Low-Power Digital Processor for Wireless Sensor Networks

by

Daniel Frederic Finchelstein

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

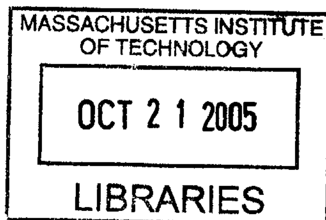
June 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 9, 2005

Certified by..
Anantha Chandrakasan
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by..
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
OCT 2 1 2005
LIBRARIES

BARKER

Low-Power Digital Processor for Wireless Sensor Networks

by

Daniel Frederic Finchelstein

Submitted to the Department of Electrical Engineering and Computer Science
on May 9, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

In order to make sensor networks cost-effective and practical, the electronic components of a wireless sensor node need to run for months to years on the same battery. This thesis explores the design of a low-power digital processor for these sensor nodes, employing techniques such as hardwired algorithms, lowered supply voltages, clock gating and subsystem shutdown. Prototypes were built on both a FPGA and ASIC platform, in order to verify functionality and characterize power consumption. The resulting $0.18\mu m$ silicon fabricated in National Semiconductor Corporation's process was operational for supply voltages ranging from 0.5V to 1.8V. At the lowest operating voltage of 0.5V and a frequency of 100KHz, the chip performs 8 full-accuracy FFT computations per second and draws 1.2nJ of total energy per cycle. Although this energy/cycle metric does not surpass existing low-energy processors demonstrated in literature or commercial products, several low-power techniques are suggested that could drastically improve the energy metrics of a future implementation.

Thesis Supervisor: Anantha Chandrakasan
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I am highly indebted to Professor Anantha Chandrakasan and to the members of Ananthagroup whose valuable input has definitely enhanced the significance of this thesis. I would also like to thank my family and Tarik, the most recent addition to it, for their love and unconditional support. Finally, I would like to thank National Semiconductor Corporation, which provided us with fabrication facilities and valuable technical support.

Contents

1 Introduction **13**

1.1 Individual Contributions 13

1.2 Background - Wireless Sensor Networks 14

1.3 Problem Description 16

1.4 Previous Work 17

2 Low-Power Processor Architecture **19**

2.1 Sensor Processor Architecture 19

2.2 Distributed Memory Architecture 20

2.2.1 Case 1: Memory Hierarchy for the DSP 21

2.2.2 Case 2: Memory Hierarchy for Specialized Processing Modules 22

2.3 Example Sensor Node Application 24

3 System Components **27**

3.1 Programmable Digital Signal Processor 27

3.2 Scalable Real-Valued FFT 29

3.3 Memory Subsystem 31

3.4 Direct Memory Access Block 32

3.5 Radio Interface 35

3.6 ADC Interface 35

3.7 Debugging Interface 37

3.8 Summary of System Components 37

4 Design Tools and Methodology 39

4.1 System requirements 39
4.2 Clocking strategy 40
4.3 FPGA implementation 41
4.4 Standard cell library characterization 42
4.5 ASIC implementation 43
4.6 ASIC Fabrication and Test Setup 45
4.7 Design for Testability 45

5 Results and Analysis 47

5.1 Functionality 47
5.2 Leakage Power Measurements 48
5.3 Dynamic Power Measurements 50

6 Future Improvements 53

6.1 Clock Gating 53
6.2 Memory Power Improvements 56
6.3 Sub-threshold Supply and DVFS 57
6.4 Layout for reducing interconnect loads 58
6.5 Level-converting pads 60

7 Conclusions 63

A Sample Source Code and Design Scripts 65

A.1 Sample DSP C source code 65

List of Figures

1-1 Wireless sensor network tracks location of moving vehicle location. . . 14
1-2 Generic architecture of a wireless sensor network node. 15

2-1 Architecture of an energy-efficient sensor processor. 21
2-2 Memory architecture using a cache to reduce access energy. 22
2-3 Instruction fetch energy for 3 different cache configurations. 23
2-4 Total SRAM access energy for 3 different memory configurations. . . 24
2-5 Example dataflow inside a sensor node. 25

3-1 Pipelined architecture of RISC DSP. 28
3-2 Automatic scaling of operands at each FFT stage to handle overflow. 30
3-3 Two different FFT saturation algorithms run on a 190Hz tone signal. 31
3-4 Interconnection of on-chip SRAMs. 32
3-5 State sequence of DMA engine. 34
3-6 Radio interface block 35
3-7 ADC interface block. 36

4-1 Synchronizer for crossing clock domains. 41
4-2 Astro view after placing standard cells. 44
4-3 Die photo of fabricated ASIC. 45
4-4 The PCB layout of the test board. 46

5-1 Core leakage power with all clocks turned off - Measured and Fitted. 49
5-2 Core energy/cycle with fft.c program running and performing continuous FFT computations. 51

6-1 Standard implementation of an enabled flip-flop. 54
6-2 Clock-gated implementation of an enabled flip-flop. 55
6-3 Variation of total energy/cycle with supply voltage excluding SRAMs and wire parasitics. 58
6-4 Effect of Wire Load on Energy/Cycle. 59
6-5 Effect of Wire Load on Maximum Frequency. 60
6-6 Differential Cascode Voltage Swing Logic (DCVSL) level-converter. . 61
6-7 Standard sense-amplifier flip-flop [18]. 62

List of Tables

1.1 Energy and functionality of existing low-energy processors 18

5.1 Measured leakage power versus simulations at typical and fast corners 50

6.1 Effect of wire parasitics on simulated critical path and dynamic cell power 59

Chapter 1

Introduction

This thesis describes the analysis and design of a low-power digital processor for wireless sensor network nodes. This chapter introduces the reader to the current research in wireless sensor networks and the key design challenges. Section 1.2 describes wireless sensor networks in detail along with some of the existing and potential applications. Section 1.3 describes the motivation for the object of the thesis, namely to design an energy-efficient processor that would prolong the lifetime of sensor networks and thus improve their viability. Section 1.4 provides a survey of the relevant work that has taken place in this field of research. These different sources were used as a reference and also as motivation for the work I present, and will later be used as benchmarks against which to compare the results of this project.

1.1 Individual Contributions

The digital sensor node processor for the μAMPS project includes a programmable DSP, FFT hardware accelerator, DMA engine, and interface blocks to an off-the-shelf radio and ADC. Several people were involved in the design and implementation of this project. My role was to assemble and test the entire system, and demonstrate its full functionality in both FPGA and ASIC platforms. Nathan Ickes designed the DSP and corresponding compiler, Alice Wang designed the original scalable FFT module ([11]), Denis Daly designed the radio interface, and Naveen Verma designed the ADC

interface.

1.2 Background - Wireless Sensor Networks

Wireless sensor networks allow for spatially and temporally dense environmental monitoring. Variables that can be tracked and interpreted include sound, light, vibration, motion, heat, radiation, chemical content, as well as many others. These embedded networks can be employed in a variety of domains, such as transportation, medical monitoring, precision farming, battlefields, factory machinery, air quality control, seismic detection, wildlife control, etc [2]. An example application is featured in Figure 1-1, where the sensor networks calculates the location of a moving vehicle.

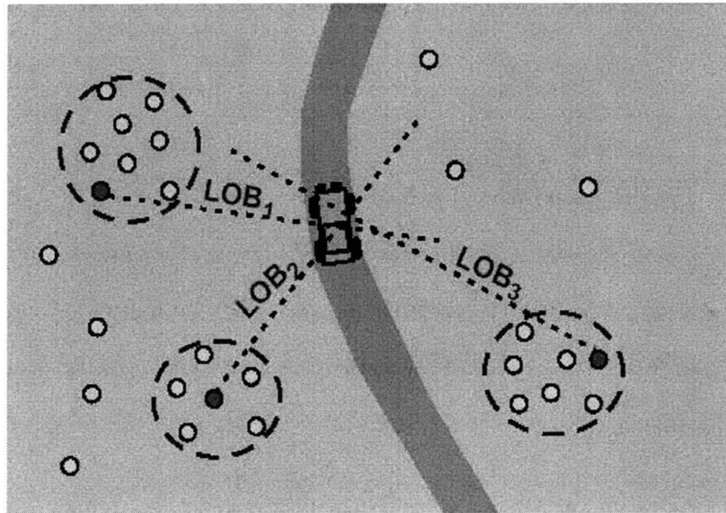


Figure 1-1: Wireless sensor network tracks location of moving vehicle location.

Wireless sensor networks can include hundreds to tens of thousands of untethered nodes, having no external data or power wires. The individual nodes, whose architecture is shown in Figure 1-2, fulfill at least one of the following main functions:

- sensing variables in the environment and converting them to an electrical signal
- processing the sensed data in either the digital or analog domain
- communicating wirelessly with other nodes in the network

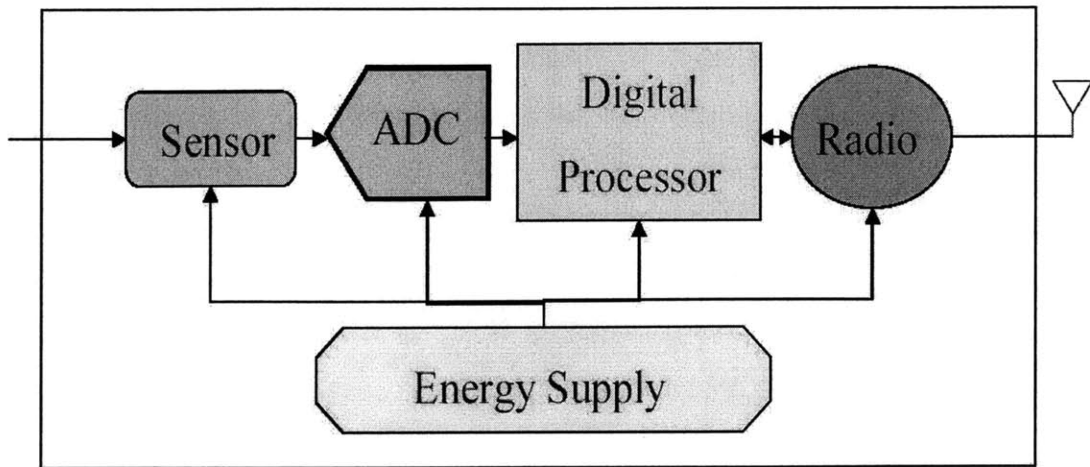


Figure 1-2: Generic architecture of a wireless sensor network node.

Due to the large number of nodes, the network must be configured in an ad-hoc fashion, according to self-configuring protocols such as the one described in [13]. These protocols must be flexible enough in order to tolerate new or failing nodes. In a simple network, the goal of each node could simply be to forward all the sensed data to a common, wired, base station, thus requiring only sensing and communication capabilities. However, the resulting communication costs would negatively affect the efficiency of a large network, due to the significant number of the nodes involved and the redundant nature of the continuously sampled data. A more intelligent network would first process the sensed data at each node (or cluster of nodes) and initiate a communication stream only when it deems that an event of interest has taken place.

The quality of wireless sensor networks can be evaluated based on several criteria. First, the aggregate results compiled by the network must accurately reflect, within an expected margin of error, the true state of the sensed environment. In the case of event monitoring, the network should minimize the number of false positives or missed events. Second, the useful lifetime of the network must be long enough to make its deployment beneficial and financially viable. If the lifetime of the individual nodes was only on the order of days, the cost of complete network deployment would be incurred every few days. For example, a reasonable lifetime of a sensor network that

tracks wildlife movement would be a year or two. A third desirable property is that the network is robust enough to handle the continuous addition and/or subtraction of nodes. Finally, the network should be reconfigurable to allow the user to change the detection settings at any time.

There are several technologies that can be employed to build cost-effective sensor networks. Sensors are needed to provide the individual nodes with the ability to read and digitize environment variables. These sensors must be as small as possible and provide the user with a tradeoff option between power consumption and accuracy. Low-power and fully-integrated digital signal processing algorithms should process the sensed data using the available energy budget. Finally, wireless communication circuits and protocols should be developed for low bit-rate and near-distance radios.

1.3 Problem Description

Wireless sensor networks can be made up of thousands of battery-powered nodes. Over a long period of time, these nodes may run out of energy and thus be rendered unable to perform their intended functions. One way to extend the lifetime of the network is to periodically replace the batteries of drained nodes or simply deploy additional nodes. However, due to the large scale of such networks, the cost of frequent hardware replacement and deployment may prove prohibitive.

Another solution would be to equip each node with devices such as solar cells that are able to harvest the ambient energy from the environment and use it to power up the electronics. Other forms of energy harvesting include thermal gradients, radio-frequency and mechanical vibration ([1]). If the environment of the node is able to provide more energy than the amount required by the electronic circuitry, the node can function indefinitely thus making the network completely self-sufficient.

In order for sensor networks to become a viable technology for a wider range of applications, the individual nodes must maximize the amount of sensing, communication and computation they perform, given the limited energy resources available from the battery and energy scavenging. In this thesis, I propose a sensor digital processor

architecture that attempts to minimize the energy consumption per computational task and thus extend the useful lifetime of the sensor node. The node's data sensing and communication energy must also be minimized, but those topics are not dealt with in this thesis.

1.4 Previous Work

Several techniques for low-power digital design have been demonstrated and documented in literature over the last couple of decades. In [10], the author describes how to use voltage scaling along with parallelism and pipelining in order to reduce the dynamically switched energy for a constant throughput multimedia application.

The throughput constraint of typical signal processing applications [5] can be greatly relaxed in the case of sensor networks, while still allowing the nodes to accurately detect events of interest that may occur in the environment. This observation enables an even further reduction in the supply voltage, to the point where the devices are operating in the sub-threshold region. An FFT processor employing this technique is demonstrated in [11], where the lowest-energy operating point occurs when the supply voltage is 0.35V and the frequency is 20KHz.

An ultra-low energy sensor node microcontroller named Dust, presented in [4], consumes about 12pJ per instruction. This processor runs off a supply voltage of 1.0V, has a 12-bit datapath, has 10Kbits of on-chip SRAM, and uses a 0.25μm technology. The main power reduction techniques it employs are component-level clock-gating, subsystems that can be shut off independently, and guarded ALU inputs.

In [14], an asynchronous approach is taken to designing a low-power DSP, named SNAP. This processor executes instructions in groups of event handlers, which are generated by the expiration of timers or the gathering of new sensor data. When no events are pending, the processor is completely idle, and only consumes leakage power since the clockless asynchronous circuit style enables the equivalent of perfect clock gating. SNAP has 32Kbits of on-chip data memory and an equal amount of data memory. In simulation, SNAP can operate off a 0.6V power supply, while consuming

about 24pJ/instruction at 28MIPS.

Several companies offer energy-efficient processors targeted for low-performance, low-power applications such as sensor networks. Atmel's 8-bit microcontroller ATmega168/V ([19]) has 128 Kbits of instruction memory, 8Kbits of data memory, and operates up to 10MHz with a 2.7V power supply. Its datasheet claims 432 pJ/cycle at 1MHz, 844 pJ/cycle at 32KHz, and 0.18μW of idle power. Another popular low-energy processor is Intel's PXA255 ([20]) implemented in 0.18μm technology, which contains a 32-bit RISC ARM core running at 33-400MHz, along with 256Kbits of on-chip instruction SRAM and an equal amount of data cache. According to its datasheet, PXA255 consumes 455pJ/instruction when running at 33MHz with a supply voltage of 1.0V, and draws 45μW during sleep mode.

The relevant statistics of the processors described above are listed together for clarity in Table 1.1. The sensor node processor which I explore for my thesis attempts to improve on the figure of merit of the processors presented in [4] and [14], by providing hardwired algorithms, optimizing the memory architecture, operating at near-threshold supply voltages, and employing other methods described in Chapter 6.

Table 1.1: Energy and functionality of existing low-energy processors

Name	*Width [bits]*	*On-chip SRAM Size [Kbits]*	*Energy/instruction [pJ]*
DUST [4]	12	10	12
SNAP [14]	16	64	24
ATmega168/V [19]	8	136	432
PXA255 [20]	32	512	455

Chapter 2

Low-Power Processor Architecture

As previously stated, a wireless sensor node must perform three main functions: sensing, processing and communication. In an integrated system, the processing element must be equipped with an interface to an analog-to-digital converter (ADC) to sample the sensed data; in addition, an interface to a wireless radio is required to facilitate communication with other nodes. This chapter describes the architecture of the low-power sensor node processor. A justification is given for the choice of hardwired algorithms and distributed memory architecture.

2.1 Sensor Processor Architecture

The sensor node can be implemented as several modules that are designed to facilitate the common tasks performed by sensor network nodes. Key tasks that can be implemented in hardware include the Fast Fourier Transform (FFT), Finite Impulse Response (FIR) filters, encryption, source coding, channel coding/decoding, compression/decompression, and interfaces to the radio and sensor. In order to achieve energy efficiency throughout the entire system, the hardware modules can use independent voltage supplies (with values as low as a threshold voltage) and operate at different clock frequencies. The drawbacks of this architecture versus using only a programmable DSP are the significant increase in system complexity and area, the need for additional data transfers between the DSP and specialized modules, and the

difficulty of inter-operability across different voltage and clock domains.

Figure 2-1 shows the proposed architecture for an energy-efficient sensor node. The digital architecture contains a basic programmable DSP that executes arbitrary programs. The DSP communicates with the specialized modules through a shared bus, with the DMA scheduling the transfer of data between modules and the bus. Data memory is accessible by both the specialized modules and the DSP. Dynamic Voltage and Frequency Scaling (DVFS) can be used to dynamically control the performance of each module and tradeoff lower energy for higher computational latency. In the first μAMPS implementation, the variable frequency and voltage supplies are off-chip, although they could be integrated in a future version. Each module's supply voltage should be set to the lowest possible value that still satisfies its speed requirement. However, there is a supply voltage below which computations become less energy efficient due to leakage currents [11]. When no computation is taking place, the supply voltage should be completely shut off from the CMOS logic in order to reduce leakage power.

2.2 Distributed Memory Architecture

A considerable percentage of the energy used by a sensor network processor is spent in moving data and instructions between on-chip memories and the processing units. This energy can be reduced by dividing the memory hierarchy into a large store (the main memory) and a smaller local store (the cache). This division has been traditionally made in order to reduce memory access time, but we can apply similar concepts to reduce memory access energy. The use of a cache, or local memory buffer, is illustrated in Figure 2-2. This architecture helps reduce average access energy as the cache accesses are performed over a smaller bus, and also since the access energy of the cache is smaller than that of the main memory.

However, the use of the cache memory hierarchy introduces an extra overhead due to the energy required to perform additional memory transfers. As illustrated by the following two examples, this trade-off must be carefully evaluated in order to

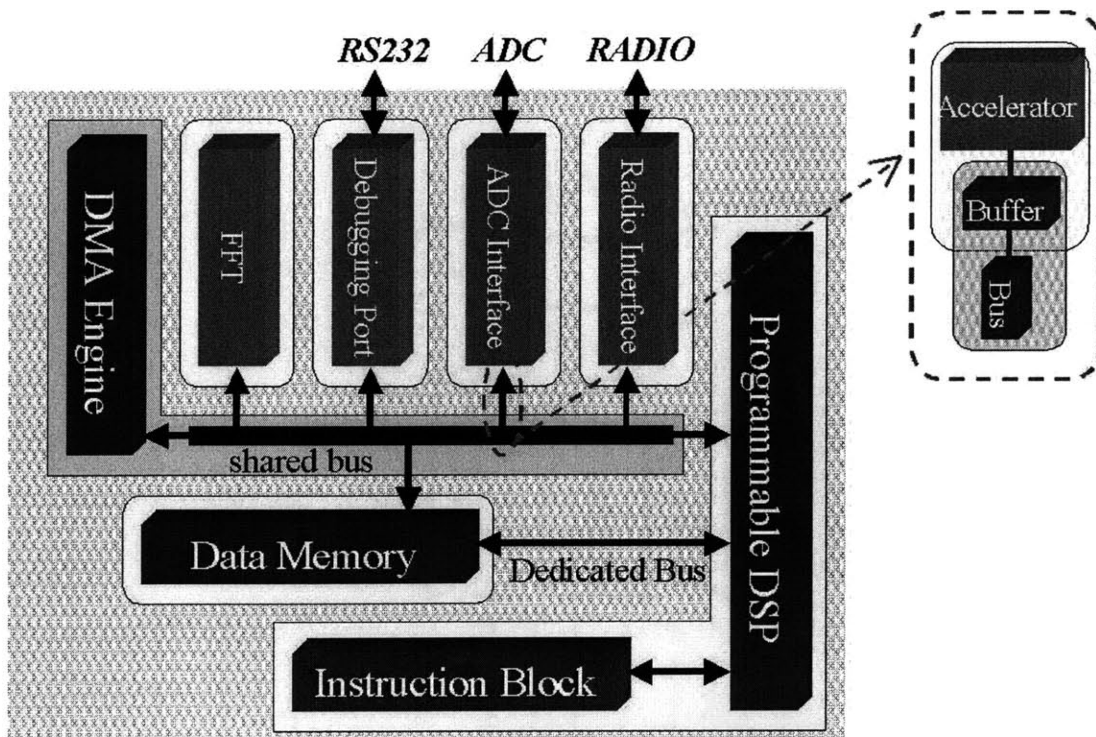


Figure 2-1: Architecture of an energy-efficient sensor processor.

determine the optimal memory hierarchy. In the first case, we consider the memory hierarchy used by the DSP to access its data and instruction memories. For the second scenario, we try to determine whether the specialized hardware modules can also take advantage of a cache hierarchy when accessing the data memory. The following analysis uses, in both cases, parameters given by the datasheets of automatically-generated SRAMs of three different sizes: 8096x16, 1024x16, and 128x16 bits. Thus, the study considers a main memory of 8096 words, with two potential cache sizes of 128 or 1024 words, all memories being implemented as SRAMs.

2.2.1 Case 1: Memory Hierarchy for the DSP

Consider a main instruction memory of size 8096 words, which the DSP accesses either directly or through an instruction cache of either 128 or 1024 words. When no cache is used, the energy per fetch is constant, and the hit rate is always 100%. When an instruction cache is used, the hit rate depends not only on the size of the

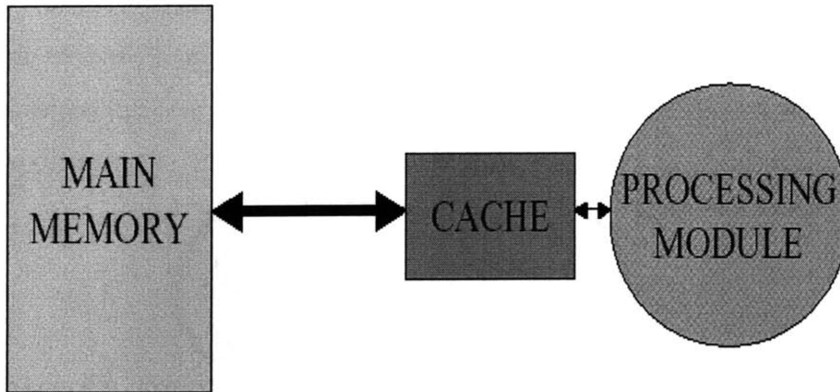


Figure 2-2: Memory architecture using a cache to reduce access energy.

cache (128 or 1024 words), but also on the nature of the code being executed on the DSP. Once a hit rate for each of these cache sizes is calculated or simulated, we can use this metric to determine whether employing a cache at all is beneficial for the energy used to fetch instructions.

As shown in Figure 2-3, a 128-word cache with a hit rate higher than 80%, or a 1024-word cache with a hit rate higher than 85%, are both more energy-efficient than having no cache at all. The comparison between the two caches must be done by determining their individual hit rates as derived from some benchmark applications. For the first μAMPS implementation, the typical application hit rates were not yet available, so we decided to not use an instruction cache.

2.2.2 Case 2: Memory Hierarchy for Specialized Processing Modules

A specialized processing module is likely to offer an energy reduction over a general-purpose processor, as the latter requires an overhead for fetching and decoding instructions. Since these modules will perform computations on a fixed subset of the main data memory, it is worth examining whether they can also benefit from a similar cache structure. The two alternatives are to either have the specialized modules access the data memory directly, or create a local cache store containing all of the

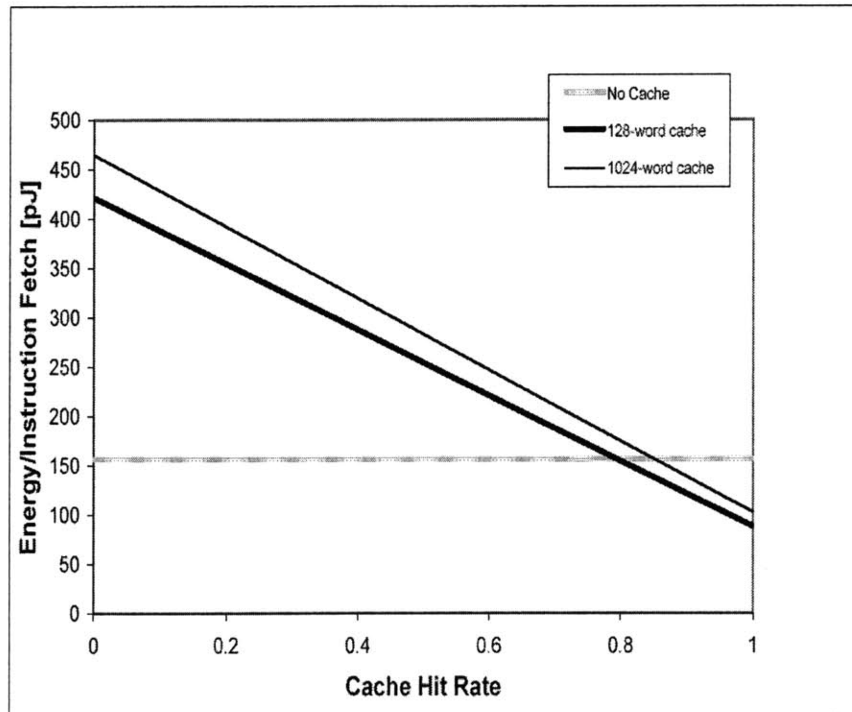


Figure 2-3: Instruction fetch energy for 3 different cache configurations.

data the algorithm needs to access.

The first choice implies that each memory access in the algorithm will be directed to a larger SRAM, and will have to take place over a larger, more loaded bus. This alternative also forces all the processing modules to run at the same clock speed, thus denying the use of fine-grained DVFS at the module level. The second choice alleviates these issues, but introduces the overhead required to transfer the inputs and outputs of the algorithm over the main bus. We study once again an 8096-word main memory, and use two local-store alternatives: 1024 or 128 words, as determined by the type of algorithm.

From Figure 2-4, we observe that if the algorithm operates on a 1024-word block, a local cache provides energy advantages only if it performs more than 6700 memory accesses. Similarly, for an algorithm with 128 input words, a local cache is beneficial when the algorithm does more than 700 memory accesses. As an example, the FFT processor of [11] does about N*log(N) memory accesses, where N is the size of the

input block. It is therefore better for the FFT processor to use a local data cache, since a 1024-point FFT performs 10240 memory accesses, while a 128-point FFT performs 896.

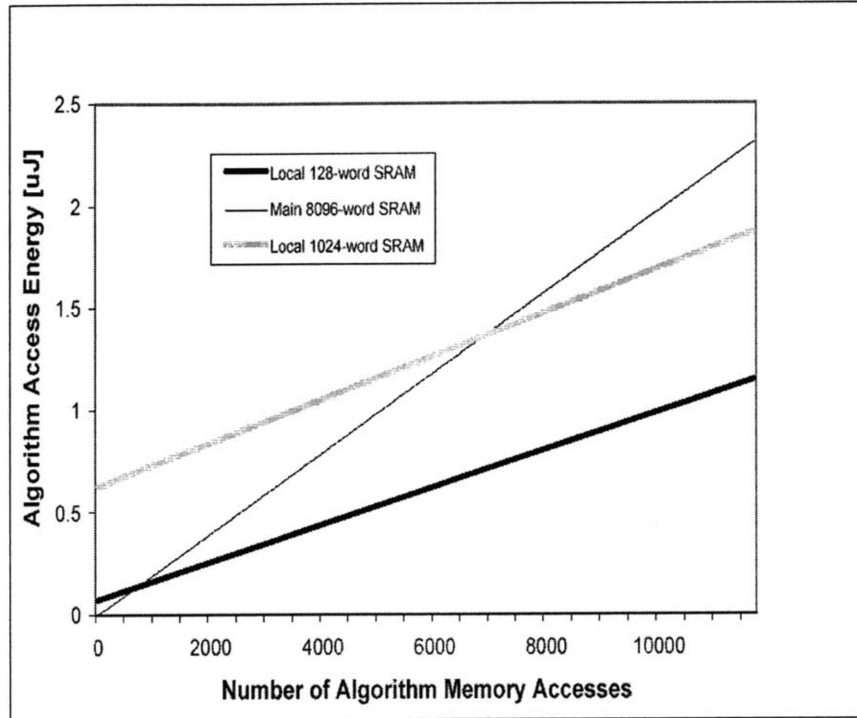


Figure 2-4: Total SRAM access energy for 3 different memory configurations.

2.3 Example Sensor Node Application

The flow of data through the proposed digital system can better be understood through observing a simple application. Consider a sensor node equipped with a microphone and whose role is to detect an acoustic signature that exceeds a set threshold at some given frequency. The operation of the sensor node is illustrated in Figure 2-5. An ADC samples the microphone's electrical output and the digital time-domain samples are then fed into the ADC interface buffers on the processing node. Subsequently, the DMA engine transfers a window of digital samples into the memory buffers of the FFT co-processor. Once the FFT algorithm completes, the

DMA engine transfers the frequency-domain data into the data memory of the DSP. The DSP searches for the peak frequency and raises a flag whenever a given threshold is exceeded. It then forms a radio packet containing the detected information and places it in its data memory. Next, the DMA engine moves the packet into the memory buffer of the radio interface. The radio is then free to transmit its packet once it detects the airwaves are not being used.

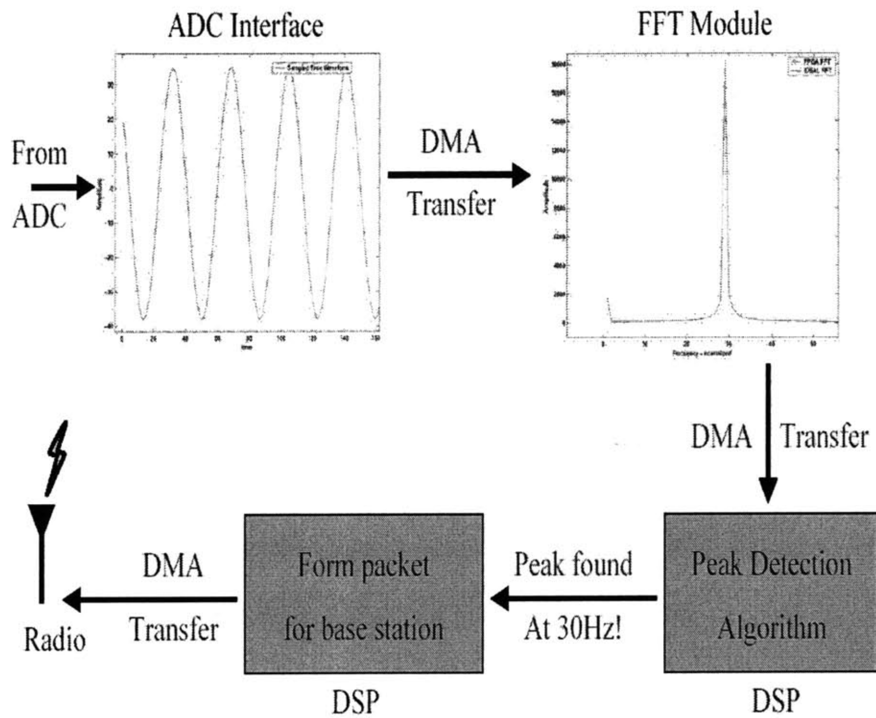


Figure 2-5: Example dataflow inside a sensor node.

Chapter 3

System Components

This chapter provides a detailed description of the system's building blocks, along with some rationale for the design adopted.

3.1 Programmable Digital Signal Processor

The core of the sensor node's processing is the programmable DSP designed by N. Ickes. This is a RISC-style processor with 16-bit instructions and data, which implements a custom instruction set. The user of the DSP can simply write his program in standard C language. Some sample C source code can be found in Section A.1. A customized compiler, based on the open-source GNU C Compiler (GCC), reads in the source files and produces machine-specific object code that can be downloaded into the DSP's instruction memory.

The hardware organization of the DSP is structured as a five-stage pipeline, as illustrated in Figure 3-1. During the first stage, the DSP reads the instruction from the instruction SRAM and places it in its instruction register (IR). In the second pipeline stage, the instruction from the IR is decoded, thus telling the DSP what type of operation should be performed, as well as which registers to read out from the register file. At the beginning of the third stage, the register contents and immediate operands (stored in the instruction itself) become available and are processed by the arithmetic unit (adder, shifter, multiplier, etc). The multiplier takes 1 cycle and

stores the most significant half of its double-word output in a co-processor register that can be further accessed by regular instructions. The read operations from or write operations to data memory are executed during the fourth stage. The DSP has separate data and instruction memory buses, so that this stage does not conflict with the instruction fetch stage. Finally, during the fifth - and last - stage, the results of the arithmetic unit or the memory read are written back into the register file.

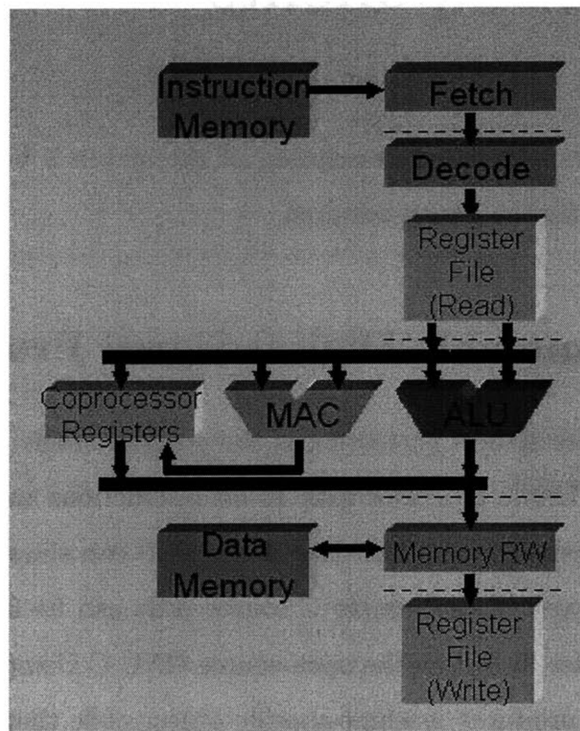


Figure 3-1: Pipelined architecture of RISC DSP.

The DSP has eight interrupt lines that could trigger the execution of an interrupt handler if the program enables them. These interrupt lines can be used to signal the receipt of a radio packet, a serial port input, the completion of a FFT, the availability of ADC samples, or can simply be triggered by external events generated through the GPIO pins.

3.2 Scalable Real-Valued FFT

To demonstrate the energy advantage of implementing algorithms directly in hardware, rather than through DSP instructions, an FFT co-processor was built. This module [11] is flexible in trading off energy efficiency for higher computation accuracy. In the case of FFTs, higher accuracy translates to performing arithmetic on higher-precision operands (FFT precision), and using more time-domain inputs (FFT points) to obtain a more exact spectral representation of the signal. An increase in FFT precision raises the amount of energy taken by additions, multiplications and memory accesses, whereas additional input points require a larger-than-linear increase in the total number of computation cycles. This design allows the sensor node processor to dynamically choose an FFT precision of either 8 or 16 bits, as well as an input FFT length of 128, 256, 512 or 1024 points.

Several changes to the original design were implemented in order to correct some inaccuracies and to facilitate the design of the memory architecture.

The first such change was related to the sequence of the memory accesses. The FFT algorithm operates in and out of the same SRAM buffer, meaning that at the beginning the memory it is filled with the input points; during execution it holds intermediate computation values; and in the end it holds the results (equal in number to the input points). The previous FFT controller performed a read and a write to each memory segment on every cycle. This necessitated the use of a dual-port SRAM, where both ports were tied to the FFT block. In addition, another port running at a different clock rate was required for interfacing with the rest of the system since the FFT has a separate clock. Since SRAMs with 3 ports are impractical to design, we decided that we could get rid of the additional port by alternating between reads and writes to the FFT memory. This led to a doubling of the latency of the FFT algorithm, but allowed for a simpler memory design.

The second change that was necessary was related to the handling of saturating inputs. The FFT algorithm performs fixed-point additions and subtractions during every cycle, which can lead to overflow or underflow. If left uncorrected, the MSB

of the resulting value will be simply ignored, eventually leading to largely incorrect results. One approach to deal with these overflows is by saturating the outputs of such computations to the maximum value. An FFT algorithm cycles through several stages as it transforms the inputs from time domain to frequency domain. The solution we chose for the overflow problem was to store an extra bit for all the computations during a stage, and thus store all intermediate results in memory with this additional bit. If overflow occurred for any of the results during a FFT stage, this additional bit would be a 1 and a flag would thus be raised. During the next stage, when the data is read out, it is all conditionally divided by 2 through a shift operation depending on the occurrence of an overflow in the previous stage, as shown in Figure 3-2. The drawback of this approach is the need to store an extra bit, leading to a 6% memory size overhead.

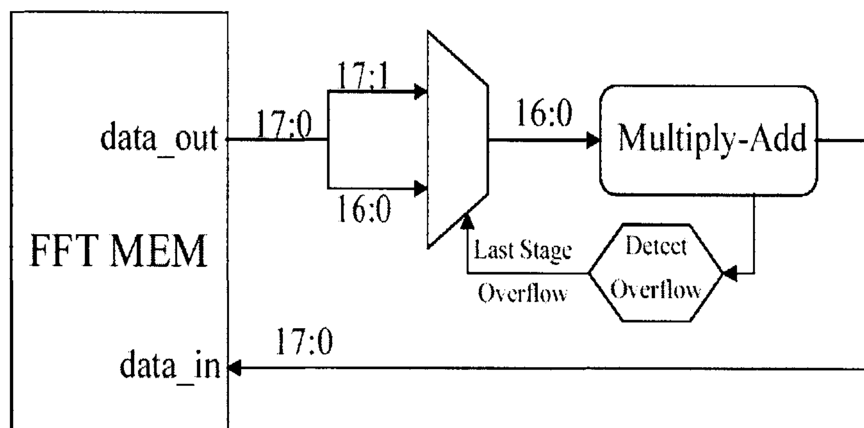


Figure 3-2: Automatic scaling of operands at each FFT stage to handle overflow.

Saturation of values during the FFT algorithm will occur most often when the input time-domain waveform is a single tone, as illustrated by Figure 3-3. Figure 3-3 also shows the results of the FFT algorithm when the two different saturation techniques are applied. As it can be observed, the second saturation algorithm leads to a nearly ideal FFT computation.

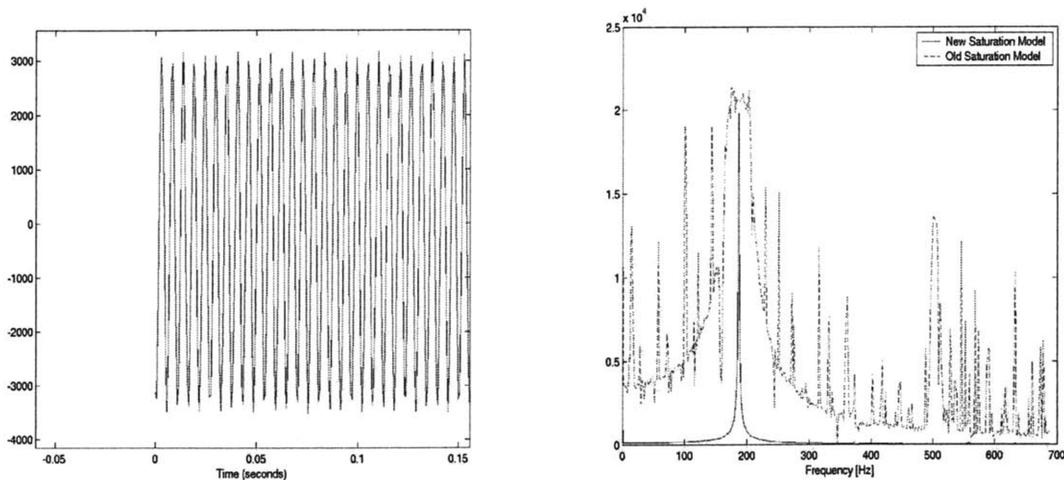


Figure 3-3: Two different FFT saturation algorithms run on a 190Hz tone signal.

3.3 Memory Subsystem

The μAMPS processor is equipped with several memory blocks, which can be grouped into three categories: instruction memory, data memory, and specialized module buffers, as shown in Figure 3-4. All of the memory blocks are implemented using single-port or dual-port synchronous SRAMs provided by the technology vendor. The interface of each memory port is composed of clock, address bus, input data bus, output data bus, write enable, output enable, memory enable, power and ground.

The instruction memory has a single port and can hold 2048 16-bit instructions. It can be read by the DSP over the instruction memory bus, and can be preloaded at startup time through the DMA engine. The instruction memory runs off the system clock and its read address is generated by the DSP's program counter. The data memory has two read/write ports, one for the DSP data bus, and one for the DMA bus, respectively. This memory can hold 16384 16-bit words; it is segmented into 16 1024-word blocks in order to allow individual power-down of memory blocks when these are not needed. The power switches for these blocks help to reduce standby leakage current and are planned for a future version. Each of the hardwired modules (FFT, ADC interface, radio interface) contains a buffer that can hold 1024 16-bit

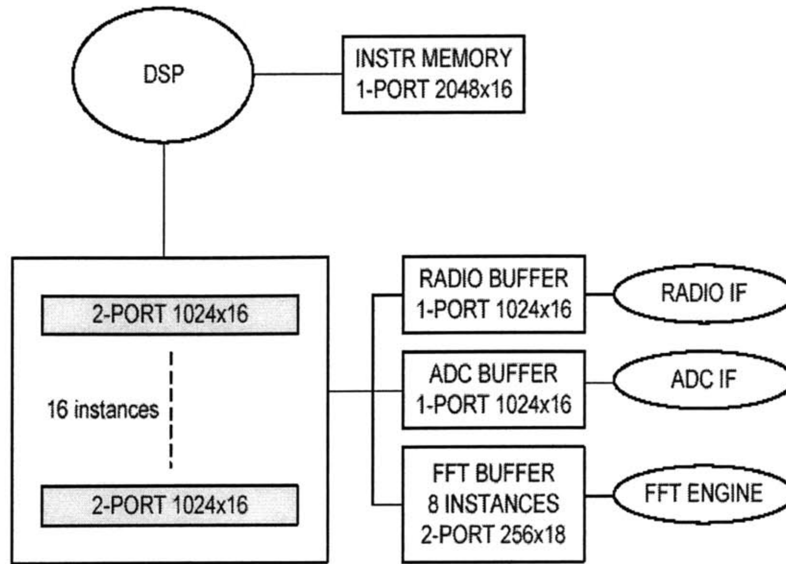


Figure 3-4: Interconnection of on-chip SRAMs.

words. One of the ports communicates with the DMA engine using the system clock, whereas the other port uses the clock of the local module. This design would allow the FFT block, for example, to operate using its own clock as dictated by a fine-grained DVFS controller described in Section 6.3.

3.4 Direct Memory Access Block

The μAMPS digital system needs to be able to move large blocks of contiguous data amongst its different processing units. For example, the data captured in the ADC interface buffer must be moved into the FFT input buffer, while the FFT results must be moved into the DSP's data memory in order to be post-processed. One of the options considered here was to have all the memory in the system memory-mapped to the DSP's address space, and execute a load and store instruction in the DSP for each word that needs to be transferred. This would cause the DSP to incur an overhead of 2 instructions and 4 cycles for each word transfer, while also rendering the DSP unavailable for useful processing for the duration of a block transfer. The increase in energy due to extra instruction fetching and decoding, the large transfer

latency and the unavailability of the DSP all make memory-mapping an inefficient option.

The solution we chose to address this problem was to have a DMA engine independently generate the memory control signals used to implement data transfers. The types of transfers supported by the DMA are listed below:

- DMEM to FFT: Transfer a block of input data or configuration words from the main data memory to the FFT buffer.
- FFT to DMEM: Transfer a block of FFT results from the FFT buffer back to main memory.
- ROM to DMEM: Transfer a block of test data from an off-chip parallel ROM to main memory.
- ROM to IMEM: Transfer the program instructions from the off-chip ROM to the instruction SRAM.
- DMEM to UART: Send a block of ascii-formatted 16-bit data words from main memory to a PC for debugging.
- UART to DMEM: Load a block of test data from a PC into main memory.
- UART to IMEM: Load the program instructions from a PC into the instruction SRAM.
- DMEM to ADC: Send a set of configuration words from main memory to program the ADC sample rate, sample width, etc.
- ADC to DMEM: Transfer the block of sampled data to main memory.
- DMEM to RAD: Send a data packet or set of configuration words from main memory to program the radio's packet length, transmit power, etc.
- RAD to DMEM: Transfer the data from a receive packet into the main memory.

The DMA block is implemented as a state machine, which goes through the following sequence in an endless loop, as shown in Figure 3-5. Initially, the DSP determines the size of the data block, its starting address in data memory, and the corresponding memory buffer to transfer to/from. The DSP programs this information into the DMA, which performs the necessary data transfers over the bus. During the transfer, the DSP is free to continue normal operation; once the DMA is done, an interrupt is issued to the DSP to alert it that the transfer is complete. The DMA is able to generate these control signals using less energy than the DSP, since it consists of very simple hardware such as counters to generate both the addresses and the next DMA states.

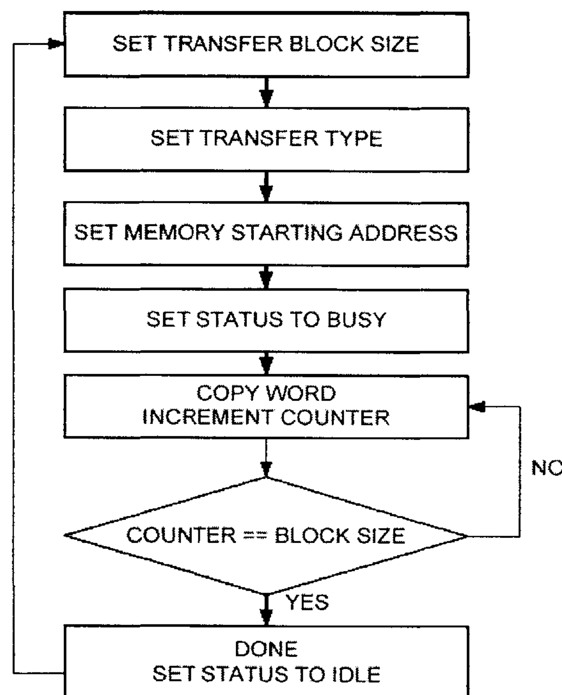


Figure 3-5: State sequence of DMA engine.

3.5 Radio Interface

The wireless sensor node communicates with other nodes or a base station by sending or receiving packets through a wireless radio, the latter being implemented in this system as an off-chip Chipcon1010 radio [22]. The radio interface block (Radio IF), illustrated in Figure 3-6, is used as the interface between the DSP and the Chipcon 1010. When transmitting, the DSP writes the entire packet into the Radio IF buffer. The Radio IF then serializes this packet and sends it over a SPI link to the Chipcon radio. On the receive path, the Radio IF forms a packet in its buffer from the received serial bit stream, and raises an interrupt to the DSP when a full packet has been received. Future versions of the Radio IF will offload more computation from the DSP by including local MAC protocol processing, such that the DSP transmits and receives only raw data and no wrapper information.

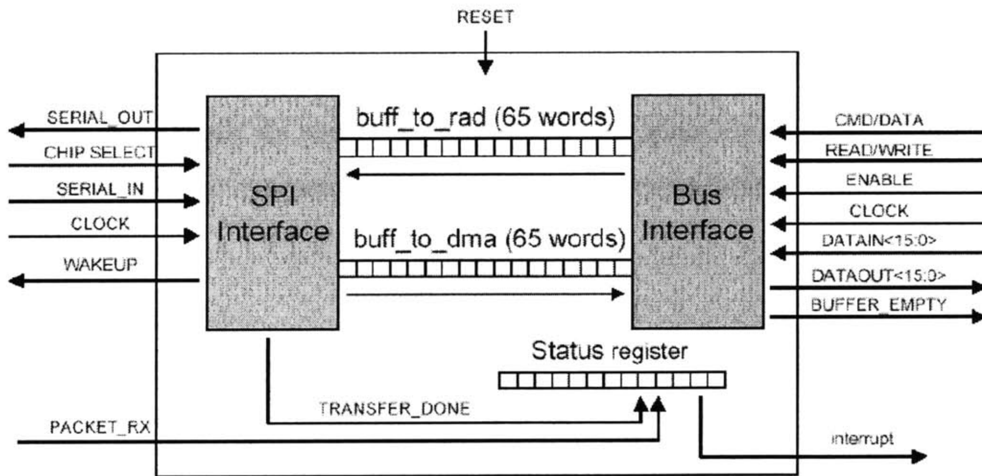


Figure 3-6: Radio interface block

3.6 ADC Interface

When the sensor node processor wishes to take some samples of its acoustic environment, its ADC samples the analog voltage output of a microphone, digitizes this

data to a 16-bit value, and feeds this data word back to the processor. The task of interacting with the ADC is quite simple, so it is best handled by some simple control hardware in order to increase energy efficiency and not overload the DSP unnecessarily. This task is the role of the ADC interface. This module is designed to communicate with the off-the-shelf ADC in [23], although the interface is simple enough to function correctly with other parallel ADCs.

The ADC interface is configured by the DSP with the desired bit precision (8 or 16 bits), number of samples, and sample rate (relative to system clock). The controller then issues a convert pulse at the start of each sample period, after which it stores the new data in the local buffer. Once all of the samples have been captured, the ADC interface issues an interrupt to the DSP, which instructs the DMA to transfer all of the samples into data memory. Since this block of samples is usually just forwarded on to the FFT block, the ADC interface also has a single-sample mode, where each new sample is directly forwarded to the FFT's memory over the DMA data bus, as shown in Figure 3-7. This avoids the redundancy of writing the sampled data in the ADC buffers and data memory, and also reduces the need for the DSP to issue double the DMA transfers (ADC to data memory, and data memory to FFT).

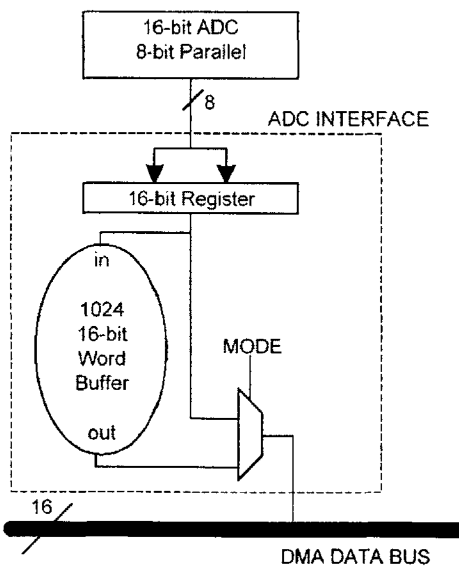


Figure 3-7: ADC interface block.

3.7 Debugging Interface

A debugging interface is critical to the development of the system hardware and also to understanding its real-time behavior. There are several facilities that were designed into the system which provide debugging capability, such as a serial UART, a general-purpose input/output (GPIO) parallel port, and a parallel trace port.

The UART interface follows the protocol described in [24] and is used to communicate to a computer that is a running a program such as HyperTerm. In this fashion, the DSP can display 16-bit words on a connected PC, or it can receive some input data or instructions from the same PC. The parallel GPIO port can be used by the DSP directly to load 16-bit data into and out of the system at the same speed as the system clock, and is useful when connected directly to a pattern generator and logic analyzer. Finally, the trace port is used to selectively output the program counter or instruction word in the form of 16-bit data, as directed by two trace control lines.

3.8 Summary of System Components

This chapter described the main components of the first μAMPSsystem implementation. The goal of this system is to have a full-featured sensor node processor, while optimizing the individual components for energy efficiency.

Chapter 4

Design Tools and Methodology

This chapter outlines the design methodology that was employed to build the sensor node processor, along with a description of the CAD tools necessary for a full-system implementation.

4.1 System requirements

The starting point was to determine the set of sensor network applications that the processing node must support. Since the focus was on an acoustic sensor node, we needed to run algorithms such as line-of-bearing, acoustic identification, and trip-wire. The line-of-bearing algorithm ([6]) uses time-domain samples from several microphones in order to triangulate the position of a sound-emitting object of interest, such as a tank on a battlefield. Acoustic identification is used to determine the type of object emitting the sound by examining its frequency signature and comparing it against other known patterns. The trip-wire algorithm parses the time-domain samples to determine whether any sound-emitting object is in the vicinity, in which case it alerts the sensor node to go into a more accurate and advanced sensing mode. It is clear that in order to run all of these algorithms, the sensor node must include at least a programmable DSP, which must be equipped with enough data memory to store the data handled in the algorithm, and enough instruction memory to hold a compiled version of the algorithms. Since the acoustic signal processing algorithms

often rely on the frequency content of a sound, the FFT algorithm will be executed often, and thus benefits from being implemented in a separate, hardwired algorithm. This set of modules, along with the required control and interface modules described in Chapter 3, constitute the hardware of the sensor node.

4.2 Clocking strategy

Since performance in sensor networks is deemed less important than energy efficiency, it was determined that the target system clock frequency would be 1MHz, which allows the sensor node to perform about 80 1024-point FFT computations per second. At this frequency, the core supply voltage can be lowered significantly to reduce dynamic switching power (Equation 4.1), as will be shown through measurement and simulation in Section 5.3.

$$P_{switching} = C_{switching} \cdot V_{supply}^2 \cdot f, \tag{4.1}$$

We decided to have the FFT module run off of a separate clock from the rest of the system, in preparation for a future implementation that will use DVFS. This will allow the DSP software to independently scale the voltage and frequency of each module to its most energy efficient point, as will be shown in Section 6.3.

The introduction of multiple clock domains complicates the design process. When a signal needs to cross clock domains, the circuit of Figure 4-1 should be used in order to prevent metastable signals from being fed as inputs to the circuit inside clock domain B. Since CLKA and CLKB have no frequency or phase relation, there will invariably occur a setup or hold timing violation whenever FFB1 is triggered by CLKB, thus causing the output of FFB1 to become metastable. However, the latter signal has a full CLKB cycle to settle to a stable value until it is sampled by FFB2, thus guaranteeing that the output of FFB2 will be stable with an extremely high probability. As an example, the mean time between failures (MTBF) for this synchronizer circuit is on the order of 10 years ([21]).

When using synchronization for a bus of signals that crosses multiple clock do-

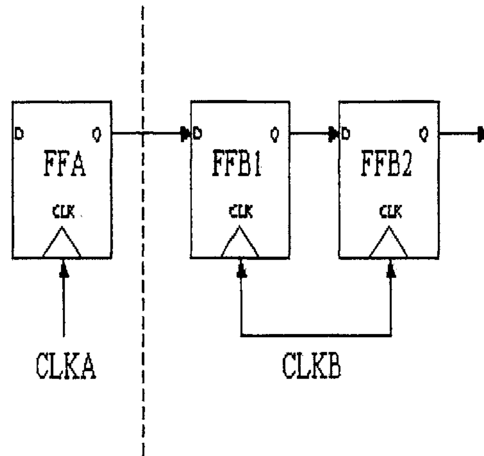


Figure 4-1: Synchronizer for crossing clock domains.

mains, it is possible that the bits will not all arrive during the same CLKB cycle. In this case, it is necessary to wait until the slowest possible path has propagated through before using the data bus on the new clock domain. Another scenario that occurs quite often in multi-clock systems such as the μAMPS processor is when a stream of data needs to be moved from one clock domain to another. In the μAMPS system, this was accomplished using dual-port SRAM buffers, where one port was written to using CLKA and the other port read from using CLKB. However, when the source and sink of the data are active simultaneously, a better solution is to use a smaller asynchronous fifo whose read and write pointers are incremented using two separate clocks, as described in [25].

4.3 FPGA implementation

With the system hardware and constraints identified, the system was implemented in Verilog, a register-transfer-level (RTL) hardware description language and simulated to verify functionality. Several Verilog simulators are commercially available, all providing similar functionality and performance. The simulators used throughout this project were Cadence Verilog-XL, Synopsys VCS, and Mentor Graphics ModelSim.

Once the Verilog code was stable and system-level simulations passed, we decided to build a full prototype of the whole system. The complete sensor node was assembled by realizing the digital logic in a field-programmable-gate-array (FPGA) and connecting it to discrete components. The peripheral off-the-shelf components connected to the FPGA were the microphone sensor of [28], the 16-bit ADC of [23] and the radio of [22]. This platform allowed us to explore and improve full-system dynamics, while highlighting any potential system-level design faults.

The FPGA design was equipped with Chipscope, a built-in logic analyzer described in [27], allowing for real-time debugging of the FPGA system and thus greatly reducing the time to identify logic or timing faults. The FPGA platform used was the Xilinx Virtex-2 FPGA from [26], which has 6 million logic gates, 2.5 Mbits of block SRAM, and 144 18x18 bit multipliers, making it more than suitable to implement the logic of the sensor node. When compiled together with the Chipscope logic, the entire design occupied 33% of the FPGA's configurable logic and 72% of the available on-chip SRAM. Without the logic analyzer soft core, the design took up only 20% and 22%, respectively. The Verilog logic was fully debugged and verified on the FPGA platform; however, in order to achieve a low-power system, it was necessary to build the digital logic into an application-specific integrated circuit (ASIC).

4.4 Standard cell library characterization

An ASIC is a collection of logic gates of various complexity, connected together to perform a digital logic function. These gates, known as standard cells, are distinguished by the logical function they implement (AND, NOR, Flip-Flop, Latch, Mux, Full-Adder, etc) and their drive strength (the ability to drive capacitive loads). In order to easily and accurately predict the timing and power consumption of the logic functions implemented using these gates, the standard cells are simulated extensively in Hspice. The outputs of these simulations are summarized in lookup tables that are stored inside the standard-cell synthesis library. This library is used by later design tools to compute path delays and power consumption.

The standard cell library provided by the vendor was characterized only for the nominal supply voltage of 1.8V. Additionally, we used Cadence Signalstorm ([36]) to characterize the standard cells at various other operating voltages, down to 0.3V. This was useful for the analysis done to characterize the circuit's power and maximum frequency over the entire supply voltage range, described in Section 6.3. The vendor's library was used in the μAMPS ASIC, since the Signalstorm characterization tool was not available at the time of design.

4.5 ASIC implementation

This section describes the tool flow that was used to build the standard-cell ASIC given a Verilog design and a well-characterized standard-cell library. Two sets of tools were important in this process: one group for generating a layout that could be fabricated, and another for performing the verification to guarantee a working silicon chip.

The following tools were used to produce the final layout, in the order given below:

- Synopsys Design Compiler - DC_shell ([33]) - synthesized the RTL logic to produce a gate-level netlist of standard cells.
- Synopsys Astro ([34]) - performed standard-cell place-and-route (PAR) to produce the layout of Figure 4-2.
- Synopsys Hercules ([35]) - performed layout-versus-schematic (LVS) to ensure a match between the final layout and final schematic.
- Synopsys Hercules ([35]) - performed design-rule-checks (DRC) to ensure fabrication rules are not violated and design-for-manufacturability (DFM) rules are followed.

The following tools were used to verify the correctness of the final layout and to characterize its power consumption:

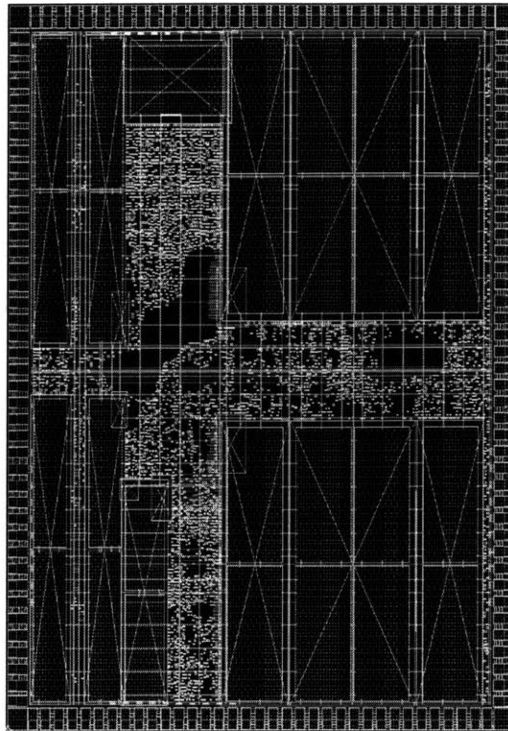


Figure 4-2: Astro view after placing standard cells.

- Synopsys Formality ([29]) - compared the final gate-level netlist against the original RTL Verilog and ensured they are functionally equivalent.

- Synopsys PrimeTime ([30]) - performed timing verification on all register-to-register paths inside the final gate-level netlist to ensure there are no hold or setup violations.

- Cadence Verilog-XL - performed back-annotated gate-level functional and timing simulations.

- Synopsys Star-RCXT ([31]) - extracted the resistive and capacitive parasitics from the layout so they can be used in subsequent timing and power simulations.

- Synopsys Nanosim-VCS ([32]) - performed top-level Hspice-like timing and power simulations of the final layout with adjustable trade-offs between simulation speed and accuracy.

4.6 ASIC Fabrication and Test Setup

Upon completion and verification of the final layout, the ASIC was manufactured in a National Semiconductor 0.18μm CMOS process with 5 metal layers, as shown in Figure 4-3. The chip has an area of 20mm^2 (5mm x 4mm), and is made up of 3.5 million transistors including the on-chip SRAMs.

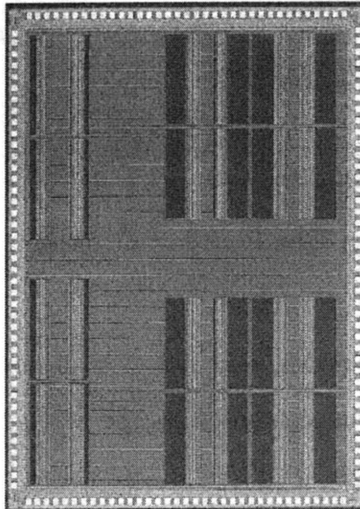


Figure 4-3: Die photo of fabricated ASIC.

In order to test the fabricated ASIC, a test PCB was built using the drawing software PCAD2001. The resulting layout is shown in Figure 4-4, and the slot for the μAMPS chip can be seen in the middle. Other than the sensor node ASIC, this board included several discrete parts (Flash ROM, 16-bit ADC, Chipcon Radio, RS232 connector), as well as test ports to facilitate the use of a pattern generator and logic analyzer. The μAMPS DSP was programmed through either the ROM or the RS232 serial port, and the C compiler was used to test out different applications.

4.7 Design for Testability

There are several test structures that can be built into the ASIC in order to make the fabricated chip more suitable for testing and verification. Some of the more important design-time testability techniques include the use of scan-chained registers,

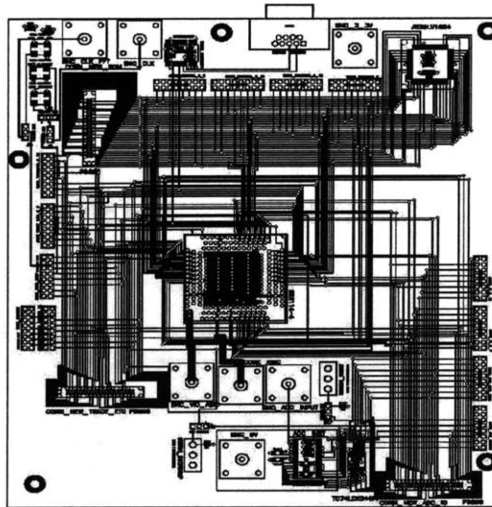


Figure 4-4: The PCB layout of the test board.

process-characterizing test circuits, and isolation of power domains. Although these test methodologies were not implemented on the first μAMPS ASIC, they are recommended for any future designs.

Chapter 5

Results and Analysis

This chapter lists results from the fabricated chip and compares these measurements to expected values from simulation.

5.1 Functionality

The first tests of the setup in Section 4.6 were used to verify the functionality of individual ASIC blocks. This section describes the results of these functional tests, most of which were successful, and provides some possible causes for the tests that failed.

- The C program of Section A.1 was compiled, burned into the instruction ROM, and successfully run on the μAMPS ASIC. This confirmed that the following main blocks were functional: DSP, FFT, instruction/data SRAMs, DMA engine, ROM interface, and GPIO outputs.

- The ADC interface was verified to work correctly by feeding a single tone (see Figure 3-3) into the TI 16-bit ADC and plotting the captured data that was dumped through the GPIO outputs.

- The FFT block was run with different configurations of block size and bit precision and the results matched the values simulated in Verilog.

- The UART interface worked correctly when sending data words from a PC to the μAMPS ASIC and vice versa. However, the μAMPS ASIC could not load its instructions stream through the UART interface, although this mode functioned correctly in the FPGA prototype, as well as during full Verilogand Nanosim-VCS simulations.

- The GPIO pins functioned correctly as outputs, but were unusable as inputs due to an easily correctable mistake made during the generation of the IO pads.

- The trace pins worked correctly and allowed us to monitor in real time the values of the program counter and instruction register.

- The radio interface between the sensor node and the Chipcon Radio was verified to work as designed, allowing us to receive and transmit packets correctly.

- A timer program was written to toggle the GPIO pins once every second. This confirmed that the DSP's timer and corresponding interrupt signal function correctly.

The μAMPS ASIC was tested and verified at supply voltages ranging from 0.5-1.8V, and clock frequencies below 5MHz, much higher than the targeted operation frequency of 1MHz.

5.2 Leakage Power Measurements

This section describes the leakage power measurements taken of the μAMPS ASIC using the test setup of Section 4.6. The measured data is compared to expected or simulated values and possible sources of mismatch are described.

The graph of idle power (with all clocks off) versus supply voltage is shown in Figure 5-1, as measured at room temperature (25°C). When the supply voltage V_{dd} is the nominal 1.8V, the μAMPS core consumes 880μW of leakage power, while at the lowest operating V_{dd} of 0.5V, the core uses up only 26μW to maintain the full system state.

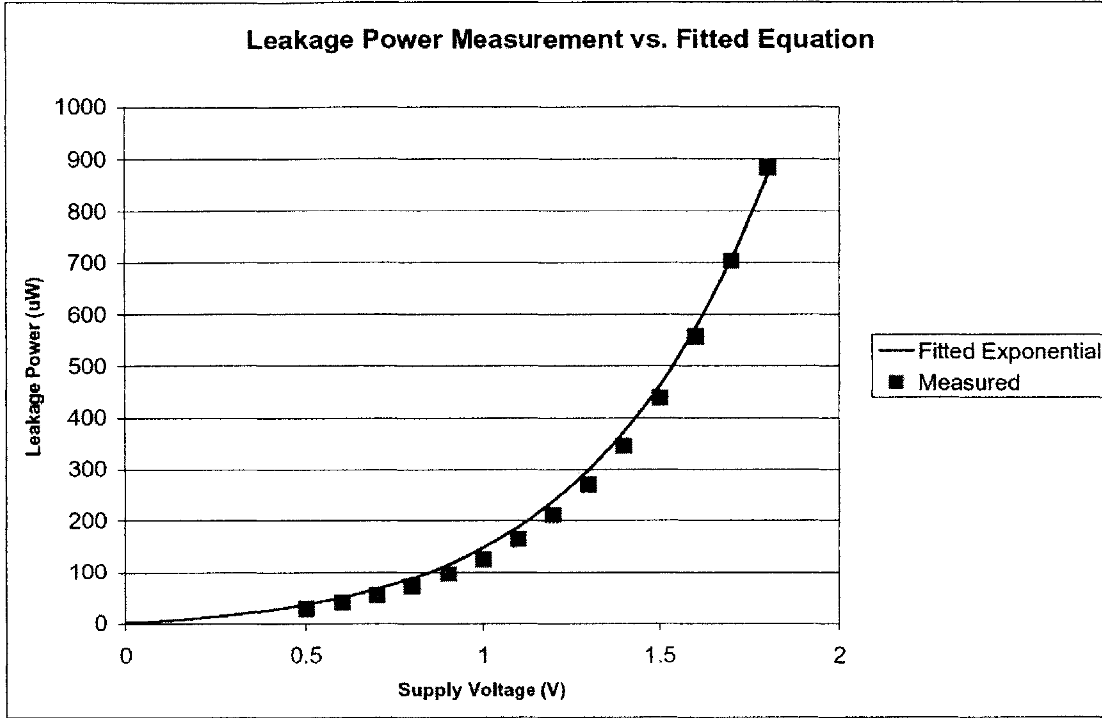


Figure 5-1: Core leakage power with all clocks turned off - Measured and Fitted.

The μAMPS system was simulated using Hspice and Nanosim-VCS in order to compare the leakage power with real measurements. These simulations use a form of the BSIM equation ([12]) for the leakage current through a transistor, shown in Equation 5.1.

$$I_{leak} = Ke^{\frac{V_{gs}-V_{th}-\gamma V_{sb}+\eta V_{ds}}{nV_T}}[1-e^{-\frac{V_{ds}}{V_T}}], \quad (5.1)$$

Where:

- K is a technology parameter
- V_{gs} is the gate-source voltage
- V_{th} is the nominal transistor threshold voltage
- V_{sb} is the substrate bias voltage (γ is a corresponding fitting parameter)
- V_{ds} is the drain-source voltage (η is a corresponding fitting parameter)

- n is proportional to the subthreshold slope (mV/dec)
- V_T is the thermal voltage

Equation 5.1 can help to explain the dependence of leakage current on the supply voltage. In the simple case when all clocks are off and there is no stacking effect where two transistors in series are off, the leakage power can be expressed as a function of supply voltage V_{dd} as shown in Equation 5.2. For $V_{dd} >> V_T$, Equation 5.2 has an exponential form, and can be fit very closely to the measured leakage power, as shown in Figure 5-1.

$$P_{leak} = V_{dd} K e^{\frac{nV_{dd}-V_{th}}{nV_T}} [1 - e^{-\frac{V_{dd}}{V_T}}] \quad (5.2)$$

The previous discussion focused on the behavior of leakage power as the V_{dd} value is varied. An important design issue is how accurately the absolute power values are predicted by the simulation and BSIM models. The simulation results of the μAMPS ASIC are shown in Table 5.1, and were obtained using BSIM transistor models at 25°C. As it can be seen, the measured values always fall between the values simulated at typical and fast device corners.

Table 5.1: Measured leakage power versus simulations at typical and fast corners

V_{dd} [V]	*Measured Power [μW]*	*Simulated Power [μW] Typical Corner*	*Simulated Power [μW] Fast Corner*
1.8	880	259	1800
1.5	436	156	1048
1.0	123	56	366
0.5	26	17	109

5.3 Dynamic Power Measurements

This section describes the power measurements taken of the chip while the input clocks are running. The active switching power of the ASIC is compared to simulated or estimated values, and potential sources of mismatch are described.

A test C program that continuously performs FFT computations using the FFT block was written and run on the μAMPS ASIC. The two clocks were both set to 100KHz and the program operation was verified to be correct through the GPIO pins. The total current through the core was measured for different supply voltages and converted to an equivalent energy per cycle metric, which is plotted in Figure 5-2. To obtain a frequency-independent measure of the active energy/cycle, the total leakage power with clocks off was first subtracted from the total power during program execution. This is because leakage power is constant, so leakage energy/cycle varies with frequency. As can be expected, the measured energy/cycle plot can be fitted quite well with a quadratic equation of the form $E = C_{eq} \cdot V_{dd}^2$, as shown in Figure 5-2. The total energy/cycle for the entire chip was also estimated for two different supply voltages using accurate simulations in Nanosim-VCS. The two estimated energy points can be seen to fall within 10% of the measured values.

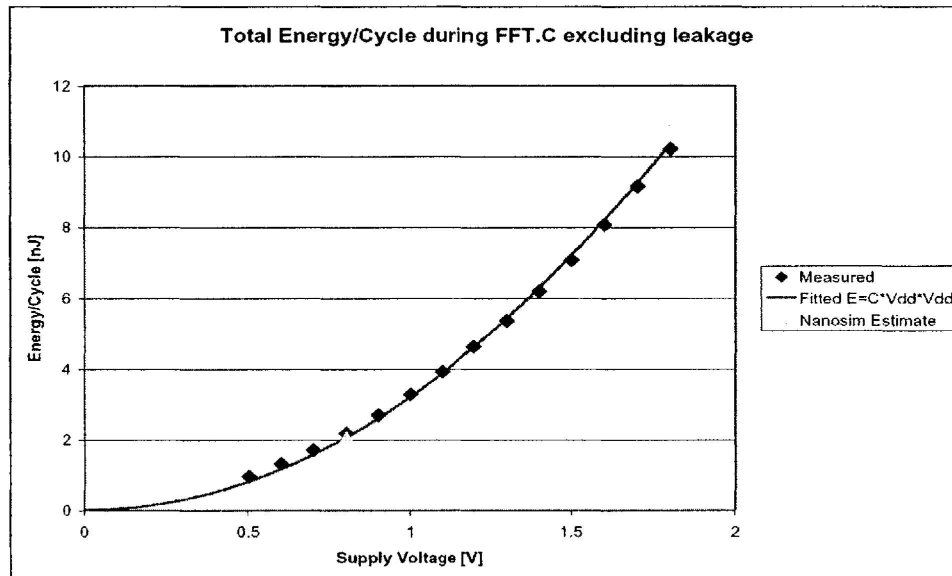


Figure 5-2: Core energy/cycle with fft.c program running and performing continuous FFT computations.

Chapter 6

Future Improvements

The μAMPS system presented in the previous chapters can be improved in several ways for energy efficiency. These optimizations are planned for a future version of the μAMPS chip. This chapter describes these proposed techniques, as well as provide low-level simulation comparisons to justify the change in design.

6.1 Clock Gating

In a synchronous digital design, a common global clock is used to feed all the state-storing registers. These registers are conditionally updated on every clock cycle as determined by an input enable signal. A straightforward implementation of an enabled flip-flop is shown in Figure 6-1. While the input enable is active high, the flip-flop stores the input value at every positive clock edge. When the flip-flop is disabled, each clock-edge causes it to actively store its previous value, thus retaining its state. For this design, the flip-flop can consume almost the same amount of power whether it is enabled or disabled, since the clock causes a lot of the internal nodes to toggle.

Ideally, the flip-flop should consume no power while disabled, while simply maintaining its state using the underlying static feedback circuitry. To accomplish this, the clock would only conditionally toggle, thus saving the power needed to charge up the clock port and the internal power of a triggered flip-flop. One way to achieve

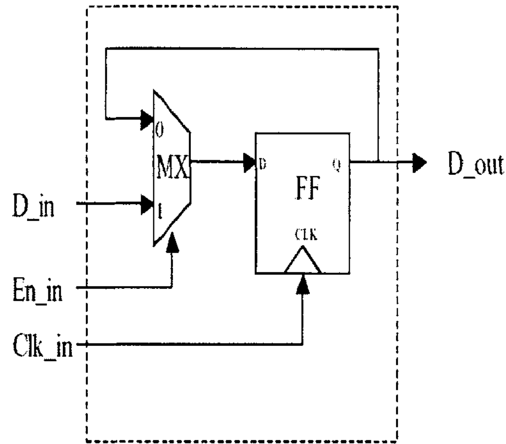


Figure 6-1: Standard implementation of an enabled flip-flop.

this is to use a glitch-free, clock-gating flip-flop as shown in Figure 6-2. If the enable signal stays low for a few cycles, the gated clock remains low and does not switch during that entire period. This saves the internal power of the flip-flop driven by the gated clock. The clock port of the flip-flop no longer needs to be charged, but we must now charge the clock port of the latch and the input to the AND gate. However, since the enable signal and corresponding clock-gating circuitry can often be shared among several flip-flops that make up a wide register, the additional overhead of the latch and AND gate is amortized. The latch of Figure 6-2 is necessary in order to ensure that a glitch on the enable signal does not propagate to a glitch on the gated clock signal ([33]). This situation is prevented by latching the enable signal while the input clock is low, when a glitch on the enable signal would not be able to propagate through the AND gate.

Register-level clock-gating can be performed automatically during the synthesis part of the design flow described in Section 4.5. In order to gauge the energy savings offered by register-level clock gating, the μAMPS design was synthesized and laid out using clock-gating cells as in Figure 6-2. The resulting layout was extracted and simulated using Nanosim-VCS with a testbench where both the DSP and FFT blocks were active and all clocks ran at 5MHz. The simulation results showed that

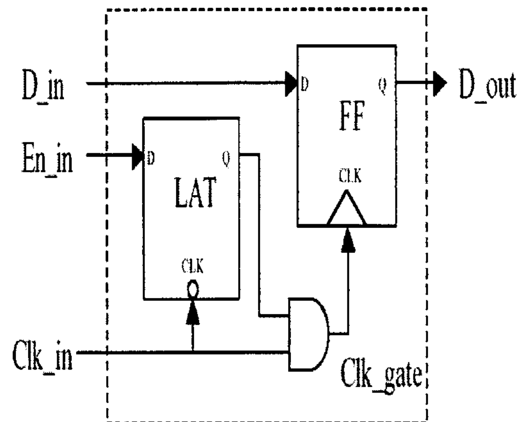


Figure 6-2: Clock-gated implementation of an enabled flip-flop.

clock-gating helped reduce the total logic power consumption (not including SRAM accesses) from 7.0 mW to 6.2 mW, a decrease of over 10%. This reduction is relatively low because most of the logic blocks were active and thus could not always be clock-gated. As the system activity rate decreases (ex: when FFT is not used), the reduction in switching power as a result of clock-gating will be even more significant.

To save additional clock power, a global enable signal can be used to gate the entire clock tree. In order to determine how our design would benefit from the use of such a signal, we can compare the switching energy during idle mode of a design that implements register-level clock gating versus one that employs perfect clock gating. The power with register-level clock gating can be simulated while reset is held high, thus ensuring that all enable signals are low and the logic has no activity. With the clocks running and the chip in reset mode, Nanosim-VCS results show that the total switching energy/cycle (SRAMs not included) amounts to 640 pJ/cycle. The optimal global clock gating during idle mode can be achieved by completely turning off the clocks, resulting in only standby leakage power, reported by DC_shell to be 69μW. When only some of the modules are active, module-level clock-gating using separate enable signals should be employed in order to prevent the clock from propagating to idle modules. With the unused clocks successfully gated, techniques for standby

leakage current reduction should be employed to lower the remaining idle power, as described in Section 6.2.

6.2 Memory Power Improvements

As discussed in Chapter 5, the SRAM blocks on the chip are all implicitly read from on every cycle, causing unnecessary power draw from the power supply. For example, while executing the program fft.c, at most 3 out of the 27 SRAM blocks need to be active for reads or writes. The Verilog code for the next version of μAMPS was modified to disable memory read ports when not active, which should cause the peak memory access energy to decrease by a factor of 9.

The μAMPS chip uses mostly dual-port SRAMs (24 out of 27) for on-chip SRAMs. This option was adopted in order to allow the DSP and DMA to have concurrent data memory access, as well as to synchronize the system clock and FFT clock domains. However, dual-port SRAMs have about twice as many transistors as single-port SRAMs, thus doubling both leakage current and silicon area. Since leakage current and long wire lengths are large contributors to the overall power consumption of the μAMPS chip, it is recommended that the next chip uses only single-port SRAMs. The concurrent DSP and DMA access to data memory can still take place, as long as it accesses different segments (the current data memory has 16 segments). Similarly, considering that the memory buffer between the DSP and the FFT blocks is not accessed at the same time from the two different clock domains, a single-port SRAM with a multiplexed clock would be sufficient as the FFT buffer.

Since the compiled SRAMs account for about 90% of the transistors on the μAMPS chip, they are also the major source of leakage current. The use of leakage-tolerant SRAM designs would allow for a drastic reduction in the total leakage current. The insertion of header or footer power-gating transistors between the SRAMs and the power supply can reduce leakage by a factor of 40 with negligible decrease in performance, as demonstrated in [15]. This technique, only applicable if multi-threshold transistors are available, can also be applied to the rest of the logic cells.

However, power gating usually causes complete loss of state information in the system's flip-flops and SRAMs. For the SRAM case, [16] shows that lowering the supply voltage from the nominal 1.0V to 0.35V, rather than shutting it off, leads to a more than 90% reduction in leakage power, while safely retaining the state of the SRAMs.

6.3 Sub-threshold Supply and DVFS

A reduction in switching power can be achieved through lowering the supply voltage of logic cells, down to the subthreshold region of operation. To enable this, we must create new standard cell libraries and memories that can function at these ultra-low voltages. These new cells must be accurately simulated at all possible operating voltages in order for timing verification to guarantee correct functionality. As the supply voltage is lowered into the subthreshold region, the circuit delays become so large that leakage energy begins to dominate the total energy used. Thus, there is an optimal V_{dd} that yields a minimal sum of active and leakage energies/cycle.

In order to find the optimal point for the μAMPS design, the standard cell libraries were recharacterized using Signalstorm for the following V_{dd} values: 1.8V, 1.5V, 1.0V, 0.6V, 0.5V, 0.4V, and 0.3V. The design was synthesized in DC_shell using each of the new libraries, and an estimate of active and leakage power was obtained using the report_power command. The power and delay of SRAMs and wires was not included in this study, the DC_shell default statistical switching activity was used when reporting active power, and the standard-cell libraries were characterized at 25°C. Figure 6-3 shows the resulting energy per cycle plot for this design. As it can be seen, for supply voltages above 0.5V, the switching energy dominates total energy, while leakage energy rises exponentially for subthreshold supply voltages. For this particular experiment, the optimal supply voltage was found to be 0.5V, which corresponds to the minimum operating voltage of the μAMPS ASIC.

In a sensor node processor, there is a tradeoff between extending battery lifetime, which tends to push circuit operation to the lowest possible supply level, and the

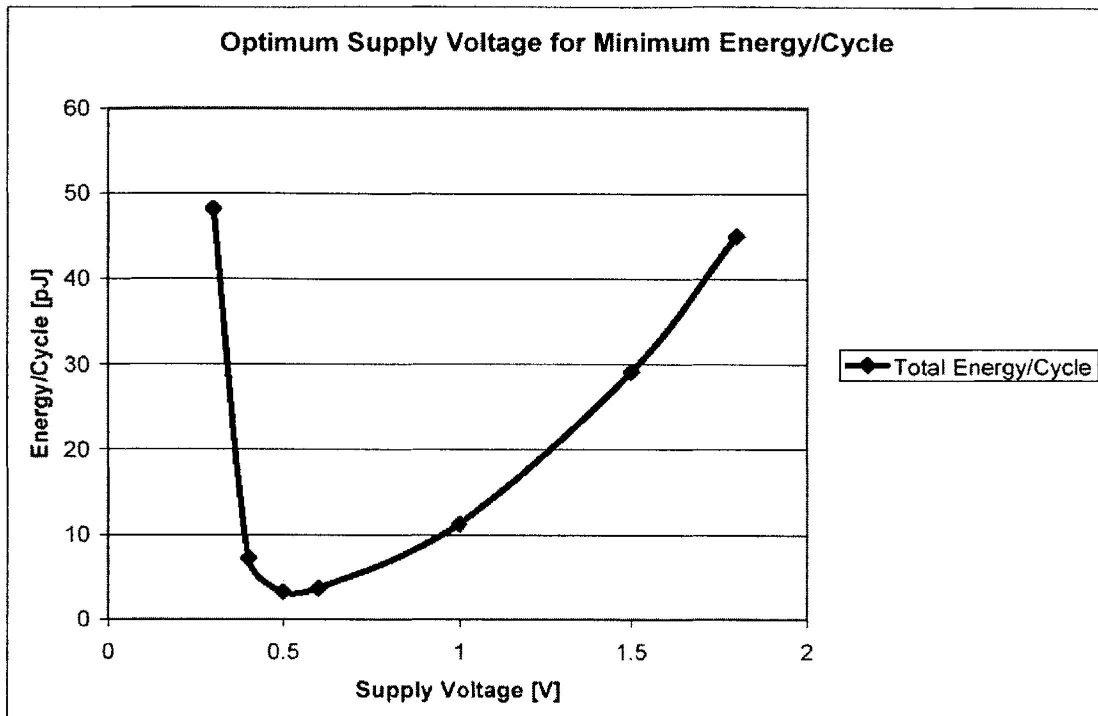


Figure 6-3: Variation of total energy/cycle with supply voltage excluding SRAMs and wire parasitics.

acceptable level of performance, which places a lower bound on the clock frequency and implicitly also on the supply voltage. The system software determines the minimum operating frequency for the executing application and sets the supply voltage as low as possible, all these without violating any of the timing paths inside the digital circuits. This scheme is known as Dynamic Voltage and Frequency Scaling (DVFS). A simple implementation of DVFS would use a lookup table to determine the operating voltage for a given frequency. Alternatively, a feedback loop would automatically adjust the supply voltage such that the circuit's critical path equals the desired clock period, as described in [17].

6.4 Layout for reducing interconnect loads

The charging up of wire loads make up a significant portion of total switching energy and delay for the μAMPS ASIC. To quantify the effect of wires, we performed a

DC_shell analysis of the post-layout design with and without wire parasitics included. Table 6.1 shows the critical path delay and dynamic switching power at 50MHz for different supply voltages, but not including the effect of SRAMs, obtained using the report_power command. The same data was converted to the equivalent energy/cycle and maximum frequency and is plotted in Figure 6-4 and Figure 6-5. As can be seen, wire loads account for 74-84% of total switching power and 10-24% of total path delay.

Table 6.1: Effect of wire parasitics on simulated critical path and dynamic cell power

V_{dd} *[V]*	*Power [µW] Without Wires*	*Power [µW] With Wires*	*Critical Path [ns] Without Wires*	*Critical Path [ns] With Wires*
1.8	3099	12143	14.9	16.6
1.5	1946	8216	17.7	19.6
1.0	740	3516	30.1	36.1
0.6	212	1210	160	204
0.5	131	823	527	696

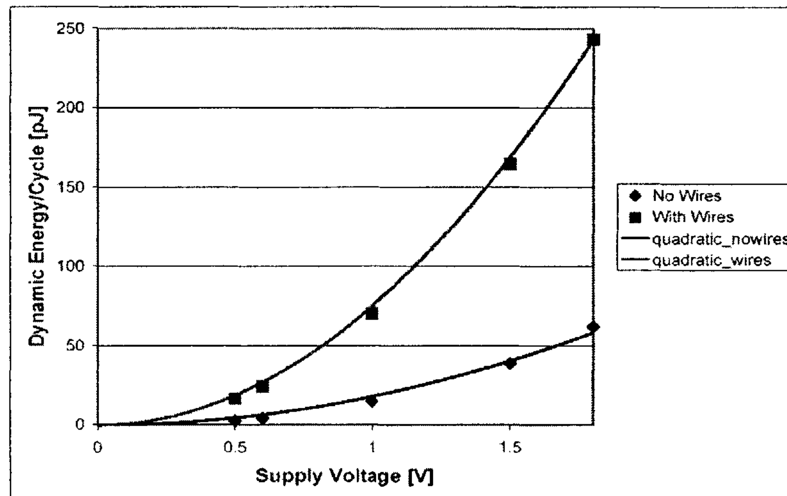


Figure 6-4: Effect of Wire Load on Energy/Cycle.

As a result of these observations, a significant effort should be placed in reducing wire loads. A good floorplan must be created during layout to place related components close to each other. The amount of fill wire insertion (to meet metal density

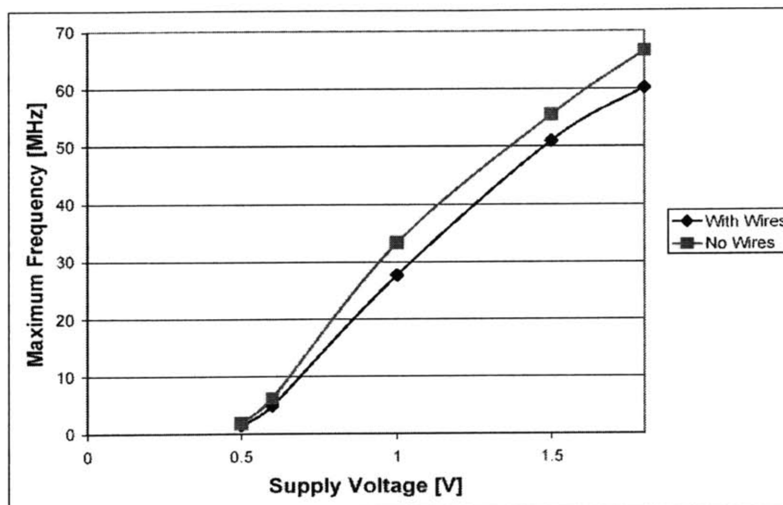


Figure 6-5: Effect of Wire Load on Maximum Frequency.

requirement) should be kept to a minimum in order to reduce unnecessary coupling between signal wires. For example, DC_shell estimates show that 7% of the total switching energy in the current design is due to additional capacitances caused by the insertion of floating fill wires in all the available wire tracks.

6.5 Level-converting pads

In the first implementation of the μAMPS chip, the core was designed to operate at a nominal voltage of 1.8V, while the output pads were designed to take a core swing as an input, and output a 0-3.3V signal. The reason for choosing a 3.3V I/O signal is to be able to correctly interface to other printed circuit board (PCB) components (ROM, UART, ADC, etc), which all run at this standard supply voltage. This was accomplished by the use of a level-converter circuit similar to the one shown in Figure 6-6, where a small differential input to the NMOS transistors causes a full differential swing on the output. At an I/O supply voltage of 3.3V, the DCVSL gate operates correctly as long as the input swing is above 0.9V. When the core supply voltage falls below 0.9V, the NMOS transistors become too weak due to their low overdrive voltage and are not able to fight the pull-up PMOS transistors, thus causing pad

failure.

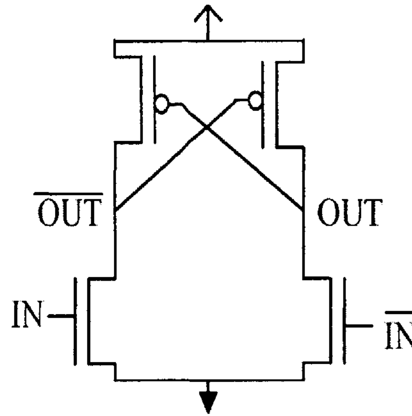


Figure 6-6: Differential Cascode Voltage Swing Logic (DCVSL) level-converter.

The μAMPS DSP core can operate correctly down to a voltage of 0.5V, at which point all the 3.3V output pads to malfunction. One possible solution to this problem is to lower the supply voltage to the I/O pads down to about 0.6V. At this core voltage, the DCVSL gate once again becomes operational due to the reduced overdrive on the PMOS transistors. The I/O supply was lowered to 0.6V during measurement in order to be able to verify correct functionality of the core down to near-threshold supply voltages. However, this limits the chip's functionality since it can no longer interface properly to the other off-the-shelf components in the system, which are designed to run at the standard 3.3V.

Another solution that allows the output pads to operate at a much higher voltage than the internal supply is to use a sense-amplifier flip-flop (SAFF) from [18], as shown in Figure 6-7. The CLK signal that feeds the SAFF swings to the full voltage of the I/O supply. When CLK is low, nodes N1 and N2 are equalized to a high value; this puts the output latch in a "hold" state where Q and $\overline{Q}$ retain their previous value. As soon as CLK transitions high, the path with the higher differential input will offer a less resistive current path to ground, and thus pull down either N1 or N2. As soon as enough of a differential voltage develops between N1 and N2, the two cross-coupled inverters will quickly regenerate N1 and N2 to complementary full-

swing values, thus storing a new state on the output latch. The voltage difference between the differential inputs D and $\overline{D}$ only needs to be larger than the input offset of the SAFF latch, which is usually smaller than 100mV.

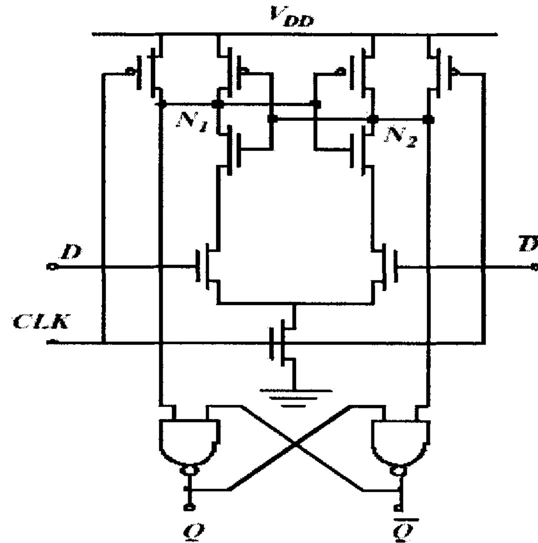


Figure 6-7: Standard sense-amplifier flip-flop [18].

In this setup, every output pad would be synchronized to the clock that was used to generate the signal, and would thus add one clock cycle of latency to the output path. Alternatively, a separate, faster, clock could be used to clock only the output pads, although this would adversely affect overall power consumption. Once the pad design is finalized, it should be characterized for delay and power consumption using Signalstorm. This should be done for all operating voltages in order to be integrated into the design tool flow.

Chapter 7

Conclusions

This chapter summarizes the main contributions, the project results, and future plans.

This thesis presented the design and implementation of a low-energy processor for sensor networks. The driving force of this project was the eventual maximization of energy efficiency, even at the cost of reduced performance, increased system complexity and larger chip area.

An architecture was developed to match a target set of wireless sensor network applications and minimize energy per computation. This architecture contains a programmable DSP, hardwired algorithms with local memory buffers, and interfaces to ADC and radio components. The digital system was fully described in Verilog and simulated to verify functionality.

A Xilinx FPGA platform was used to physically verify the full sensor-node system, including the digital logic and off-the-shelf ADC and radio. Since FPGAs are usually used as a logic prototyping platform, an ASIC was built to characterize the energy and performance of a final sensor node processor implementation. This thesis described the key steps in the design, verification, and testing of the ASIC. The first ASIC implementation produced a functional chip that was demonstrated as a stand-alone sensor node. Given the availability of some networking software, a wireless sensor network made up of several of these sensor nodes could easily be demonstrated as well.

The first ASIC was used mainly as a way to identify a comprehensive ASIC design

flow, at which it succeeded quite well. As a next step, this thesis proposes some aggressive energy reduction techniques that should make a future implementation very energy efficient.

Appendix A

Sample Source Code and Design Scripts

A.1 Sample DSP C source code

This section contains sample C source code which was used to program the μAMPS DSP. The program shown below performs an FFT using the hardwired FFT module and then displays the output for debugging. At the start of the main function, the 16-bit general-purpose I/O port is configured to be an output port. Next, the DSP registers the interrupt handler function fft_handler, which is called whenever the FFT interrupt is raised by the FFT module. The FFT interrupt line also needs to be enabled, since all interrupts are ignored by default. The program then enters an infinite while loop, in which it first pulls a block of test data from the off-chip ROM and places it in the DSP data memory. The DSP then displays the data through the debug port, from which a logic analyzer produces a plot as illustrated by Figure 3-3. Next, the DSP programs the FFT module to perform a 128-point, 16-bit FFT, and then feeds it the block of 128 data words, after which it waits for the FFT to be done. When the FFT completes execution, the fft_handler function is called, where the results of the FFT are read back into the DSP's memory and printed out through the debug port to produce a spectral plot similar to the solid line of Figure 3-3. Finally, the DSP performs some multiplications whose result is verified through the

debug port to ensure the multiplier unit works correctly.

```
// defines compiler libraries, common functions, constants
#include <uamps.h>
// holds FFT configuration parameters
#include "fft_defines.h"
// 128-word FFT
#define FFT_LENGTH_CODE FFT_LENGTH_128
#define FFT_LENGTH 2<<(10-FFT_LENGTH_CODE-1)
// 16-bit precision
#define FFT_PRECISION FFT_PRECISION_16

// memory block to hold FFT input/output data
unsigned int data[1024];
// memory block to hold configuration words
unsigned int cfg[100];
// set when a FFT interrupt is received by DSP
int fft_done=0;
// loop counter
int i=0;

// this is the FFT interrupt handler
void fft_handler(void)
{
  // at this point FFT should be complete
  // so DMA reads out words into DSP data memory
  u_dma_transfer(U_DMA_TYPE_FFT_TO_DMEM, &data, FFT_LENGTH);
  // wait for DMA engine to complete transfer
  while (!u_dma_is_idle());
```

```
  u_gpio_set_outputs(0xDEAD);
  // print read data to the gpio pins
  for (i=0; i<FFT_LENGTH; i++){
    u_gpio_set_outputs(data[i]);
  }
  // raise the done flag to indicate FFT is done
  fft_done =1;
}

// this is the main program thread
int main(void){
  int val =0;
  // configure gpio pins as outputs
  u_gpio_set_directions(0xFFFF);

  // set up fft interrupt handler
  u_register_interrupt_handler(fft_handler, U_FFT_IRQ_CHANNEL);
  u_enable_interrupt_source(U_FFT_IRQ_CHANNEL);
  u_enable_interrupts();

while (1) {  // run forever
  // DMA Read data from ROM and put it in data memory
  u_dma_transfer( U_DMA_TYPE_ROM_TO_DMEM, &data, FFT_LENGTH);

  // print read data to the gpio pins
  for (i=0; i<FFT_LENGTH; i++){
    u_gpio_set_outputs(data[i]);
  }

  // Now that we have adc data in memory, do FFT on it
```

```
  cfg[0] = FFT_CMD_LOAD_LENGTH + FFT_LENGTH_CODE;
  cfg[1] = FFT_CMD_LOAD_BITPRECISION + FFT_PRECISION;
  // configure FFT
  u_dma_transfer( U_XFER_DMEM_FFT_CFG, ((short)&cfg), 2);
  // send input data to FFT
  u_dma_transfer( U_XFER_DMEM_FFT, ((short)&data), FFT_LENGTH);
  while (!u_dma_is_idle());

  // Loop until fft is done
  while (fft_done == 0);

  // test the DSP multiplier
  for (i=0; i<FFT_LENGTH; i++){
    val = i*2;
    u_gpio_set_outputs(val);
  }
}
  while(0);
  return 0;
}
```

Bibliography

[1] S. Roundy, P. Wright, and J. Rabaey, "A Study of Low Level Vibrations as a Power Source for Wireless Sensor Nodes," *Computer Communications*, vol. 26, no. 11, pp. 1131-1144, July 2003.

[2] Deborah Estrin, "Embedding the Internet: This Century Challenges," *Presentation to the UCLA Computer Science Department*

[3] Ya-Lan Tsao, Ming Hsuan Tan, Jun-Xian Teng, Shyh-Jye Jou, "Parameterized and low power DSP core for embedded systems," *ISCAS '03. Proceedings of the 2003 International Symposium on*, Volume: 5 ,25-28 May 20

[4] B.A. Warneke, K.S.J. Pister, "An ultra-low energy microcontroller for Smart Dust wireless sensor networks," *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, 15-19, Feb. 2004, Pages:316-317 Vol.1

[5] I. Verbauwhede, C. Nicol, "Low power DSP's for wireless communications," *ISLPED '00, 2000. Proceedings of the 2000 International Symposium on*, 26-27 July 2000, Pages: 303-310

[6] R.A. Mucci, "A Comparison of Efficient Beamforming Algorithms," *IEEE Trans. on Accoustics, Speech and Signal Processing*, vol. 22, no.3, June 1984, pp. 548-558.

[7] M.L.L. Brackenbury, "An instruction buffer for a low-power DSP," *Sixth International Symposium on Advanced Research in Asynchronous circuits and Systems*, 2000.

[8] W.S. Lu, A. Antoniou, S. Saab, "Sequential design of FIR digital filters for low-power DSP applications," *Conference Record of the Thirty-First Asilomar Conference on signals, systems, and computers*, 1997.

[9] S. Mathew, R. Krishnamurthy, M. Anders, S. Hsu, S. Borkar, "Advanced circuit techniques for high-performance microprocessor and low-power DSPs," *DCAS-04 Proceedings of the 2004 IEEE Dallas/CAS workshop on the implementation of high performance circuits*, 2004.

[10] A.P. Chandrakasan, R.W. Brodersen, "Minimizing power consumption in digital CMOS circuits," *Proceedings of the IEEE*, Volume 83, Issue 4, April 1995, Pages 498-523.

[11] A. Wang, A.P. Chandrakasan, "A 180mV FFT processor using subthreshold circuit techniques," *Digest of technical papers, 2004 IEEE International Solid-State Circuits Conference*, Feb 9, 2004, Pages 310-319, Vol 1.

[12] Z. Chen *et al.*, "Estimation of standby leakage power in CMOS circuits considering accurate modeling of transistor stacks," *Proc. Int. Symp. Low Power Electronics and Design*, Monterey, CA, Aug. 1998, pp. 239-244.

[13] Srisathapornphat, C. Jaikaeo, C. Chien-Chung Shen, "Sensor Information Networking Architecture," *International workshops on parallel processing*, 2000, Pages 23-30.

[14] C. Kelly, V. Ekanayake, R. Manohar, "SNAP, a sensor-network asynchronous microprocessor," *Proceedings of the ninth international symposium on asynchronous circuits and systems*, 12-15 May 2003, Pages 24-33.

[15] P. Royannez et. al., "90nm Low Leakage SoC Design Techniques for Wireless Applications" *Proceedings of ISSCC'2005*, Feb 6-10, 2005.

[16] Huifang Qin et. al., "SRAM Leakage Suppression By Minimizing Standby Supply Voltage" Proceedings of the 5th International Symposium on Quality Electronic Design, Pages 55-60, 2004.

[17] T.D. Burd et. al., "A Dynamic Voltage Scaled Microprocessor System" *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, November 2000.

[18] J. Montanaro et al., "A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor," *IEEE Journal of Solid-State Circuits*, pp. 1703-1714, November 1996.

[19] ATmega168/V 8-bit Microcontroller Datasheet, http://www.atmel.com/dyn/resources/prod_documents/2545S.pdf, 2005.

[20] Intel PXA255 Processor Datasheet, http://www.intel.com/design/pca/prodbref/252780.htm, 2005.

[21] J. Rabaey, A. Chandrakasan, B. Nikolic, "Digital Integrated Circuits", 2003, Pearson Education, Inc.

[22] Chipcon 1010 User Manual, http://www.chipcon.com/files/CC1010DK_User_Manual_2_01.pdf, Chipcon AS, 2005.

[23] 4 channel, 16-bit sampling CMOS A/D Converter, http://focus.ti.com/docs/prod/folders/print/ads7825.html, Texas Instruments, 2005.

[24] UART RS-232 Protocol, http://www.beyondlogic.org/serial/serial1.htm, 2005.

[25] Synthesis and scripting techniques for designing multi-asynchronous clock designs, www.sunburst-design.com/papers/CummingsSNUG2001SJ_AsyncClk.pdf, 2001.

[26] Xilinx Virtex-2 FPGA,
http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Platform+FPGAs, 2005.

[27] ChipScope Documentation,
http://www.xilinx.com/literature/literature-chipscope.htm, 2005.

[28] Microphone Sensor, Knowles Acoustics EA-1842,
http://www.knowlesacoustics.com/images/data_sheets/, 2005.

[29] Synopsys Formality Documentation,
http://www.synopsys.com/products/verification/verification.html, 2005.

[30] Synopsys PrimeTime Documentation,
http://www.synopsys.com/products/analysis/sta_brochure.pdf, 2005.

[31] Synopsys Star-RCXT Documentation,
http://www.synopsys.com/products/avmrg/star_rcxt_ds.html, 2005.

[32] Synopsys Nanosim Documentation,
http://www.synopsys.com/products/mixedsignal/nanosim/nanosim.html, 2005.

[33] Synopsys Design Compiler Documentation,
http://www.synopsys.com/products/logic/design_compiler.html, 2005.

[34] Synopsys Astro Documentation,
http://www.synopsys.com/products/avmrg/astro_ds.html, 2005.

[35] Synopsys Hercules Documentation,
http://www.synopsys.com/products/hercules/hercules.html, 2005.

[36] Cadence Signalstorm Library Characterization Documentation,
http://www.cadence.com/products/digital_ic/signalstorm/index.aspx, 2005.

[37] Synopsys TetraMax Documentation,
http://www.synopsys.com/products/solutions/galaxy/test/test.html, 2005.

3506. 34