging in simulation, and interactively debugging F18 code on live chips at this level. The level of difficulty is comparable with that involved in writing machine code for any other machine this simple, with only 33 opcodes, stacks, and a few registers. Like any other machine code environment, simplicity "rules," and the opportunities for optimizing the methods used are legion

Streamed Machine Code

There are various ways in which the GreenArray capability of port execution may be combined with external memory to feed long programs of F18 code into a node for execution at external memory speed. We are experimenting with one such method now, and will soon be experimenting with a second. When it is necessary to execute a program whose necessary size exceeds the capacity of local memory at high speed, this is a good way to accomplish that; programs that are written in slightly extended F18 code can be run in the 10 to 100 MIPS performance range depending on the external memory. No energy measurements exist at present but, when they do, the dominant factor will probably be external memory cycles.

High Level Language Interpreters

By using a large external memory device and devoting a very small fraction of a GreenArray chip to the task, it is practical to build a software interpreter or emulator for a virtual machine that can run any desired high level programming language. There is nothing new or particularly challenging about building such environments; systems programmers have done this sort of thing for decades, and the result is suitable for running general applications that are within the performance capabilities of the virtual machine implementation. What **is** new is the miniscule amount of hardware and power with which GreenArray enables it to be done.

It would not be difficult to build a virtual machine supporting C, and there are many people and companies in the US alone for whom building such a machine and completing a "port" of the C language compiler and library to the virtual machine would be simply a repetition of something they had done before. Once this has been done, the GreenArray chip can run any C program which fits in the external memory and will satisfy any C application requirement that is met by the resulting execution speed.

As an illustration of this level of programming, we have done the same thing as indicated above with the high level programming language Forth, using the model called eForth, as a design exercise. While this is not necessarily the best that can be done … the first try at something is seldom optimal … the resulting properties illustrate what can be achieved.

A 32 megaword Micron SDRAM chip is directly connected to the combined 40 pin RAM interface of the GA144 nodes 007, 8, and 9. F18 code in nodes 007, 8, 9, 107 and 108 comprise a complete, dual port, random-access SDRAM controller which generates the required

clock and refresh signals continuously and provides, inside the chip, the interface of a dual port SDRAM controller for read and write operations.

At a board level, the difference between an idle GA144 and one which is controlling the SDRAM actively, but is not doing any read or write operations, is on the order of 40 mW. This is the cost of the active SDRAM chip and of the five node SDRAM controller, including core power for the GA144, I/O power for the GA144, and power for the SDRAM chip. It would apply to any application that needed large memory.

We then developed a two-node virtual machine interpreter, using nodes 105 and 106, to support eForth. The complete eForth system and compiler was then ported to this virtual machine. The resulting compiled binary for the virtual machine, comprising about 5,000 words of executable code, is loaded into the external SDRAM. This yields a system that can compile and run high level Forth code, in a virtual machine that can address and access 32 megawords of external RAM. The whole thing can be loaded from any external boot device including SPI flash which may in that case be used as read/write mass storage.

The performance of this particular eForth implementation benchmarks comparably with that of threaded Forth implementations on the DEC LSI-11/73, the 11/44, and the MicroVAX II. These machines were all on the same order of performance as defined one VAX MIP, an industry standard unit of measure for many years.

Is it the best we, or anyone else, could possibly do? No; we consider it a design exercise and only the first effort made along these lines for our architecture. The next virtual machine will probably give much better performance. Further, its performance can be enhanced since the virtual machine includes facility for interacting with and controlling resident machine code in other nodes, and for making use of streamed machine code. So, in this regard, it is a model for a multilevel programming system itself, capable of implementing critical functions in either of the above mentioned higher performance, more specialized methods.

However, not every embedded application requires mainframe performance. Much good work can be done with performance on the order of one to ten VAX MIPS. Meanwhile, what is the power cost of this method of implementing and running high level language applications?

Our measurements indicate that the incremental cost of running a compute bound eForth program is 22 milliwatts over the cost of simply powering up and refreshing the SDRAM, or a total of 62 milliwatts over the cost of an idle board and chips.

In this particular implementation if eForth is waiting for an event then the 22 milliwatts of its operation are conserved, so we retain the fine grained conservation capability intrinsic to the GreenArray architecture.

With a sufficiently large external 1.8v static RAM replacing SDRAM, the virtual machine would run at higher speed, would use less power, and when the virtual machine was inactive the 40 milliwatts used in clocking and refreshing the RAM would vanish. Likewise the RAM controller would be reduced from five to three nodes.

Again, not every conceivable application will be satisfied with the performance attainable by an external virtual machine that uses only a handful of GreenArray nodes for its implementation. However, if the application requirements are such that this is a feasible method, this may very well be the lowest power solution – 22 or 62 milliwatts in this configuration, substantially less with static RAM – capable of running that high level

language. And, given the ability of the high level virtual machine to go stand by in a fine grained manner when awaiting events, its advantages in terms of energy budget will increase as the performance demands of the application decrease and the duty cycle becomes sparser concomitantly.

4. Conclusion

We believe GreenArray technology can solve most, if not all, application problems that are within its capability at a lower cost in energy than will any other existing computer architecture. Evidence to the contrary is always of interest!

Greg Bailey is President of GreenArrays, maker of multicore processors with integrated peripherals. For more information, please visit our website: http://www.GreenArrayChips.com