A Micropower DSP for Sensor Applications

by

Nathan J. Ickes

B.S., Massachusetts Institute of Technology (2001)
M.Eng., Massachusetts Institute of Technology (2002)

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

Doctor of Philosophy in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

Author ....
Department of Electrical Engineering and Computer Science
May 23, 2008

Certified by ....
Anantha P. Chandrakasan
Professor
Thesis Supervisor

Accepted by ....
Terry P. Orlando
Chair, Department Committee on Graduate Students

A Micropower DSP for Sensor Applications

by

Nathan J. Ickes

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering

Abstract

Ultra-low power systems, such as wireless microsensor networks or implanted medical devices, are driving the development of processors capable of performing increasingly complicated computations using mere microwatts of power. This thesis describes the design of a micropower DSP intended for medium bandwidth microsensor applications (such as acoustic sensing and tracking) which achieves 4 MIPS performance at 40 µW (10 pJ per instruction) operating at 450 mV and fabricated in 90 nm CMOS. Energy efficiency optimizations include a custom CPU instruction set, a miniature instruction cache with a novel replacement strategy, hardware accelerator cores for FIR filter and FFT operations, and extensive power gating of both logic and memory.

The tradeoffs of cache size, line length, and replacement policy for very small (a few hundred words or less) caches are explored, as are the design implications of optimizing the cache for minimum energy without regard to performance (since on-chip memory access is already single-cycle). A replacement policy designed to reduce thrashing in miniature instruction caches is presented.

Efficient control of power-gated circuits requires consideration of the minimum off time, or break-even time. An energy model for determining the break-even time is developed, which correlates with measurements of the power-gated domains on the DSP.

The energy savings obtained from hardware accelerators for FIR filtering and FFT operations are measured, and a model is developed to predict the actual net power reduction in a real system, including factors such as sampling rate, leakage power, latency requirements, and power gating overhead.

Thesis Supervisor: Anantha P. Chandrakasan
Title: Professor

Acknowledgments

I would like to begin by thanking Professor Anantha Chandrakasan for his guidance and support as my advisor. Anantha has been exceptional at giving me the freedom to identify research problems that particularly interest me, and finding me the resources to pursue those problems, as well as providing insight on how to identify particularly interesting and worthwhile problems.

I would also like to thank Professors Arvind and Sam Madden for serving on my thesis committee, and for their advice, insights, and enthusiasm that have help shape this thesis. I am particularly grateful for their input on ways to make this work more interesting to a broader audience including the architecture and sensors communities, in addition to the circuits community.

Daniel Finchelstein and I worked together on two previous generations of the DSP chip described in this thesis, and in the process the two of us identified many of the design issues that shaped the architecture of this third generation DSP. During the design of these earlier chips, Daniel worked out solutions to many, many tool and design flow issues, which I am extremely grateful not to have had to solve myself. I would also like to thank Vivienne Sze, Denis Daly, Fred Chen, Naveen Verma, and David Wentzloff for sharing their expertise.

Joyce Kwong and Yogesh Ramadass read initial drafts of portions of this thesis and provided insightful feedback.

This work was sponsored by the DARPA PAC/C program, the Focus Center Research Program, and Texas Instruments. Fabrication of the µAMPS DSP was donated by ST Microelectronics.

Nathan Ickes
May, 2008

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Power efficiency is today a universal constraint in computer system design, either due to heat dissipation limits or battery lifetime concerns. Ultra-low power applications, such as wireless microsensor networks, or implanted medical devices, are driving the development of processors capable of performing increasingly complicated computations using mere microwatts of power.

1.1 Microsensor Applications

Microsensor networks may consist of many—perhaps hundreds or thousands—of miniature sensor nodes scattered throughout an area of interest and linked by a wireless network. The network of sensors collaborates as a whole, combining measurements made by each individual node and delivering high-quality observations to a central base station. The large number of nodes in a microsensor network results in high-resolution, multi-dimensional observations and fault-tolerance superior to more traditional sensing systems. This makes them attractive for a wide range of applications, such as inventory tracking, environmental monitoring, machine-mounted sensing, medical monitoring, and building climate control. For many applications, a primary advantage of microsensor networks is the spatial diversity of the data collected by the network as a whole. This diversity can be exploited to reveal details about the network's environment that would not necessarily be visible using a single

Table 1.1: Energy harvesting techniques[a]

Energy Source	*Technology*	*Power Output*	
Machine vibration	Electromagnetic	2.3 µW/cm^3	[5]
Falling rain	Piezoelectric	1 nJ/drop	[6]
Human body heat	Thermoelectric	250 µW	[7]
Indoor lighting	Photovoltaic	10 µW/cm^2	[b]

[a]Data courtesy of Yogesh Ramadass.

[b]Measured from an amorphous silicon solar cell (Trony SC1025IDS) taken from an inexpensive calculator.

large sensor. Alternatively, the sensor network may be used to imitate a single very large sensor, one that might be impractically large to build or deploy [1,2].

The desirability of extremely small, yet long-lived sensor nodes makes extreme power efficiency the central issue in microsensor design. Many applications call for battery lifetimes measured in years. Achieving a one-year battery life with a 1 Ah lithium coin cell (3 V, 4 cm^3, 11 kJ [3]) requires limiting average power consumption to approximately 350 µW, and many applications would benefit from nodes with significantly smaller volumes and longer lifetimes. The Holy Grail of microsensor design is a self-powered node, scavenging energy from ambient solar, thermal, or mechanical sources. (Table 1.1 compares the obtainable power for several energy scavenging techniques.) While at least one viable solar powered microsensor node has been demonstrated [4], it is physically large and limited to outdoor applications where it receives direct sunlight. With commercially available microsensor nodes currently consuming watts or milliwatts or power, many potential microsensor applications are not yet viable and await the development of new node architectures with orders of magnitude lower power consumption.

While the space of microsensor applications is extremely wide and varied, the following characteristics are fairly common: [8,9]

Low duty cycle. In most applications, nodes operate with an extremely low duty cycle. Events of interest to the network may be spaced minutes, hours, or days apart, meaning that nodes can be idle over 99% of the time. Minimizing standby power consumption is critical, as standby energy may greatly exceed active energy.

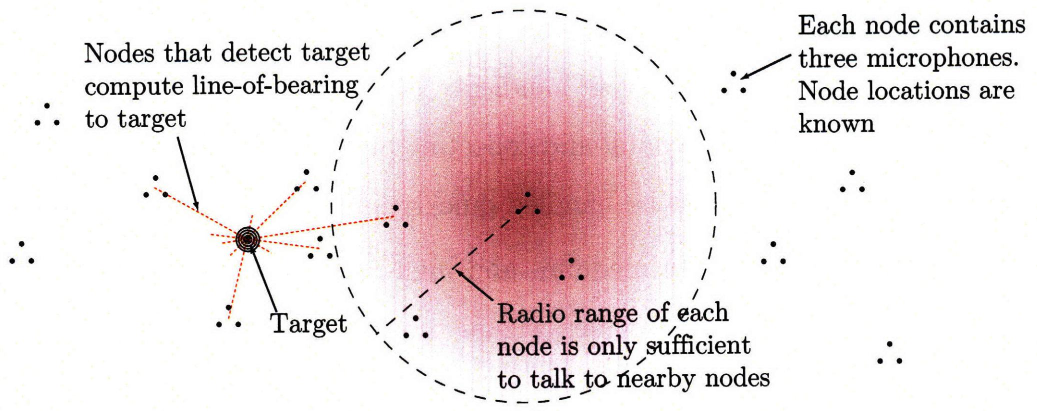
Event driven. Microsensor applications are largely event driven: the work profile consists mainly of short interrupt event handlers. Typical events handled by nodes include sending or receiving radio data, and collecting measurement data. To maximize node lifetime, these events must be handled quickly and efficiently.

Localized data processing. In medium- and high-bandwidth applications, significant energy can be saved if preliminary signal processing and data analysis occurs within the network. For instance, nearby nodes might aggregate their data streams using beamforming algorithms, thereby reducing the amount of data that must be sent to the network basestation. Beamforming can be computationally intensive, and implementing such algorithms may significantly increase the peak processing capability required on each node.

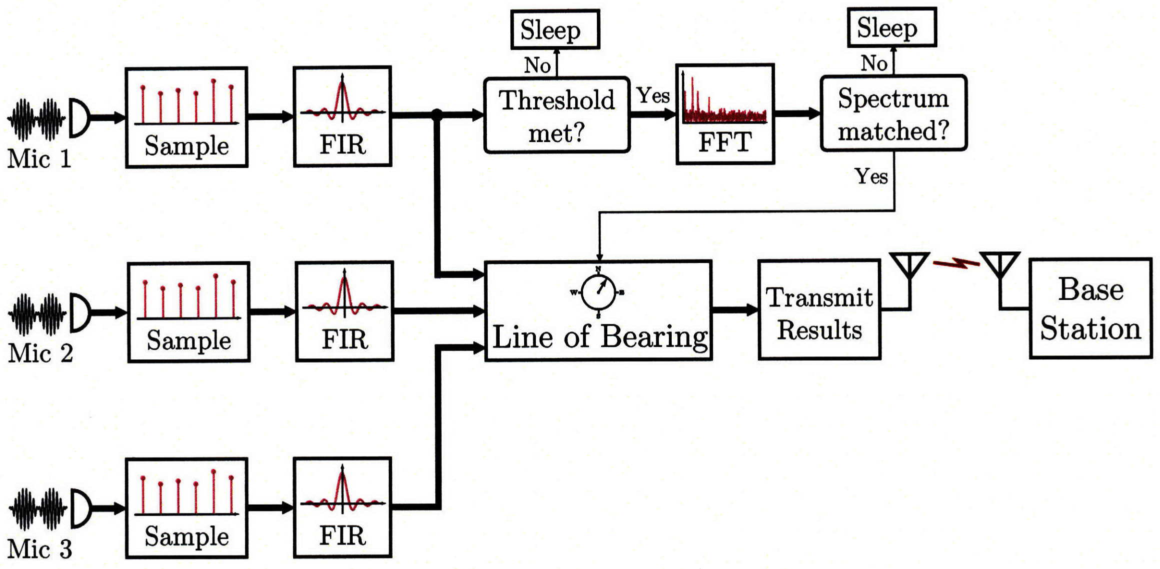
Unpredictable performance requirements. Performance demands on any given node are variable and unpredictable before deployment. Local variations in node density can modulate the distance between adjacent nodes, for instance, creating variations in the nodes' required radio transmission power. The ambient noise level on sensor readings can change, creating variations in the amount and type of signal processing required.

Microsensors have been an active area of research for several years. Early architectures based on off-the-shelf components, such as the Berkeley Mote [10] platform, are commercially available, and descriptions of a number of custom microsensor chips have been published [11–14]. Much of the early work on microsensors focused on very low-bandwidth applications, such as environmental sensors that monitor slow-varying quantities such as temperature or light level. In such applications, individual sensor nodes perform almost no actual computation: the raw measurements recorded by each node's sensors are transmitted to the network basestation, or stored in the node for later retrieval. Most or all data analysis takes place offline, outside the microsensor network. The principle tasks of the software running on each node include handling low-level hardware events (such as timer interrupts) and implementing the network protocol stack and power management strategy.

The MIT µAMPS (micro, adaptive, multi-domain, power aware sensors) project


Nodes that detect target compute line-of-bearing to target
Each node contains three microphones. Node locations are known
Target
Radio range of each node is only sufficient to talk to nearby nodes

(a)



(b)

Figure 1-1: Acoustic tracking application

(for which the DSP described in this thesis was designed) targets a more computationally intensive application space. µAMPS microsensors are designed for acoustic tracking and other applications requiring sensor sampling rates of $1 - 100\,\mathrm{kS/s}$ and significant post-acquisition signal processing, such as filtering, compression, or spectral analysis. Nodes built from off-the-shelf components for these applications consume from tens of milliwatts to more than a watt of power when active [15–18]. The goal of the µAMPS project is to demonstrate a node architecture optimized for these applications with sufficiently low power consumption to enable self-powered operation.

Figure 1-1 illustrates an example acoustic tracking microsensor application. Nodes are scattered at known locations throughout a region of interest. Each node is equipped with an array of three microphones, spaced 50 cm or so apart, at the corners of an equilateral triangle. Most of the time, the nodes are in a deep sleep state to save energy, but when a node detects a loud enough sound, it wakes up and begins recording from its microphones. The first task of an awakened node is to determine (from the frequency spectrum of the recorded signal) whether the sound source is interesting, as defined by the task for which the network was deployed. Perhaps the network is being used to track certain types of vehicles, or to track the movement and behavior or certain species of wildlife, based on their calls. If a noise is deemed interesting, each node that detects the noise computes the line of bearing from its location to the source of the noise, based on the phase differences of the signals recorded from the node's three microphones. The line of bearing results are then radioed to the network basestation, where the location of the sound source can be determined by triangulation.

The key signal processing algorithms for this application include a Fourier transform, and a line-of-bearing estimation (such as the algorithms described in [19, 20]). Additionally, some filtering of the raw microphone data is likely important, in order to reduce the effects of noise at frequencies that are not relevant to the application.

1.2 The µAMPS DSP

The basis of this thesis is a 4 MIPS, 10 pJ per instruction DSP designed to form the core of a µAMPS sensor node. We now introduce the DSP, describe its features, and summarize its measured performance. In each of the remaining chapters of this thesis, we will examine a particular subsystem of the DSP in detail, discussing the design decisions that were made, and the theory behind those decisions.

The block diagram for a complete µAMPS node, built around our custom DSP chip, is shown in Figure 1-2. The node consists of three primary components: the DSP, a custom 12-bit 100 kSPS ADC [21], and a commercial ZigBee radio (the ChipCon

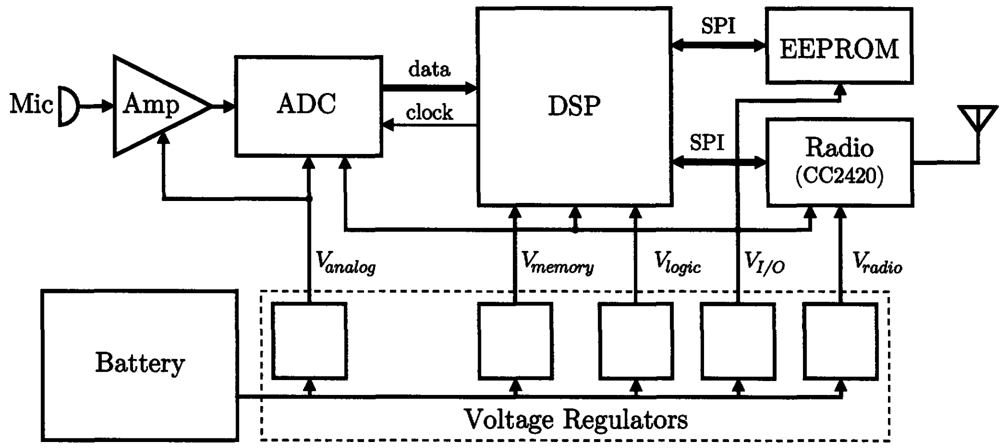


Figure 1-2: µAMPS sensor node architecture

CC2420). A serial EEPROM is provided, for nonvolatile code and data storage. The node's power source is a small battery, and multiple voltage regulators are used to generate the different voltages required by each component.

1.2.1 DSP Chip Architecture

Figure 1-3 illustrates the architecture of the DSP chip itself. The key features of the DSP include:

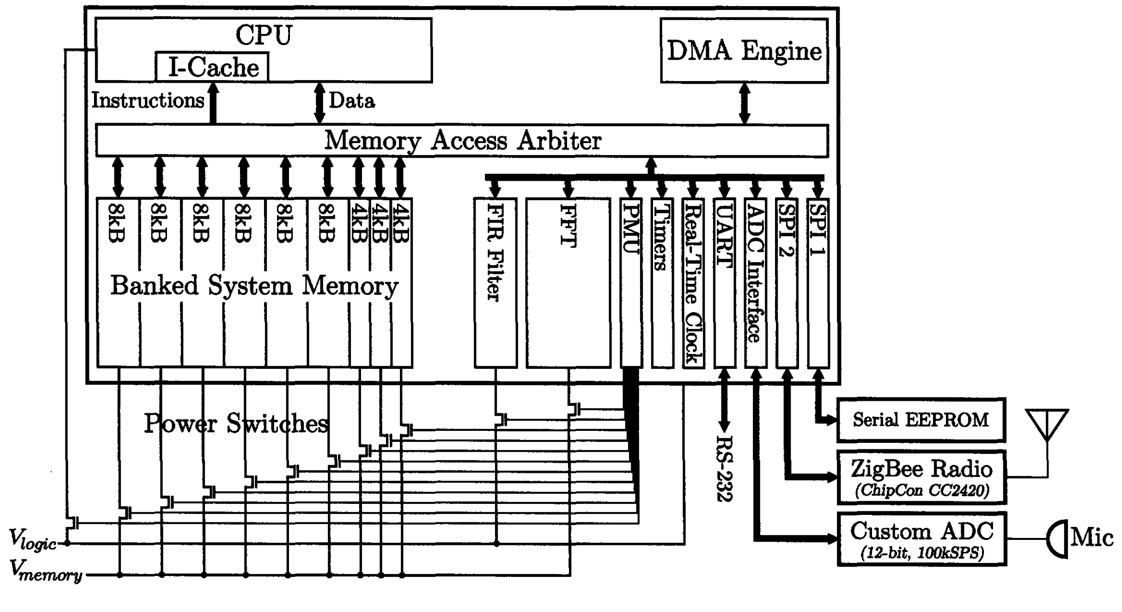


Figure 1-3: DSP block diagram

- A 16-bit RISC CPU with a custom instruction set designed specifically to ease instruction decoding and simplify the pipeline control logic, in order to minimize the execution energy per instruction.

- 60 kB of on-chip memory (SRAM), utilized for both program and data storage. To reduce access energy, the memory is divided into nine smaller blocks. There are no system-wide busses. Instead a crossbar-like arbiter circuit routes memory transactions generated by the CPU and DMA engine to the appropriate peripheral device or memory block.

- A miniature 64 word instruction cache reduces the memory access rate, and significantly decreases the average instruction fetch energy.

- Accelerator cores for FIR filter and FFT operations are implemented as memory-mapped peripherals. (Additional accelerators could be easily added in future versions of the chip.) The FIR filter accelerator implements up to a 16-tap symmetric filter, and can perform automatic downsampling (generating one output sample for every n input samples), making it suited for use as a digital anti-aliasing filter. The FFT core computes Fourier transforms on 64-, 128-, 256-, or 512-point complex inputs or 128-, 256-, 512- or 1024-point real-valued inputs, with 16-bit precision.

- A DMA engine is included, for efficiently copying data between various components of the DSP, particularly between the accelerator cores and main memory.

- On-chip peripherals include an RS-232 UART (for debugging purposes), two SPI interfaces (for communicating with the radio and an external EEPROM), a timer system with input- and output-compare functionality, a real-time-clock for timing long-duration events, and a simple power-management unit.

- The CPU, accelerator cores, and each of the nine memory banks are all power gated, using external power switches.

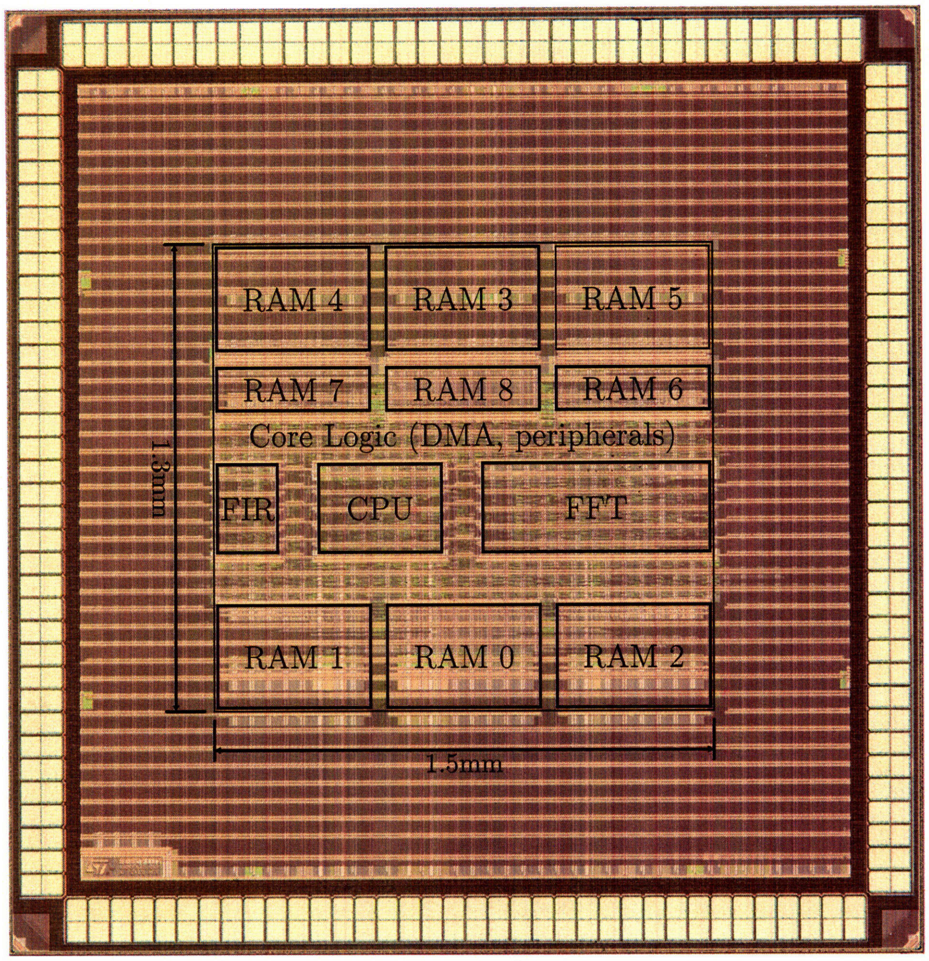


Figure 1-4: DSP die photo

The DSP is implemented in ST Microelectronics' 90 nm low-power CMOS process technology, and contains approximately 6.3 million transistors (6 million of which are contained in the on-chip memory). Figure 1-4 shows a photo of the DSP die, with the locations of the processor and memory components identified. The chip dimensions are limited by the number of I/O pads; all of the active circuitry located at the center of the die area. (Since was a test chip and ease of debugging was more important than minimizing die area, no attempt was made to minimize the number of I/O signals.)

1.2.2 Performance

Testing and characterization of the DSP was conducted using the test board shown in Figure 1-5, which incorporates all of external components from Figure 1-2 necessary

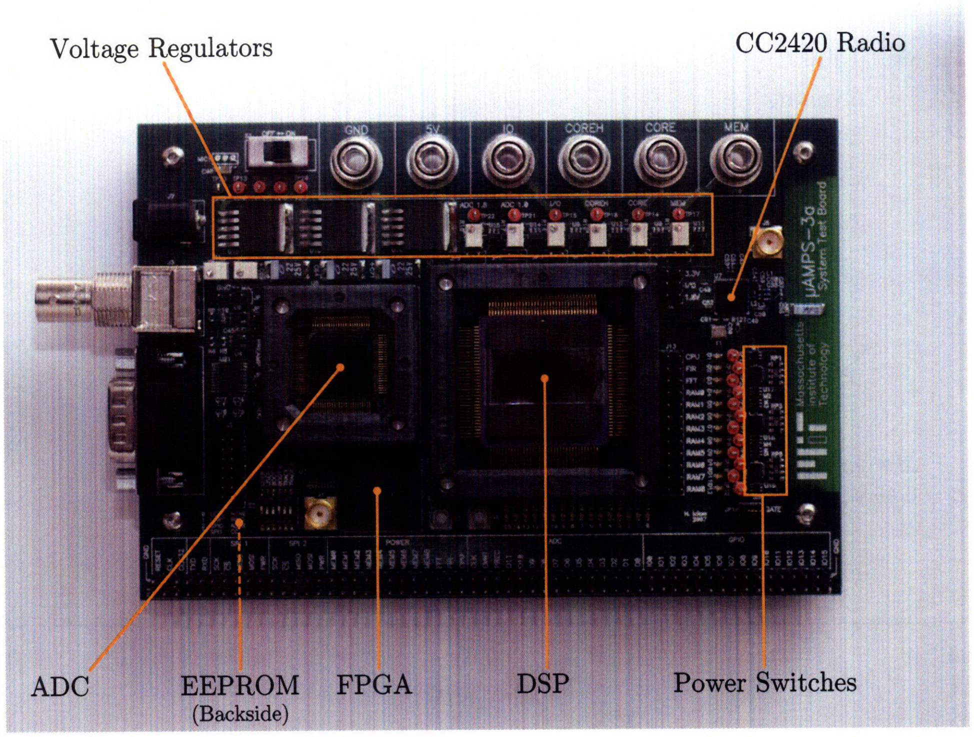


Figure 1-5: DSP test board

to create a complete sensor node. An actual node would be considerably smaller than the test PCB, which includes extra components (such as an FPGA) and connectors to facilitate testing.

Figure 1-6 shows the measured energy per clock cycle and maximum clock speed for the DSP, as the power supply for the logic portions of the chip is scaled from 800 mV down to 440 mV (the lowest voltage the DSP will operate at, at any clock speed). The power supply for the on-chip memory banks was fixed at 800 mV. During these measurements, the CPU was executing a 16-tap FIR filter (implemented in software, not using the FIR accelerator). The FIR and FFT accelerator cores, along with all but two of the memory banks, were powered off.

At 0.8 V, the DSP operates at 50 MHz and 22 pJ per instruction. The optimal, minimum-energy-per-cycle operating point occurs at 450 mV and 3.95 MHz, where the DSP consumes 10.3 pJ per cycle (41 µW).

As the supply voltage is decreased, dynamic ($\propto CV^2$) energy consumption decreases correspondingly. Because the memory voltage is not scaled, the portion of the

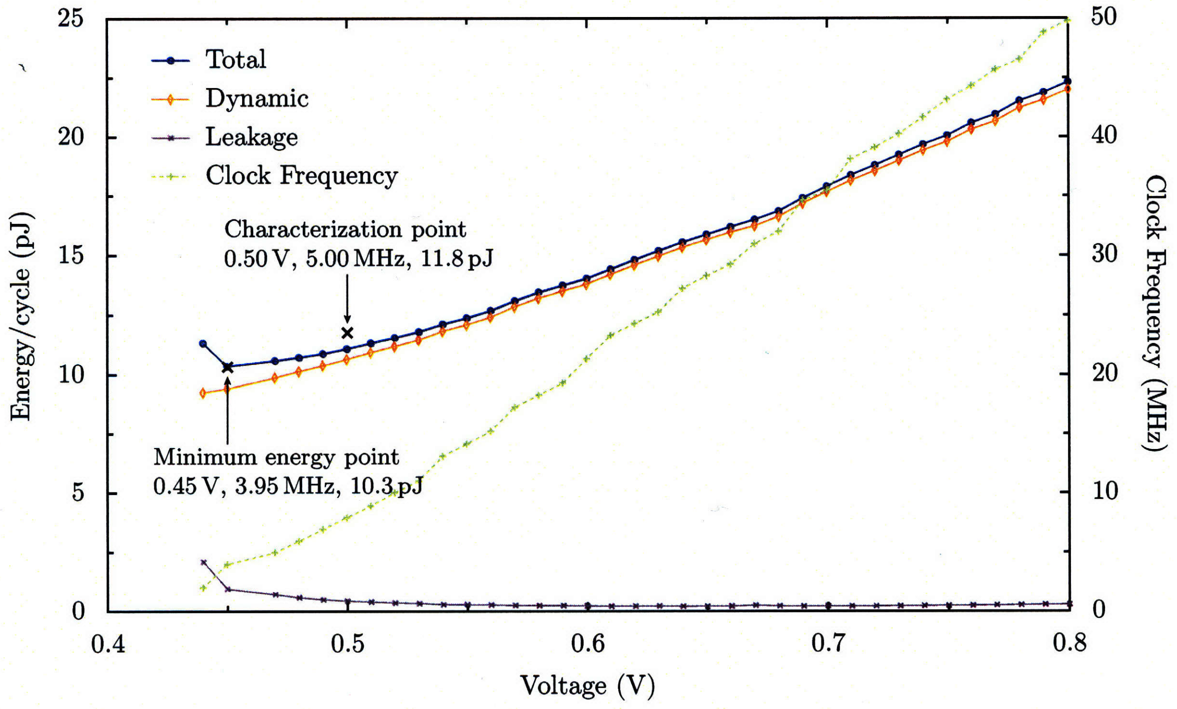


Figure 1-6: Energy per cycle, as a function of logic voltage. (The memory voltage is fixed at 0.8 V.) Maximum operating frequency is also plotted (dashed line), against the right-side axis.

total dynamic energy per cycle due to the memory does not change, and therefore the total dynamic energy per cycle flattens out at low voltages, where the constant memory energy dominates over the shrinking logic energy. Although static leakage power also decreases with voltage, the increasing clock period causes a net increase in leakage energy per cycle as the voltage decreases. This results in the observed minimum energy point at 450 mV.

The efficiency (energy per instruction) and performance (maximum clock fre-

Table 1.2: Comparison of the µAMPS DSP with other micropower processors

Optimal operating point:	Voltage (V)	Clock Frequency (MHz)	Energy/instruction (pJ)
Smart Dust [11]	1.0	0.50	12
SNAP/LE [12]	0.6	28	24
Subliminal [22]	0.4	1.6	2.7
Sub-V_{th} MSP430 [23]	0.5	0.43	27.3
µAMPS (this work)	0.45	3.95	10.3

quency) of the µAMPS DSP are compared against other recently published micropower processors in Table 1.2. Although other processors have achieved lower energy per instruction, the µAMPS DSP nonetheless significantly advances the pareto-optimal frontier, achieving either lower energy or higher speed than other published architectures. It is important to note, however, that energy per instruction and maximum clock speed are only two of many measures of processor performance. Table 1.2 does not, for example, illustrate the benefit of accelerator cores, or power gating, or consider the amount of program and data memory included in each architecture. (Memory access energy accounts for a significant portion of the average instruction energy, and larger memories have higher access energies.) Furthermore, the average energy per instruction depends significantly on the workload being measured, and there are, as yet, no standard benchmark suites for micropower processors. We opted to use an FIR filter as our energy characterization benchmark, because it represents a significant processing task which thoroughly exercises our CPU, including the multiplier and barrel shifter units. For simpler workloads, the average energy per instruction may be lower (we have measured as little as 6 pJ), particularly if the instruction cache hit rate is very high.

Many of the measurements we will present in the following chapters were performed at the characterization point identified in Figure 1-6, corresponding to 5 MHz and 0.5 V, which is only slightly less efficient (11.8 pJ/instruction) than the lowest energy operating point. Operating at 5 MHz facilitates communicating with the DSP's bootloader program via RS-232 at a standard baud rate. Increasing the voltage to 0.5 V (slightly above the minimum voltage required for 5 MHz operation) increases reliability by reducing susceptibility to power supply noise.

1.3 Thesis Contributions

The main contributions of this thesis are in the following four areas.

1. *Memory power optimization.* This work represents the first micropower processor design to explicitly focus on minimizing memory power throughout the

architecture. Memory power has been addressed not only in the memory implementation itself, but in almost all components of the design, including the CPU instruction set and the accelerator cores.

2. *Instruction cache design.* This work also represents the first use of a non-trivial instruction cache in a micropower processor. We explore the tradeoffs of cache size, line length, and replacement policy for very small (a few hundred words or less) caches, and the design implications of optimizing for minimum energy without regard to performance (since on-chip memory access is already single-cycle). A replacement policy designed to reduce thrashing in miniature instruction caches is presented.

3. *Modeling of power-gating.* Efficient control of power-gated circuits requires consideration of the minimum off time, or break-even time. We measure the break-even time for each of the power-gated domains of the DSP, and develop an energy model for a power-off/power-on cycle, which matches well with measured data.

4. *Hardware accelerators.* We present a framework for assessing the energy savings delivered by an accelerator core and then use that framework to evaluate the accelerator cores implemented on the µAMPS DSP. We also develop a model for determining the effective power savings obtained from an accelerator as part of a complete application, and test that model using a simple application running on our DSP.

1.4 Thesis Organization

The chapters that follow each concern one particular portion of the µAMPS DSP design. Chapter 2 discusses the use of power gating, including the importance of the minimum off time, or break-even time. In Chapter 3 the CPU and its instruction set is described and compared against other architectures, both commercial and custom. Chapter 4 considers the memory system, particularly focusing on the design of the tiny instruction cache. Chapter 5 deals with the accelerator cores. Finally, the contributions and conclusions of this work are summarized in Chapter 6.

Chapter 2

Power Gating

The low duty cycle nature of microsensor applications makes minimizing idle mode power a high priority. Clock gating, which is inserted automatically and transparently by modern synthesis tools, can greatly reduce dynamic power consumption in idle logic. However, this does not reduce leakage power, which then becomes the dominant source of idle-mode power consumption, particularly for deep-sleep states and modern sub-100 nm process technologies.

Power gating is a well-known mechanism for reducing idle-mode leakage power, but in practice is complicated to implement and, despite over a decade of publications on the subject [24–28], has not yet become a common ASIC design technique. Mainstream synthesis and layout tools are only beginning to support multiple voltage domains and do not yet support automatic insertion of power-gating devices into a design. The control of power gating is complicated because, except for small circuit blocks, power cannot be turned on and off on a cycle-by-cycle basis as is the case in clock gating. Some amount of planning ahead is required before powering off a logic block, to ensure that power can be restored in time before the logic is needed again.

An obviously important characterization of the performance of a power-gating design is the leakage reduction ratio: by what factor is leakage reduced when the power is turned off? To achieve a high reduction ratio, the power gating switch must be a significantly better (i.e., lower leakage) device than the transistors in the circuitry being power gated. One way to accomplish this is to use a higher threshold voltage device for the power switch: this is known as MTCMOS [24]. Alternatively, boosting the gate voltage to the power switch can permit shrinking the switch to achieve lower off-mode leakage, without increased voltage drop during power-on mode. [29]. In the µAMPS DSP, both of these techniques are used, though unlike in true MTCMOS, the switches are off-chip.

A second—and much less studied or reported—metric of power gating performance is break-even time, or minimum off time: the minimum time a circuit must be powered off in order to achieve any net energy savings, after accounting for the energy expended turning the circuit off and on.

In this chapter we first discuss how power gating was implemented on the µAMPS DSP, and then describe a method for measuring break-even time experimentally and a simple circuit model based on the experimental results.

2.1 Implementation

The modular nature of our DSP's architecture is well suited to coarse-grained (module level) power gating. We implemented twelve independent power domains, consisting of each of the nine memory banks, the FFT and FIR accelerator cores, and the CPU. These are the largest modules in the µAMPS architecture, and together account for more than 90 % of the total leakage power, as shown in Figure 2-1. The remaining non-power-gated logic on the DSP constitutes a thirteenth, always-powered domain (the "core" category in Figure 2-1). This domain includes the memory arbiter, as well as the DMA engine, boot ROM, and other small modules (timers, serial ports, ADC interface, etc.) which were too small to warrant power gating individually.

2.1.1 On-Chip Circuitry

As shown in Figure 2-2, the twelve power-gated domains of the DSP are implemented as islands inside the always-on "core" domain. Each domain can be powered on or off independently of all the other domains. No signals pass directly from one power-gated domain to another. The interfaces between power domains are always between one

Table 2.1: Variables for power gating analysis

Name	*Description*
C_{GD}	Gate-drain parasitic capacitance of a power switch transistor
C_{GS}	Gate-source parasitic capacitance of a power switch transistor
C_L	Effective virtual-supply node capacitance of the module being power gated
E_{switch}	Energy required to turn a power switch on (or off, in the case of a PFET)
$E_{recharge}$	Energy required to charge the virtual supply node back to V_{DD} when the power-gated module is turned back on.
f	Frequency at which power switch is turned on and off (in the break-even time experiment)
P_{SW}	Power drawn from the switch drive power supply (averaged across an entire power-gating cycle)
P_{DD}	Power drawn from the V_{DD} supply (averaged across an entire power-gating cycle)
P_{leak}	Leakage power of the power-gated module when powered switch is on
Q_1	Charge drawn from C_L when the power switch is turned off
Q_2	Charge lost from C_L due to leakage, while the power switch is off
Q_3	Charge returned to C_L when the power switch is turned on
Q_{DD}	Total charge drawn from the V_{DD} power supply over the course of a complete power-gating (on/off) cycle
Q_{switch}	Total charge drawn from the V_{SW} power supply over the course of a complete power-gating (on/off) cycle
t_{be}	Break-even time: minimum time that a power-gated module must remain powered off in order to save any net energy
V_V	Voltage of the virtual supply node

domain that is always powered, and one that is power gated.

In order to allow the memory power domains to operate at a higher voltage than the logic portions of the chip, voltage level converters are used on every memory input signal. The level converter, as shown in the inset portion of Figure 2-2, is of a differential-cascode voltage switch (DCVS) design. This traditional design is simple, but its drawbacks are that it is relatively slow and has high dynamic power consumption (compared to more exotic level converter designs), due to the contention caused by the cross-coupled PFET devices. However, the total number of level converters used (294 total among all the memory domains) is small, and only a very small frac-

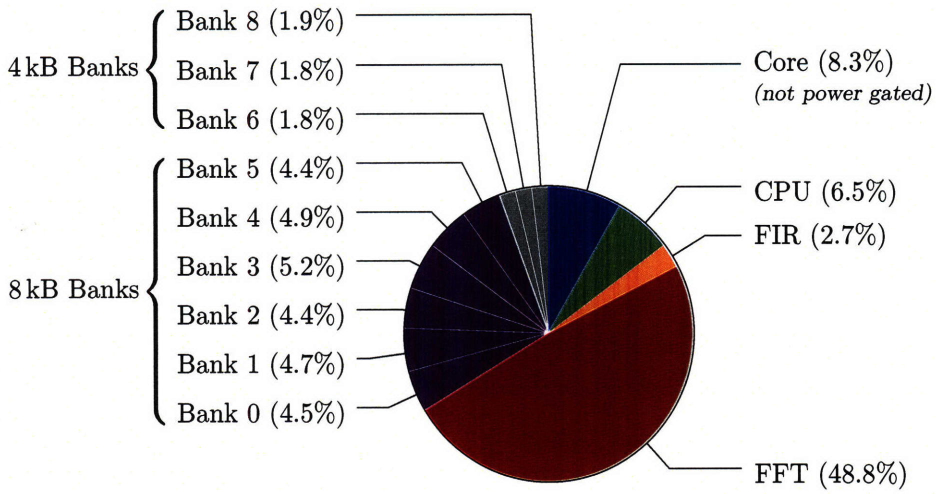


Figure 2-1: Breakdown of leakage power by power domain, for the entire DSP chip.

tion of these will toggle on any given cycle. The level converter was designed to convert from an input voltage as low as 0.5 V to an output voltage as high as 1.0 V.

When a power domain is turned off, the inputs and outputs of that domain are isolated, using the AND gates shown in Figure 2-2. The input isolation gates prevent toggling input signals from propagating into the power gated module and causing

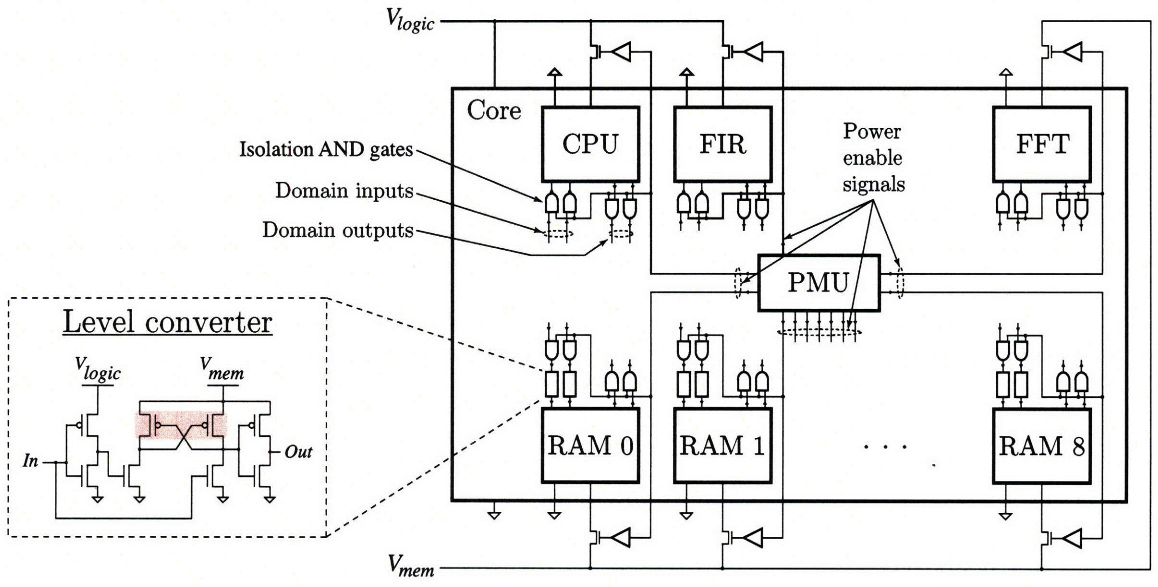


Figure 2-2: The twelve gated power domains of the DSP form islands within the thirteenth, always-on "core" domain. Inputs and outputs of each gated domain are isolated using AND gates whenever that domain is powered off. DCVS-style level converters (inset) enable the memory domains to operate at a higher voltage than the logic portions of the chip. (Shaded transistors in the level converter represent high V_{th} devices.)

unnecessary dynamic power consumption. The output isolation gates serve to ensure that midrange voltages which may be present on the outputs of the domain when it is powered down are resolved to a legal voltage before being propagated.

When a domain is powered off, the internal state of its circuitry is lost. This means that memory banks can only be powered off when their entire contents are no longer needed, and logic domains must be reinitialized when powered back on. Mechanisms have been proposed in the literature for retaining the state of flip-flops when logic is powered off [26]. However, a disadvantage of any form of state-retention is a necessary increase in leakage power during the off state. We opted to forgo any automatic state retention during power-down, because the CPU as well as the FIR and FFT accelerators each contain relatively little state that must be retained (at most about 400 bits), and this can be saved and restored manually, through software.

2.1.2 Power Management Unit

A very simple power management unit (PMU) provides an interface for software running on the CPU to manually control which domains are powered on. The PMU is implemented as a set of memory-mapped registers that may be read or written by the CPU. Power to each module is controlled by a power-enable register in the PMU: each bit in the register corresponds to one of the power domains.

Power gating of the CPU requires some additional assistance from the PMU, as the CPU power-up process cannot be entirely controlled in software like it is for the FIR and FFT accelerators. An extra interrupt mask register in the PMU selects which interrupt sources may trigger a wakeup of the CPU from the powered-off state. Wakeup of the CPU is triggered when an interrupt request (such as a timer event) matching the wakeup interrupt mask occurs. Power and clock to the CPU are enabled immediately and the CPU reset signal is asserted. After a programmable number of clock cycles (to allow the CPU voltage to stabilize), the CPU reset is released and code execution begins at a special "warmstart" vector. Code located there restores the state of the CPU (i.e., the register file contents) from memory and then resumes execution at the point where it left off when the CPU was powered down.

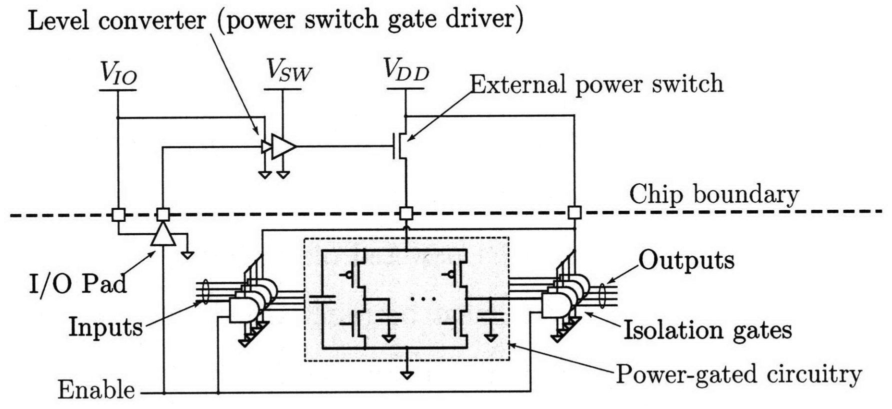


Figure 2-3: Power gating implementation with off-chip power switches

2.1.3 Off-Chip Circuitry

The power switches for each of the gated domains are implemented off-chip, using discrete NFETs. While external switches are in some ways an unattractive implementation—they increase the PCB component count, require additional I/O pins, and cannot be custom sized to the circuitry they are controlling—their advantages are they provide extremely high off-resistance, and they allow easy access for making measurements.

Figure 2-3 illustrates the power switch implementation. NFET power switches were chosen due to availability (at the time of the design) of parts with lower threshold voltage and lower gate capacitance. Given the 0.5 V target operating voltage for the DSP, utilizing a PFET device with a threshold voltage of less than −0.5 V would have required driving the gate below the ground rail of the chip, in order to turn the switch on. With an NFET switch, the gate drive voltage must be at least $V_{DD} + V_{th}$ relative to the system ground in order for the switch to turn the switch on. A Vishay SI1912EDH device was chosen, which has a specified threshold voltage of $V_{th} = 0.45$ V.

Discrete level converters (Fairchild FXLP34P5X) are used to drive the gates of the power switches, converting between the DSP I/O voltage (2.5 V) and the desired gate drive voltage. Using the level converters allowed experimentation with the gate drive voltage in order to achieve optimal performance, and also created a convenient means for measuring the energy consumed driving the switches (see Figure 2-4).

The gate drive voltage was determined experimentally by incrementally raising the gate voltage until no further increase in the output voltage to the DSP was measured. The gate voltages determined this way were 1.1 V for the logic-only domains (CPU and FIR, at $V_{DD} = 0.5\,\text{V}$) and 1.4 V for the SRAM domains (FFT and RAM0–RAM8, at $V_{DD} = 0.8\,\text{V}$).

We were unable to measure the leakage reduction ratio when the power switches are turned off because the off-mode leakage currents were too small to measure reliably. We estimate the off-mode leakage for each domain to be less than 10 pA, which would roughly correspond to a leakage reduction of five orders of magnitude.

2.2 Break-Even Time

When a power domain is turned off and then back on, significant energy is consumed driving the gate of the power switch, recharging the domain to its operating voltage, and restoring the internal state of the power-gated circuitry. If the power-off interval is overly short, this overhead energy can exceed the energy saved from power gating the load. A key parameter of any power-gating implementation is therefore the break-even time: the minimum time that a module must remain powered off, in order for the energy savings during the off period to exceed the overhead energy associated with powering it back on.

As our power gating implementation relies on software for manually restoring state after power-on, for simplicity we neglect the state restoration energy in this chapter and deal only with the circuit-level switch and recharge energies. We will come back to the state-restoration issue in Chapter 5.

Determining the break-even time experimentally is relatively easy, and does not require any understanding or modelling of the underlying behavior of the power-gating circuit. Figure 2-4 shows the experimental setup. A function generator—set to output a 50 % duty-cycle square wave—is connected to the power enable signal (the input of the external level converter) for one domain of our DSP, and we measure the average switch (P_{SW}) and load (P_{DD}) power as functions of the function generator frequency f. The raw power measurements (for the FIR accelerator domain of the

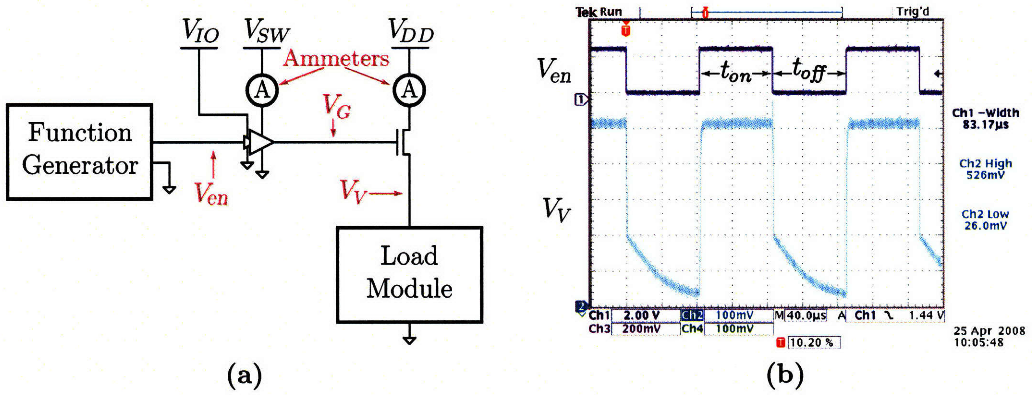


Figure 2-4: Experimental setup for measuring break-even time. The power enable signal for the power domain under test is driven with a variable-frequency square wave, while the average power consumption from the V_{SW} and V_{DD} supplies is measured. The oscilloscope screen capture in (b) shows voltage waveforms recorded for the enable signal and the virtual supply node, V_V.

DSP) are shown in Figure 2-5(a). The break-even time t_{be} is $1/(2f_{be})$, where f_{be} is the function generator frequency at which $P_{SW} + P_{DD}$ equals the leakage power P_{leak} of the load when powered on continuously. The measured break-even time for each of the DSP's power domains is reported in Table 2.2. In general, the lower the leakage power, the longer the break-even time.

In order to interpret and eventually model the operation of our power-gating circuit, it is more convenient to work in terms of energy and time, instead of power and frequency. Figure 2-5(b) illustrates the same experimental results, reformulated in energy terms as follows. During each cycle of the power enable signal, energy is drawn from V_{SW} to charge and discharge the gate of the power switch. The switch energy per cycle is

$$E_{switch} = \frac{P_{SW}}{f}$$

Note that the the measurements in Figure 2-5(b) show that the switch energy is generally invariant of t_{off} (but does increase slightly for very short off times—this effect will be explained later).

During the power-on portion of the cycle, the energy consumed by the load is $t_{on}P_{leak}$ (and because the duty cycle is 50 %, $t_{on} = t_{off}$). During the power-off portion of the cycle, the energy consumed by the load is zero (the off-resistance of the switch

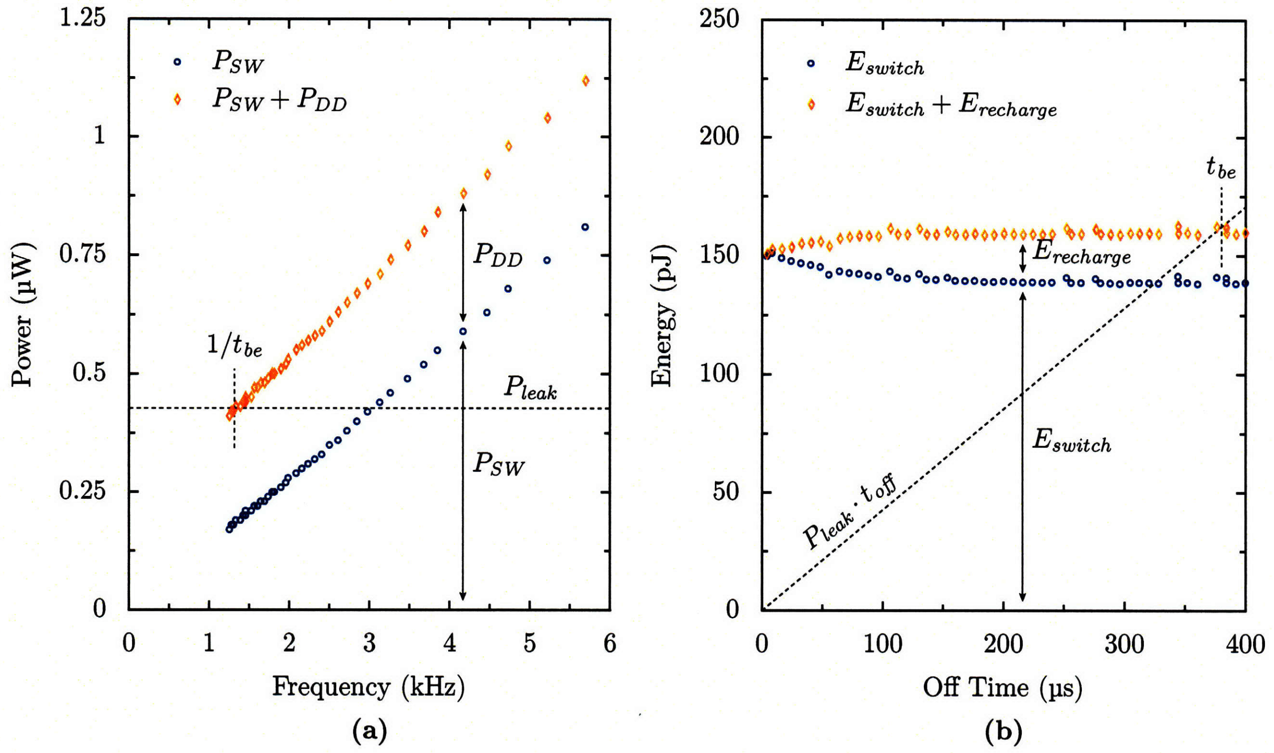


Figure 2-5: Measuring t_{be} experimentally. In (a), the raw power measurements are plotted to show how t_{be} can be determined directly from the experimental results. The break-even point occurs when the total power consumption $P_{SW}+P_{DD}$ while power gating matches the leakage power P_{leak} when no power gating is performed.

In (b), the results of the same experiment are expressed in terms of energy per on-off cycle. The break even point occurs when the switch and recharge energy associated with a given power-off period t_{off} matches the energy $P_{leak}\cdot t_{off}$ that would have been consumed if the load had not been power gated.

is essentially infinite). When the power switch is turned on, a brief pulse of power is drawn from V_{DD} to charge the load back up to the power supply voltage. The integral of this power pulse, $E_{recharge}$ is measurable by calculating the amount by which the energy drawn from V_{DD} per cycle exceeds the amount expected during the power-on portion of the cycle.

$$E_{recharge}=\frac{P_{DD}}{f}-t_{off}P_{leak}$$

As a function of t_{off} or f,

$$\begin{aligned}E_{recharge}(t_{off}) &= 2t_{off}P_{DD}-t_{off}P_{leak}\\ E_{recharge}(f) &= \frac{P_{DD}}{f}-\frac{P_{leak}}{2f}\end{aligned}$$

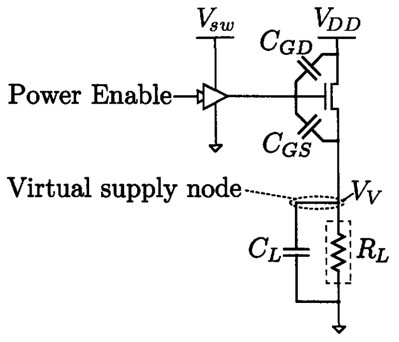


Figure 2-6: Model for power gating circuit

Again, in Figure 2-5(b) the experimental results show $E_{recharge}$ initially grows with t_{off}, but saturates at some asymptotic value.

In energy terms, the break-even time can be defined as the minimum off-time for which the sum $E_{switch}(t_{off}) + E_{recharge}(t_{off})$ is less than the leakage energy saved during the off period.

$$[E_{recharge}(t_{off}) + E_{switch}(t_{off})]_{t_{off}=t_{be}} = t_{be}P_{leak}$$

Figure 2-6 shows a simplified schematic of the power gating circuit, including the main parasitic capacitances, C_{GD} and C_{GS}, of the power switch. The circuitry being power gated has been reduced to a single capacitor C_L, representing the effective power-to-ground capacitance of the virtual power supply node and any internal nodes in the logic-1 state, and a resistor R_L, representing the leakage paths through the load. Subthreshold leakage current in an off ($V_{GS} \approx 0$) transistor is a weakly exponential function of the drain-source voltage, due to drain-induced barrier lowering (DIBL). BSIM3 [30] models the leakage current as

$$I_D = I_0 \exp\left(\frac{V_{GS} - V_T + \eta V_{DS}}{nkT/q}\right)\left(1 - \exp\left(-\frac{V_{DS}}{kT/q}\right)\right)$$

As shown in Figure 2-7 for both single-transistor SPICE simulations and actual measurements from complete circuit blocks, representing subthreshold leakage ($V_{GS} = 0$) as a fixed resistance is only a fairly crude model. The effective resistance $R_L = V_{DS}/I_{DS}$ changes significantly with V_{DS}, particularly for very small V_{DS}. The change is not monotonic: as V_{DS} decreases, R_L initially increases, then sharply decreases.

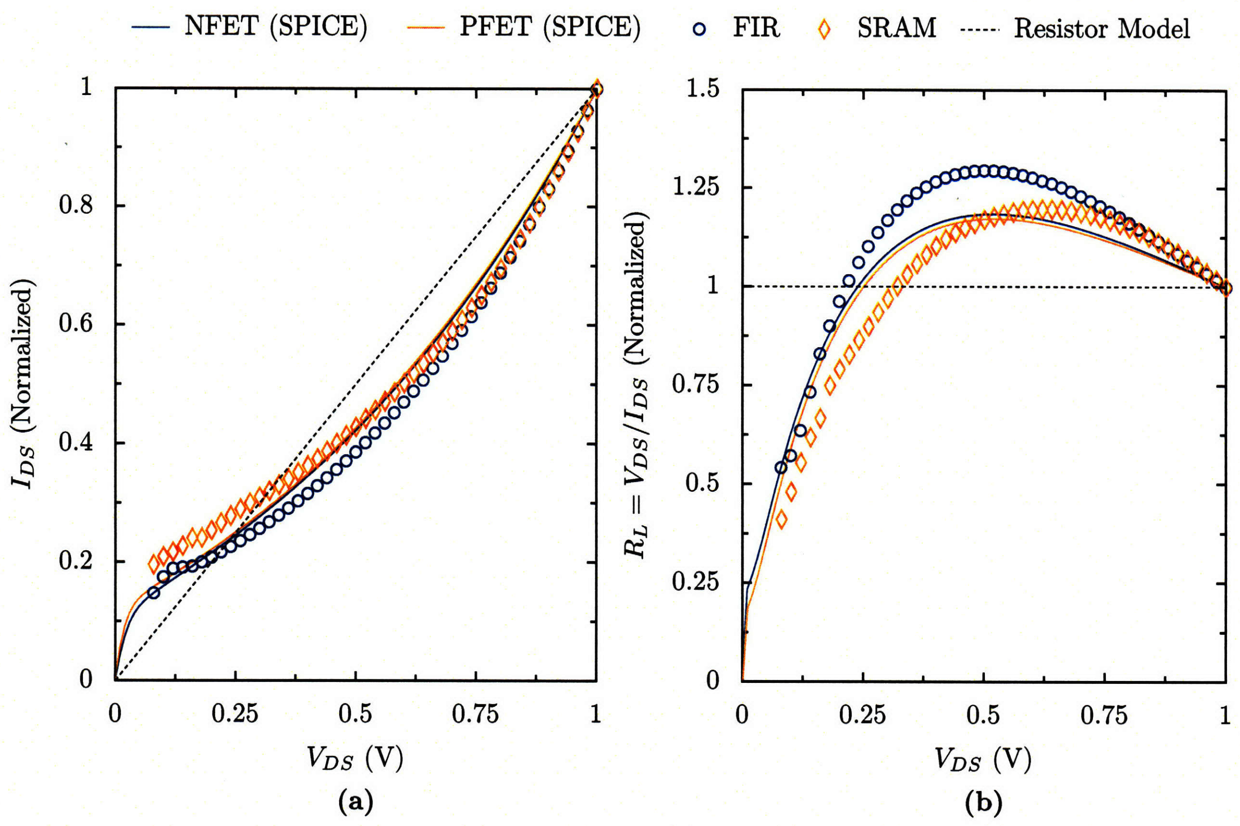


Figure 2-7: (a) Leakage current as a function of V_{DD}. (b) Effective resistance

If we record with an oscilloscope the voltage V_V on the virtual supply node after the power switch is turned off, as shown in Figure 2-8, the non-linearity of R_L is apparent. Compared to an exponential decay curve, the decay of V_V is first slower, then faster, corresponding to a decrease in R_L as V_V decreases. Also apparent in Figure 2-8 is a significant step change in V_V at $t = 0$. The step change occurs when the power switch is turned off, and the step change in the switch gate voltage V_G is coupled to V_V through C_{GS}.

The step change and decay rate of V_V, together with the capacitances C_{GD}, C_{GS}, and C_L explain the changes in E_{switch} and $E_{recharge}$ with respect to t_{off}. To show this, we conduct a step-by-step accounting of the energy transfers that occur during a complete power gating cycle. The resulting model matches the observed behavior of our system, and can be used to extract the capacitance values C_{GD}, C_{GS}, and C_L from the measured data.

A complete power-gating cycle can be broken down into four phases, as shown

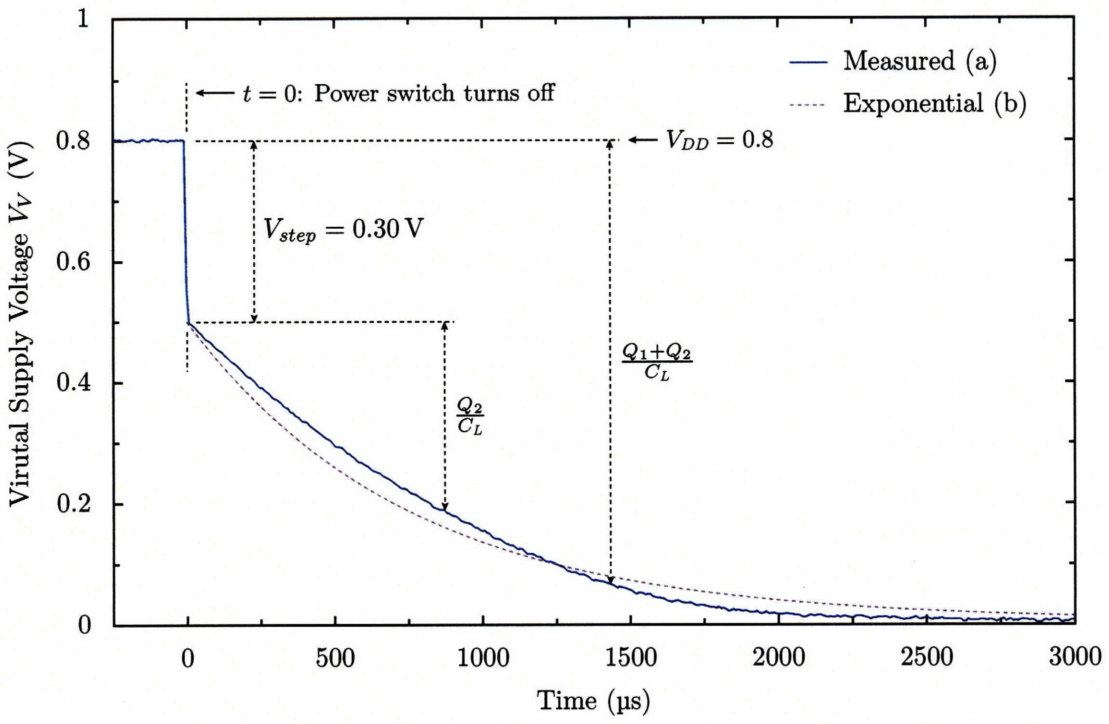


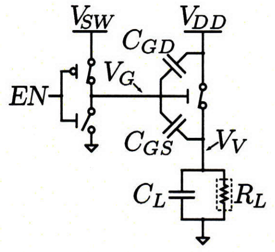
Figure 2-8: Virtual supply node voltage (V_V) decay after the power switch is turned off at $t = 0$. The measured voltage is shown as line (a). The example exponential decay curve (b) shows that the decay of V_V is initially slower, but eventually faster than exponential.

in Figure 2-9. Each phase represents charging or discharging some capacitors in the circuit, and we assume the power-on and power-off periods are long enough that each charge transfer goes to completion (the voltages reach steady-state). Some overlap between the phases is tolerable (and even expected), but due to the superposition principle, this will not affect our energy accounting.

We start with the circuit in the power-on state (phase 0), with the power switch turned on ($V_G = V_{SW}$) and the virtual supply node (V_V) at V_{DD}.

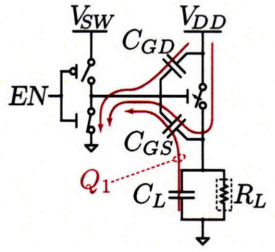
In phase 1, the power switch is turned off. Charge flows to discharge the gate side of the power switch parasitics (C_{GS} and C_{GD}) from V_{SW} to ground. Most of this charge is drawn from V_{DD}, but some of it will come from the load capacitance C_L because the power switch will open before V_G reaches zero. The exact amount of charge drawn from the load cannot be predicted from our simple model, so we simply parameterize it as Q_1. The charge drawn from the load causes the step change of $-\frac{Q_1}{C_L}$ in the virtual supply voltage that was visible in Figure 2-8. The voltage change

Phase 0:
Initial



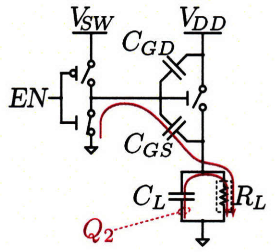
$$\Delta V_V = 0$$
$$\Delta V_G = 0$$
$$Q_{SW} = 0$$
$$Q_{DD} = 0$$

Phase 1:
Disable



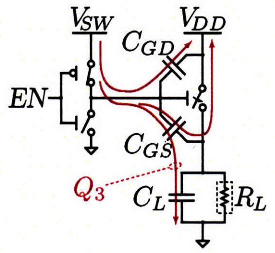
$$\Delta V_V = -\frac{Q_1}{C_L}$$
$$\Delta V_G = -V_{SW}$$
$$Q_{SW} = 0$$
$$Q_{DD} = C_{GD}V_{SW} + (C_{GS}(V_{SW} - \frac{Q_1}{C_L}) - Q_1)$$

Phase 2:
Decay



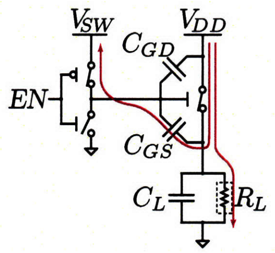
$$\Delta V_V = -\frac{Q_2}{C_L}$$
$$\Delta V_G = 0$$
$$Q_{SW} = 0$$
$$Q_{DD} = 0$$

Phase 3:
Enable



$$\Delta V_V = \frac{Q_3}{C_L}$$
$$\Delta V_G = V_{SW}$$
$$Q_{SW} = C_{GD}V_{SW} + C_{GS}(V_{SW} - \frac{Q_3}{C_L})$$
$$Q_{DD} = -C_{GD}V_{SW} - (C_{GS}(V_{SW} - \frac{Q_3}{C_L}) - Q_3)$$

Phase 4:
Recharge



$$\Delta V_V = \frac{Q_1}{C_L} + \frac{Q_2}{C_L} - \frac{Q_3}{C_L}$$
$$\Delta V_G = 0$$
$$Q_{SW} = -\frac{C_{GS}}{C_L}(Q_1 + Q_2 - Q_3)$$
$$Q_{DD} = (Q_1 + Q_2 - Q_3) + \frac{C_{GS}}{C_L}(Q_1 + Q_2 - Q_3)$$

Figure 2-9: A power-gating cycle, broken down into phases for energy analysis. Initially (phase 0) the power switch is on. In phase 1, the power switch is turned off. In phase 2, the virtual supply node voltage decays due to leakage current through R_L. In phase 3, the power switch is turned on again. In phase 4, the virtual supply node is charged back up to V_{DD}.

on C_{GS} during this phase is $V_{SW} - \frac{Q_1}{C_L}$, and the charge drawn from V_{DD} is $C_{GD}V_{SW}$ (for C_{GD}), plus $C_{GS}(V_{SW} - \frac{Q_1}{C_L}) - Q_1$ (for C_{GS}).

In phase 2, the virtual supply node voltage decays as the charge on C_L and C_{GS} dissipates due to leakage current in the power-gated circuitry. The amount of charge Q_2 lost from C_L depends on how long the circuitry is left powered off, so Q_2 is left as another parameter of the model. The change in V_V is $-\frac{Q_2}{C_L}$.

In phase 3, the power switch is turned on again. Charge is drawn from V_{SW} to charge the switch parasitic capacitors. Some of this charge is pushed back into V_{DD}, offsetting the charge that was drawn from this supply in phase 1. As in phase 1, a portion Q_3 of the charge passing through C_{GS} is absorbed in the load capacitance, causing another step change in the virtual supply voltage of $\frac{Q_3}{C_L}$ volts.

In phase 4, the virtual supply node is charged back up to V_{DD}. The charge that must be returned to C_L is $Q_1 + Q_2 - Q_3$. An additional charge of $\frac{C_{GS}}{C_L}(Q_1 + Q_2 - Q_3)$ must be delivered to C_{GS}, and this charge is in turn reabsorbed by V_{SW}. In reality there is some overlap in time between phases 3 and 4 because the switch turns on before V_G reaches V_{SW}. However, as previously stated, due to the principle of superposition we can analyze the phases as though they occurred sequentially, and the net energy change will be the same.

Totalling up the charge delivered by each supply over the entire cycle, and multiplying by the supply potentials yields the net energy delivered by each supply.

$$E_{switch} = V_{SW}\sum Q_{SW} = (C_{GS} + C_{GD})V_{SW}^2 - C_{GS}V_{SW}\frac{Q_1 + Q_2}{C_L} \tag{2.1}$$

$$E_{recharge} = V_{DD}\sum Q_{DD} = (C_{GS} + C_L)V_{DD}\frac{Q_2}{C_L} \tag{2.2}$$

The charge Q_3 cancels out of both equations. The voltages $\frac{Q_2}{C_L}$ and $\frac{Q_1+Q_2}{C_L}$ can be determined by observing the decay of V_V. $\frac{Q_1+Q_2}{C_L}$ represents the total decay of V_V, $V_{DD} - V_V(t)$ at the moment when the switch is turned back on at time t. $\frac{Q_2}{C_L}$ represents the decay of V_V at time t, minus the step change that occurs at $t = 0$ when the switch is turned off.

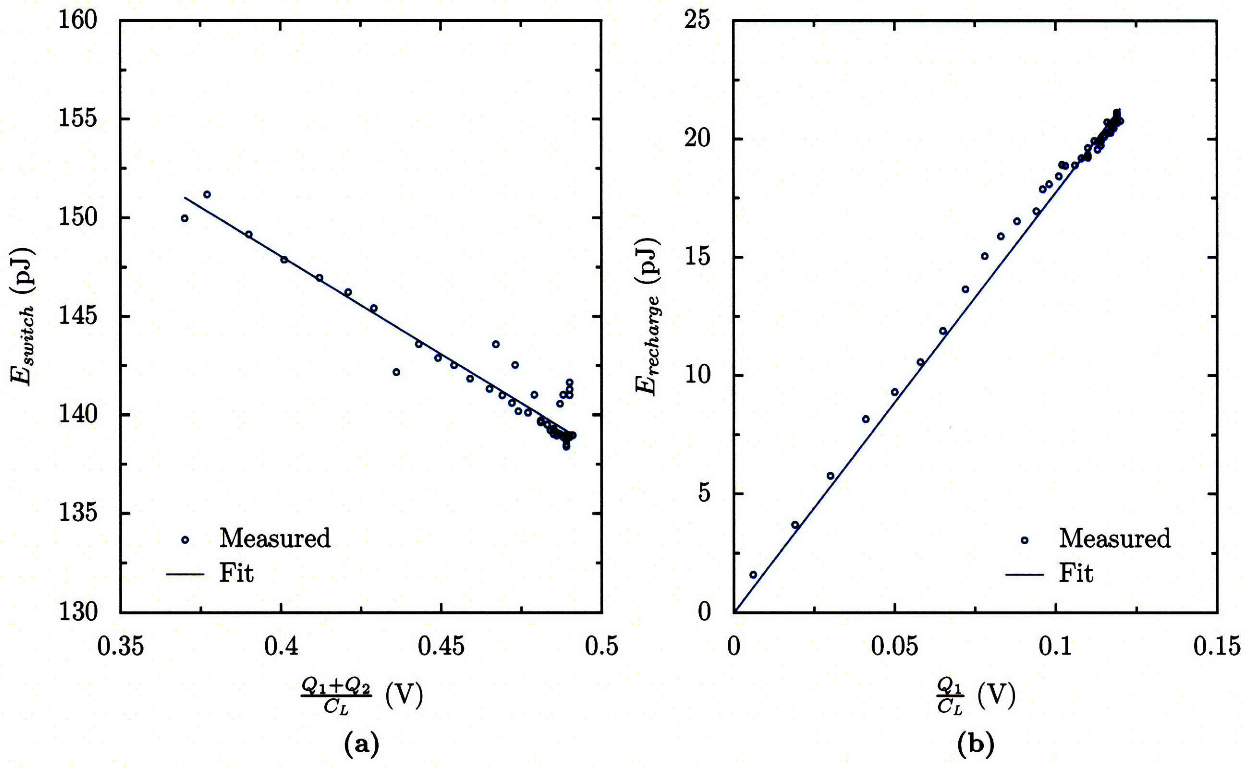


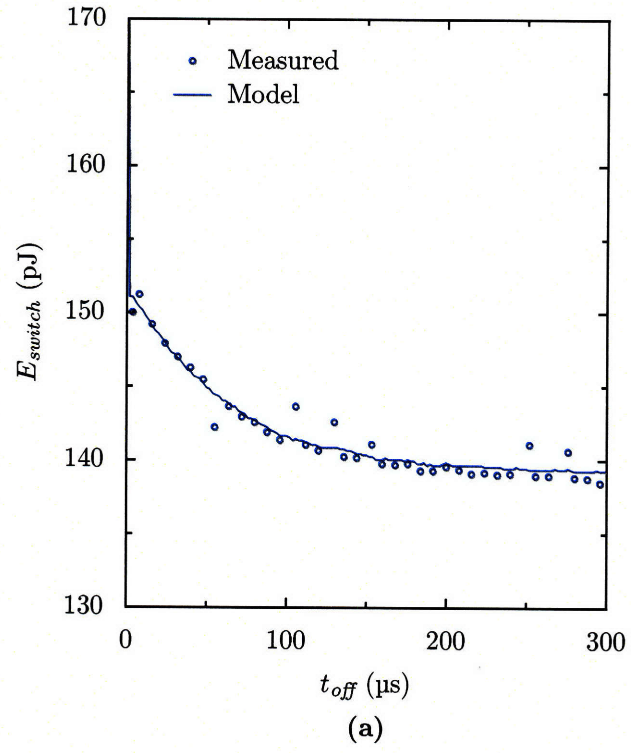
Figure 2-10: Determining C_{GD}, C_{GS}, and C_L by linear fitting.

The capacitances C_{GD} and C_{GS} can be determined from a linear fit of E_{switch} versus $\frac{Q_2}{C_L}$, as shown in Figure 2-10(a). Then, C_L is determined from the average

$$C_L = \left\langle \frac{E_{recharge} - C_{GS}V_{DD}\frac{Q_2}{C_L}}{V_{DD}\frac{Q_2}{C_L}} \right\rangle$$

Figure 2-11 illustrates how well equations 2.1 and 2.2 predict E_{switch} and $E_{recharge}$. Table 2.2 summarizes the data for all of the power-gated domains on the DSP.

Because the decision to use off-chip power switches prevented us from optimizing the switch sizes to the circuits being power gated, our power switches are larger than necessary, resulting in a correspondingly larger than necessary E_{switch}. We end this analysis, therefore, by examining what would happen to the break-even time if E_{switch} were reduced. Figure 2-12 shows that generally t_{be} reduces linearly with E_{switch}. If E_{switch} can be made small enough, then the break-even point occurs before V_V has decayed fully and $E_{recharge}$ has reached its asymptotic maximum. This results in a further reduction in t_{be}, as evidenced by the increased sensitivity of t_{be} to E_{switch} for



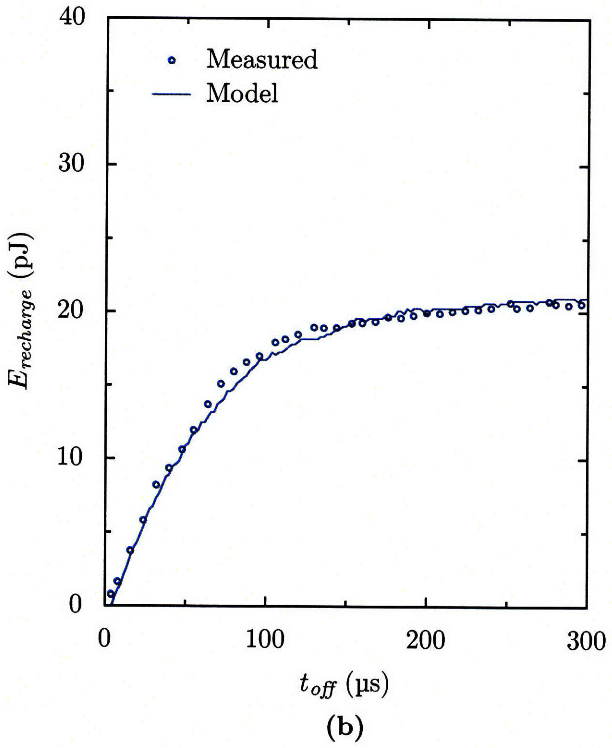


Figure 2-11: Agreement between the model and measurements for E_{switch} (a) and $E_{recharge}$ (b).

Table 2.2: Power gating parameters

		CPU	FIR	FFT	8 kB SRAM	4 kB SRAM
VOLTAGE						
V_{DD}	(V)	0.5	0.5	0.8	0.8	0.8
V_{SW}	(V)	1.1	1.1	1.4	1.4	1.4
POWER						
P_{leak}	(µW)	0.99	0.43	3.02		
ENERGY $(t_{off} = \infty)$						
E_{switch}	(pJ)	155	151	254	253	240
$E_{recharge}$	(pJ)	34	21	243	297	133
TIME						
t_{be}	(µs)	175	380	108	1792	1871
CAPACITANCE						
C_{GD}	(pF)	64	65	57	61	61
C_{GS}	(pF)	92	90	92	87	85
C_L	(pF)	263	265	293	318	221

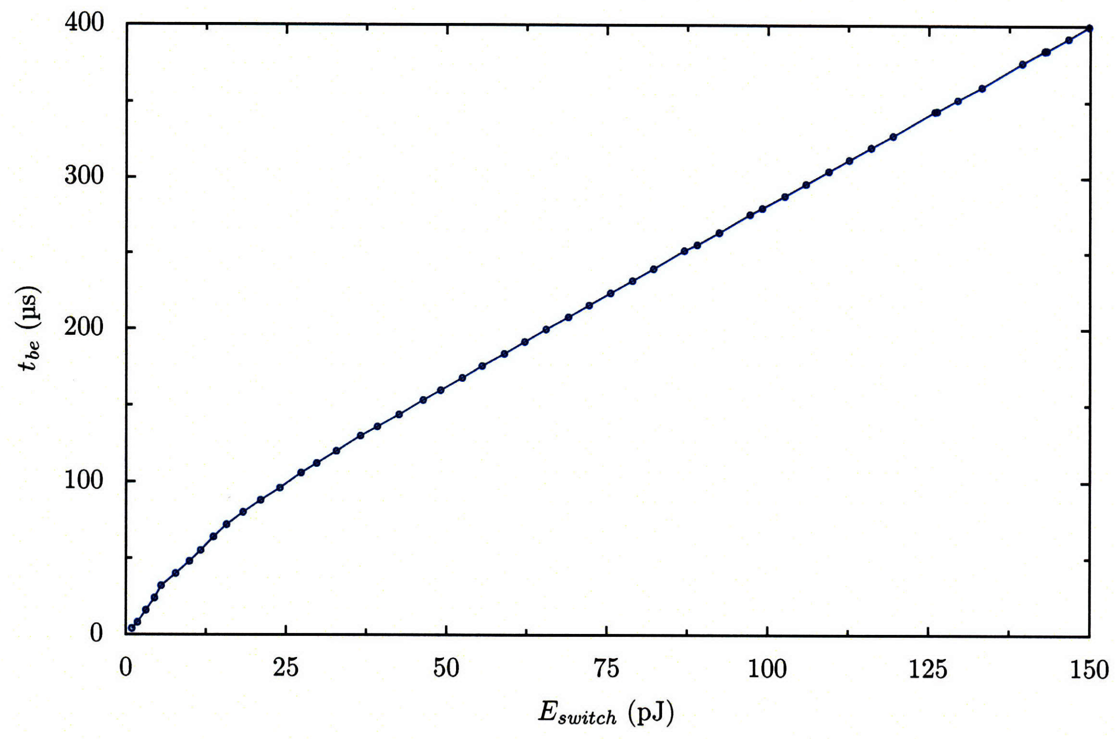


Figure 2-12: t_{be} as a function of E_{switch}.

$E_{switch} < 25\,\text{pJ}$ in Figure 2-12.

2.3 Simulations of On-Chip Switches

To estimate the break-even times that could have been achieved using on-chip switches, the design of an on-chip switch for the CPU power domain was analyzed in simulation. An NFET was used in a footer configuration (between the power-gated module and ground) as shown in Figure 2-13, to maximize the gate-source voltage that could be applied to the switch without exceeding the oxide breakdown voltage.

Choosing an optimal size for an on-chip power switch transistor requires balancing off-mode leakage power (a smaller switch will have less leakage when turned off) and performance degradation (a smaller switch will have a larger voltage drop when turned on) [28]. From the derivative of the frequency-versus-voltage curve in Figure 1-6, in order to limit the performance degradation at an operating voltage of 0.5 V the voltage drop across the power switch must be less than 1 mV.

A gate drive voltage of 1.0 V was selected for the power switch, because this voltage

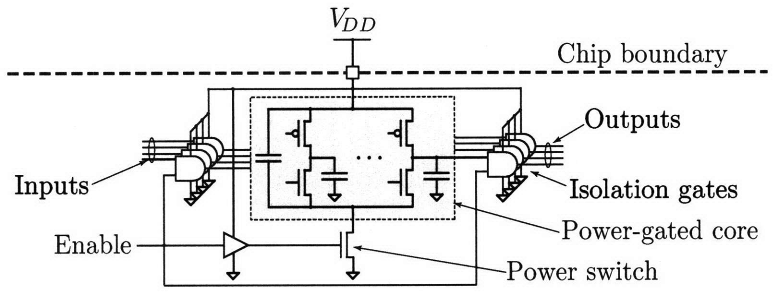


Figure 2-13: Power gating implementation with on-chip power switches.

is required by the I/O pad drivers, and is therefore already available. Two threshold voltages are available in the our process technology: standard (SVT) and high (HVT). We consider power switch implementations using both threshold voltages.

From transistor-level simulation, the peak instantaneous current drawn by the CPU is roughly 1 mA at $V_{DD} = 0.5\,\text{V}$. The on-resistance of the power switch can therefore be no more than 500 Ω to limit the voltage drop to 1 mV. An HVT power switch must be 44 % wider than a SVT switch with the same on-resistance, which results in a corresponding increase in E_{switch} for the the HVT switch, as shown in Figure 2-14. However, the leakage current for the HVT switch is significantly lower than for the SVT switch. When powered off, the leakage power of the CPU domain is reduced by 580 × when an HVT switch is used, compared to 87 × for an SVT switch. The break-even time for the HVT switch is only 12 % longer than for the SVT switch (Figure 2-14). For both switches, the break-even time occurs well before the virtual-ground node has fully (dis)charged

2.4 Conclusions

Power gating is implemented on the µAMPS DSP at the module level, allowing individual memory banks, accelerator cores, or even the CPU to be powered off if they will be idle for an extended period. Discrete, external NFETs are used for the power switches, which offer extremely high off-resistance but require significant energy to turn on and off for each power gating cycle. The switch energy directly impacts the break-even time: the minimum time that a module must remain powered-off in order

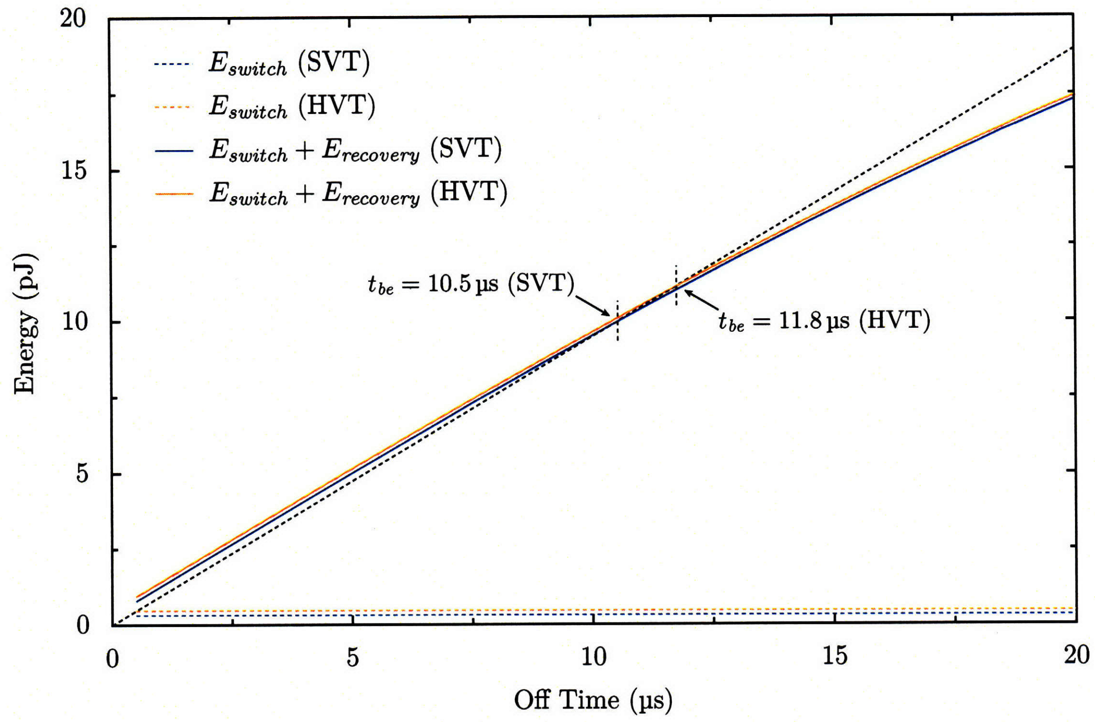


Figure 2-14: Break-even time for on-chip switches.

to save any net energy. Our power-gating implementation is therefore well suited for applications with "bursty" workloads, where idle times are long and minimizing idle-mode power consumption is most important, and is less suited for low-rate, continuous workloads, where the agility to take advantage of brief idle periods in between computations is most important.

We have described a method for measuring break-even time experimentally, and an energy model based on the experimental results. For our module-level power gating implementation, we measured break-even times ranging from hundreds of microseconds for logic domains to milliseconds for memory domains (which have much lower leakage than the logic domains, due to the use of high-threshold voltage devices for the memory bitcells). These timescales, corresponding to hundreds or thousands of clock cycles at 5 MHz are appropriate for controlling power-gating through software, which should permit the design of more sophisticated control algorithms than could be implemented purely in hardware.

Chapter 3

CPU

At the center of the µAMPS DSP is a general-purpose processor core, with a custom instruction set architecture (ISA). We opted to create our own ISA, rather than co-opt an existing one (and its toolchain) from a commercial processor because no existing architecture we found had all the attributes we consider important for an efficient, micropower processor. No commercial processor (with an openly documented ISA) we know of achieves close to 10 pJ per instruction.

An ISA is only as good as its compiler, and the more radical the ISA, the more radical the compiler must be to generate efficient code for it. In order to focus on circuit design, rather than compiler design, we modified the standard GNU tools (GCC, GAS, etc.) to support our architecture. GCC performs most of its optimizations independent of the code-generation back end, so even a simple back end inherits sophisticated optimization passes.

This chapter begins with a survey of existing commercial and academic processor architectures, from which many ideas in the µAMPS ISA were inspired or borrowed. The key features of the µAMPS ISA are then described. (A full reference of the µAMPS instruction set can be found in Appendix A.) Finally, performance benchmark and energy profiling results for the CPU are presented.

Table 3.1: Comparison of processor architectures

	Commercial					Research				
	8051[a]	AVR[b]	MSP430[c]	Cortex M3	M·CORE	Smart Dust [11]	SNAP/LE [12, 31]	Subliminal 2 [32]	Sub-V_{th} MSP430 [23]	µAMPS
PROCESSOR SPECIFICATIONS										
Data width (bits)	8	8	16	32	32	12	16	8	16	16
Inst. length (bits)	8–24	16–32	16–48	16–32	16	17	16–48	12	16–48	16
Cycles per inst.	1–8	1–5	1–5	1–3+	1–2	1	1–2	1	1–5	1
Registers[d]	2+1	32	12+4	8+8	16		15+1	8	12+4	8+8
Branch offset (bits)	8	7	10	8/11/22	11			6	10	11
Multiplier	no	yes	no[e]	yes	yes	no	no	no	no	yes
Barrel shifter	no	no	no	yes	yes	no	yes	yes	no	yes
Pipeline stages		3?	3	3	4		na[f]	3	3	3
Voltage (V)	1.8	1.8	1.8			1.0	0.6	0.2	0.5	0.5
Process tech. (nm)				90	250	250	180	130	65	90
Gate count				33–60k		3.7k				6.5k
Clock Freq. (MHz)	25	4	4.15	50	80	0.50	28	0.14	0.43	4.0
Energy/inst. (pJ)				40		12		0.66		2.9
ADDITIONAL SPECIFICATIONS (COMPLETE CHIP)										
Prog. mem. (bytes)	64k	64k	4k			2k[g]	4k		16k[h]	60k[h]
Data mem. (bytes)	4k	4k	256			1k	4k			
Energy/cycle (pJ)	300	720	350	47	250		24		27	10.3

[a]Silicon Labs C8051F93x series.
[b]Atmel ATmega644P
[c]Texas Instruments MSP430C1121A
[d]n+m indicates n general purpose registers and m special purpose registers
[e]Some other models include a hardware multiplier as a memory-mapped peripheral.
[f]Not applicable: asynchronous design.
[g]Program memory is 1k 17-bit words.
[h]Combined program/data memory.

3.1 Survey of Related Processor Architectures

Table 3.1 surveys a range of processor architectures comparable to our DSP (either in performance or power consumption), including a number of commercial processors commonly used for microsensor applications, and a number of custom architectures from the literature. Regrettably, there are many holes in the table where data was not available.

The Intel 8051 is the oldest architecture (originally released in 1980) in the table, and is the only accumulator-based architecture in a field of load-store machines. Despite its age, the 8051 remains a popular architecture, and multiple low-power implementations of it have been produced, such as the Silicon Labs part described in Table 3.1, which operates down to 0.9 V (although power numbers are only published for 1.8 V).

The Atmel AVR and Texas Instruments MSP430, both of which have a Harvard architecture (like the 8051), were once by far the most popular processors for microsensor systems (e.g., [10]). Both are designed to be C compiler friendly (and, unlike the 8051, are supported by the GCC compiler), with large register files. Cortex-M3 (based on the ARM Thumb 2 instruction set) and M·CORE are 32-bit datapath architectures, but with 16-bit instruction sets.

Among the custom microsensor processor architectures, Smart Dust [11] is the earliest. Power-reduction features of this chip included extensive clock gating, guarded ALU inputs to reduce glitches, a short pipeline (no pipelining of datapath), and single cycle per instruction operation. Smart Dust operates at 1 V, which is lower than normal for a 0.25 µm process, but well above subthreshold.

The remaining custom architectures in the table, except for the µAMPS DSP, all operate at or below V_{th}. SNAP/LE [12, 31] is an asynchronous architecture, with an interesting message coprocessor that performs sophisticated event handling without need of a software operating system. Subliminal 2 is a second generation architecture by Nazhandali et al., which claims the lowest energy per instruction, lowest voltage, and lowest clock speed in our table. Between the first and second generations, Subliminal changed from variable-length instructions to fixed length and from accumulator to load-store, and reduced the energy per instruction by a factor of three (in the same process). In designing the µAMPS ISA, we independently arrived at these same decisions.

Finally, an ultra-low-voltage version of the MSP430 architecture by Kwong et al. [23] illustrates how subthreshold operation can bring a commercial architecture down to the tens of picojoules per instruction range.

3.2 µAMPS CPU Architecture

To achieve low energy per instruction, our primary design strategy was to minimize the complexity of the control logic in the processor. To this end, two key design tenets were established at the beginning of the ISA design process.

- All instructions execute in one clock cycle (CPI=1)
- All instructions have the same 16-bit length.

Executing every instruction in one clock cycle results in a simple pipeline model, where one instruction is issued on every cycle, and instructions are retired in order. Since the instruction length matches the memory width, one instruction can be fetched on every cycle (except for one rare condition, in which case a no-op instruction is injected into the pipeline while the instruction fetch unit is stalled for one cycle[1]). This is similar to the M·CORE architecture, except that M·CORE allows some exceptions (loads, sores, multiplication, division, and branches) to the one-cycle-per-instruction rule. There is no branch delay slot in the µAMPS ISA, and bypassing is used to prevent load delays, so the compiler need not perform any instruction scheduling.

A second design goal was to minimize the number of data memory accesses. As will be seen from the energy profiling results in Section 3.3, load and store operations consume roughly twice as much energy as other instructions. We selected a load/store architecture in an effort to reduce memory traffic. Stack-based, accumulator, and register-memory architectures all perform at least one memory access per ALU operation. The µAMPS register file comprises eight general-purpose registers, plus an additional eight special-purpose registers (multiplier results, stack pointer, subroutine/exception return address, etc.) usable only by a subset of all instructions (similar to the Thumb ISA). Figure 3-1 illustrates the programming model. Instructions are provided to copy data between general- and special-purpose registers. Other

[1]Such stalls occur when a data memory operation from a load or store instruction accesses the same memory bank where instructions are being fetched from. Since each memory bank can perform only one memory operation per cycle, instruction fetch is stalled for one cycle while the data memory read or write is performed. This condition is easily avoided by keeping instructions and data in different memory banks.

General Purpose Registers

r7	General Purpose
r6	General Purpose
r5	General Purpose
r4	General Purpose
r3	General Purpose
r2	General Purpose
r1	General Purpose
r0	General Purpose
	15 0

Special Purpose Registers

r15	Condition Codes
r14	Exception Return Pointer
r13	Stack Pointer Limit
r12	Stack Pointer
r11	Linkage Pointer
r10	Multiply Result 2
r9	Multiply Result 1
r8	Multiply Result 0
	15 0

Program Counter

pc	
	15 0

Figure 3-1: Programmers model

instructions (such as stack operations and multiplications) are hardwired to use specific registers, thus saving space in the instruction encoding. The program counter is a separate register, only modifiable by branch and jump instructions.

The processor contains three functional units: an ALU implementing add, subtract, and bitwise logical operations (AND, OR, XOR, NOT), a barrel shifter, and a multiply-accumulate (MAC) unit. The MAC consists of a 16×16-bit single-cycle multiplier and a a 48-bit accumulator register. The accumulator is readable and writable as special purpose registers r8, r9, and r10.

3.2.1 Instruction Set Organization

Instructions are encoded in only eight different instruction formats, as shown in Figure 3-2. Fields common to multiple formats appear in the same place in each format. Because most instructions can use only the eight general-purpose registers as operands, there is sufficient space in the instruction encoding to employ three-operand instructions for the most common arithmetic and logical operations (add, subtract, AND, OR, XOR), thereby reducing the number of register-to-register copy instructions necessary. In close agreement with [32], we found that the advantage of 3-operand instructions versus 2-operand instructions (where one of the source operands is over-

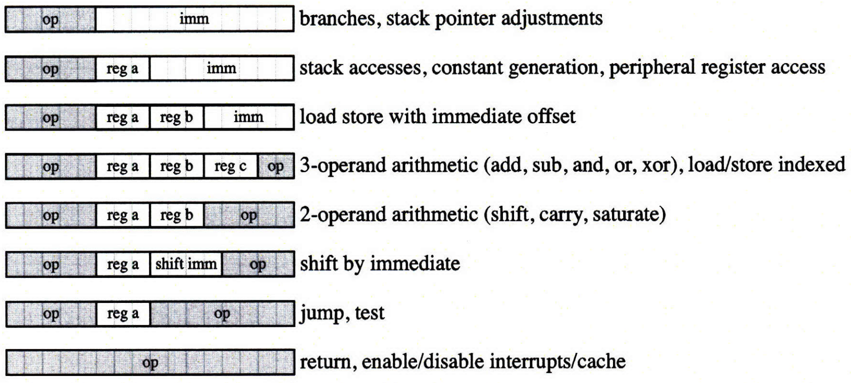
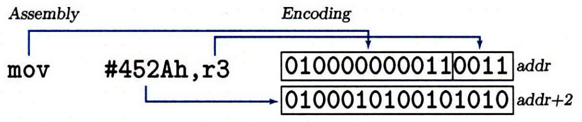

op | imm — branches, stack pointer adjustments
op | reg a | imm — stack accesses, constant generation, peripheral register access
op | reg a | reg b | imm — load store with immediate offset
op | reg a | reg b | reg c | op — 3-operand arithmetic (add, sub, and, or, xor), load/store indexed
op | reg a | reg b | op — 2-operand arithmetic (shift, carry, saturate)
op | reg a | shift imm | op — shift by immediate
op | reg a | op — jump, test
op — return, enable/disable interrupts/cache

Figure 3-2: Instruction encoding

written with the result of the computation) is about 10 %, both in code size and execution time.

In some cases, limiting every instruction to one 16 bit word and one cycle required splitting instructions that would have otherwise required multiple words or multiple cycles into multiple, individual instructions. For example, on some architectures, such as the MSP430, loading a 16-bit constant into a register is a two-cycle, two-word instruction, with the first word of the instruction specifying the load constant operation, and the second word containing the constant value to load.

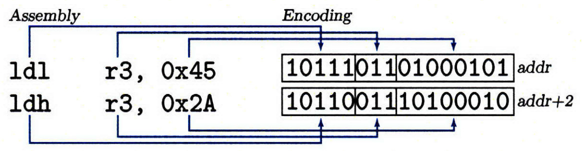
Load 0x452A into r3:



```
mov    #452Ah,r3    010000000011|0011
                    0100010100101010
```

In the µAMPS instruction set, this task is implemented using two separate instructions, each of which contains an 8-bit immediate field and loads half of the constant. (One instruction loads the MSBs and the other the LSBs.)

Load 0x452A into r3:



```
ldl    r3, 0x45    10111|011|01000101
ldh    r3, 0x2A    10110|011|10100010
```

Besides simplifying the pipeline control logic, an additional benefit of this approach is that 8-bit signed constants can be loaded in only one cycle.

The use of a dedicated stack register enables the implementation of special stack operation instructions (load/store from stack, increment/decrement stack pointer) with larger constant fields than could be encoded if an arbitrary general purpose register was used as the stack pointer. Special load and store instructions provide direct access to the data memory address range encompassing the control registers for the on-chip peripherals, eliminating address generation overhead for these frequently accessed locations.

For a complete reference of the µAMPS instruction set, see Appendix A.

3.2.2 Pipeline

The µAMPS CPU employs a three-stage (fetch, execute, and write back) pipeline, illustrated in Figure 3-3. This was the shortest possible pipeline possible, due to the built-in output registering of the SRAM macros used for data and program storage. A four stage pipeline was investigated, but was abandoned before complete, as energy estimates indicated it would consume about 50 % more energy than the three-stage design, not counting significant additional logic required to properly handle interrupts in the longer pipeline. Nazhandali et al. also found three stages to be optimal [32]. Increasing the pipeline depth potentially decreases the energy consumed per instruction by shortening the critical path (thereby allowing more instructions to be executed per second, mitigating the effect of leakage) and by reducing glitching. However, these effects are offset by increased dynamic power, due both to the flip-flops required to implement the additional pipeline registers and to the additional logic required to handle stalls, bypassing, and exceptions in the more complex pipeline.

3.3 Measurements and Benchmarking

Unfortunately, there is to date no standard suite of benchmarks for microsensor processors. Nor does there seem to be an established benchmark suite for microcontrollers in general. An obvious reason is that microsensor and microcontroller programs tend to involve extensive, low-level interaction with nonstandard on-chip peripheral devices, and thus porting them to other architectures is more complicated than simply recompiling the source code.

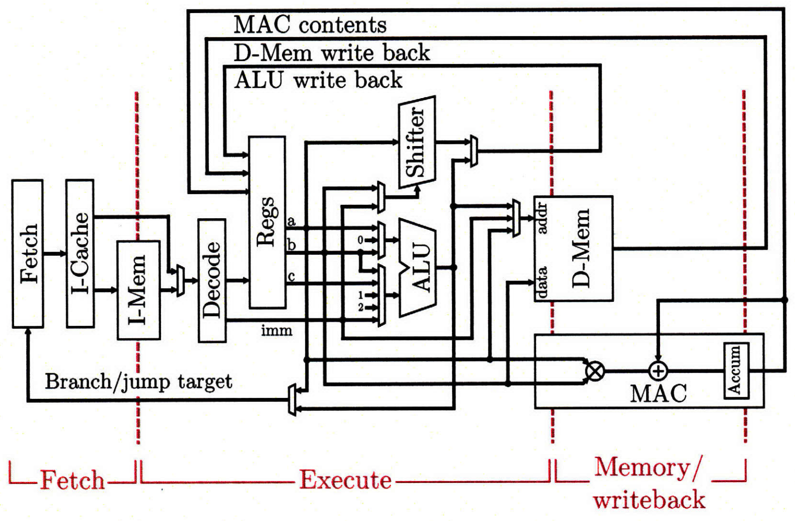


Figure 3-3: CPU pipeline architecture

We selected a few small, computational tasks that could be compiled and run on most of the commercial architectures mentioned in Section 3.1 as a makeshift benchmark suite for evaluating the µAMPS ISA. The benchmark tasks are examples of tasks that a microsensor application may be required to perform, such as error checking, data compression, and encryption. Comparison of code densities and execution times with the AVR, MSP430, Thumb, and M·CORE architectures, as shown in Figure 3-4), demonstrates that our instruction set matches up well with the established, commercial instruction sets, despite the fact that little time has been spent so far optimizing the µAMPS compiler back end for best code generation. For this test we used the GCC compiler[2] for each architecture. Cycle counts are from the simulators built into the GNU debugger (GDB), except for the AVR architecture, for which we used Avrora [33]. Code density was measured when optimizing for space (`-Os`), execution time when optimizing for speed (`-O3`). Thumb and M·CORE, with their 32-bit datapaths, have a significant advantage for the **tea** benchmark, which mostly consists of 32-bit shift operations. Otherwise, the µAMPS architecture clearly performs on par with the commercial instruction sets.

The energy consumption of each instruction in the instruction set can be measured, as described in [34], by measuring the CPU power consumption while it executes the

[2]Version 3.2.3 for the MSP430 and version 4.0.3 for all other architectures

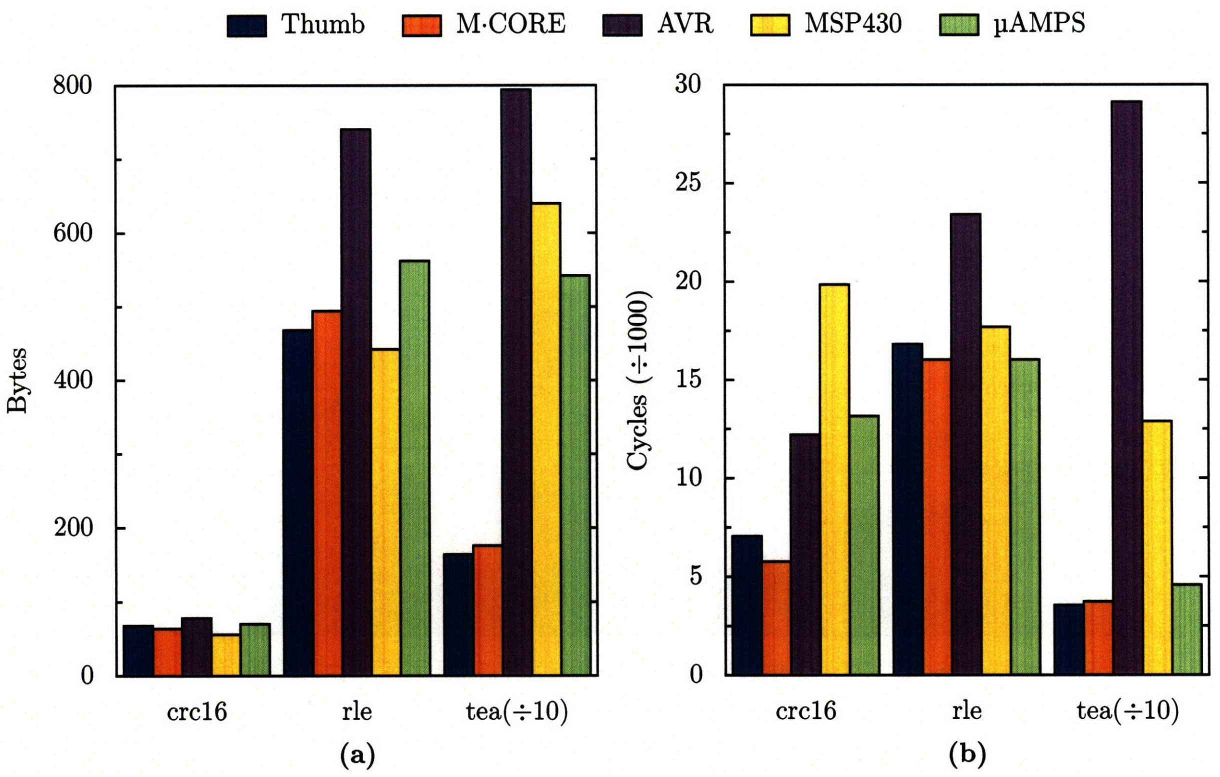


Figure 3-4: Comparison with commercial architectures, in terms of code density (a) and execution time (b).

target instruction over and over in a loop. We used loops consisting of about 30 copies of the target instruction (with varying operands), followed by a branch back to the beginning of the loop. An important limitation of this method is that it does not accurately characterize the instruction decode energy, since the same instruction is being executed over and over again, but we can still detect interesting trends in the data.

Figure 3-5 illustrates the instruction energy profiling results for the µAMPS CPU. Some instructions could not be profiled because they could not be placed in a loop (e.g., return from subroutine, return from interrupt, software interrupt). The instructions have been divided into groups, based on function. The branch instructions were profiled twice, differentiating taken and untaken branches. Untaken branches are among the least energy-expensive instructions, on par with the truly trivial instructions such as `nop`, or instructions which simply set or clear one bit in the condition code register. Taken branches consume roughly the same energy as arithmetic op-

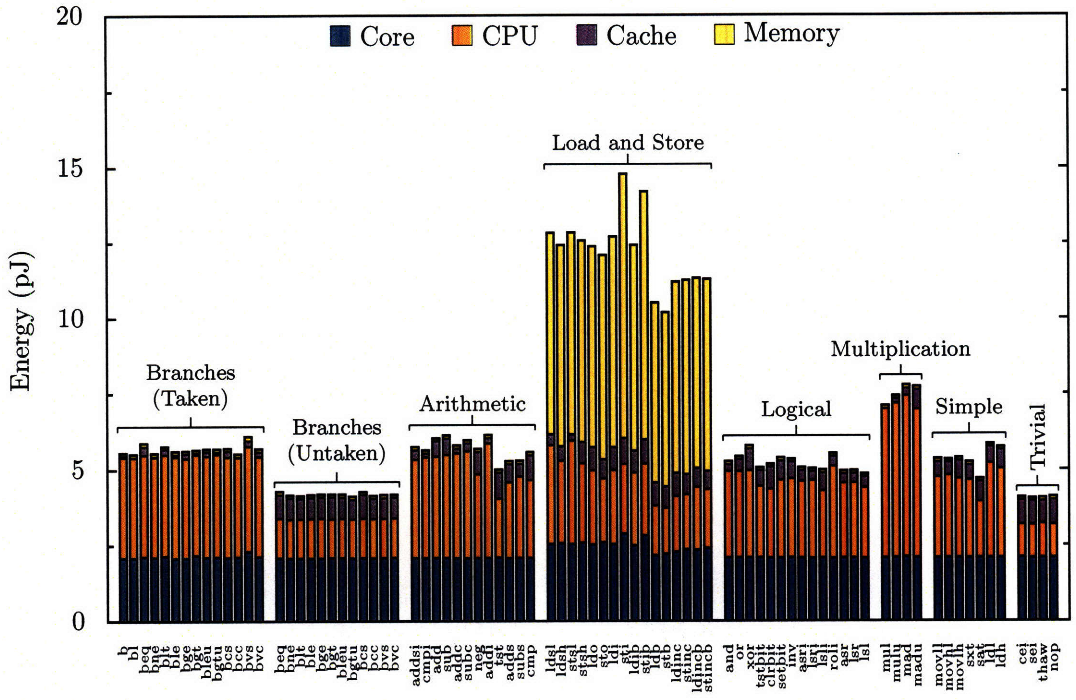
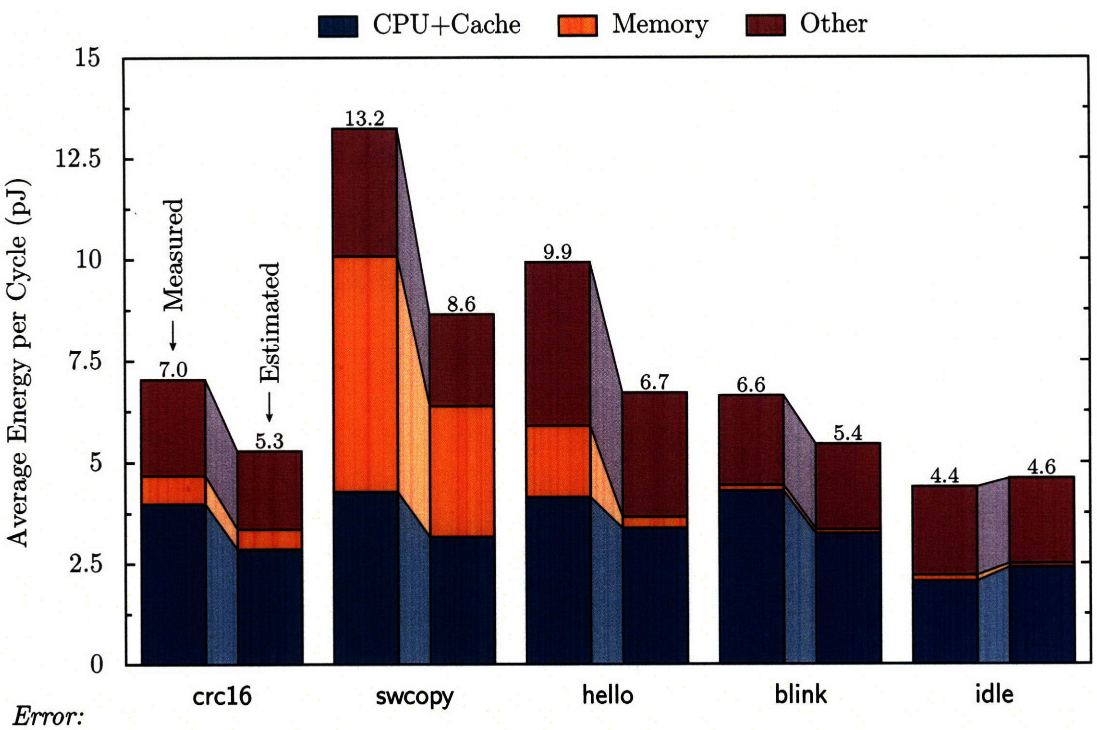


Figure 3-5: Variation in instruction energy across the µAMPS instruction set.

erations (as expected, since the ALU is used to generate branch targets), logical operations, and other simple instructions such as copying data between registers. Multiplication instructions are somewhat more energy expensive than instructions that use the ALU or shifter. Finally, for load and store instructions, the data memory access energy more than doubles the energy cost relative to other instructions.

Approximately 2 pJ of the energy per instruction consumed in the core power domain (memory arbiter and on-chip peripherals) is attributable to global clock tree power and could be at least partially eliminated with better clock gating in a future version of the chip. We relied entirely on automatically inserted clock gating, which only affects the last few levels of the clock tree, leaving much of the global clock tree toggling on every cycle.

The instruction profiling results can be used to estimate the energy consumption of a given program from an instruction trace of that program obtained from an instruction-level simulation—though the accuracy of such an estimate is limited, because inter-instruction effects are not accounted for. Figure 3-6 compares estimated and measured average energy per instruction for five benchmarks. All five



Error:	crc16	swcopy	hello	blink	idle
CPU+Cache	−28 %	−26 %	−19 %	−25 %	17 %
Memory	−29 %	−45 %	−84 %	−27 %	−33 %
Core	−18 %	−29 %	−24 %	−4 %	−4 %
Total	−25 %	−35 %	−32 %	−18 %	5 %

Figure 3-6: Comparison of measured and estimated (from the instruction profiling results shown in Figure 3-5) for five different benchmarks.

benchmarks are small enough to fit entirely within the instruction cache, so cache miss energy need not be accounted for. In general, the energy per instruction is underestimated by 20 − 30 %.

3.4 Conclusions

Based on our survey of existing architectures, low-voltage, or even subthreshold operation is essential for achieving sub-10 pJ per instruction efficiency, but architecture also has an essential role to play in achieving the lowest energy per instruction.

The principle characteristic of the µAMPS ISA design is a strict adherence to a fixed-length, single-cycle instruction set, simplifying instruction decoding and the pipeline control logic. Benchmark comparisons show that this was accomplished while achieving code density and execution time on par with comparable commercial architectures. The CPU alone consumes roughly 3 pJ per instruction, with multiplications requiring slightly more energy, and trivial operations like untaken branches requiring somewhat less. Due to the additional memory access energy, load and store instructions consume roughly twice as much energy overall as all other instructions.

Chapter 4

Memory Architecture

Memory is the dominant source of both leakage and dynamic power consumption in modern low-power embedded microprocessors. Of the 6.3 million transistors on our DSP die, 95 % are in SRAM memories, making memory the dominant source of leakage power. As Figure 3-5 illustrated, load and store instructions consume twice as much energy as other instructions, just due to the memory access energy. Targeting both the standby power and access energy of the memory system is therefore one of the most effective means of reducing the overall system energy usage.

In this chapter we consider two different mechanisms for reducing memory power in micropower processors: 1) instruction caching, and 2) power-gated, multi-bank memory. While both these techniques are also used in higher-power systems, the constraints of a sub-10 pJ/instruction class system require a different approach to their implementations. The already low access energy and minimal latency of on-chip memory leaves room for only an extremely small cache. We therefore consider carefully how to best manage small caches, with an emphasis on minimizing energy instead of maximizing performance.

4.1 DSP Memory Model

Due to the prohibitive energy cost of off-chip I/O (estimated at tens of picojoules per pin per transition, not counting static power), all memory for the DSP is implemented on chip, in the form of SRAM. SRAM is used for both data and program storage. On power-up, a bootloader program copies program data from an external serial EEPROM into the on-chip memory for execution.[1] The bootloader program is stored in a 138-word ROM (implemented using standard cell logic).

Because instructions and data are both stored in SRAM, we implemented one unified memory space, instead of separate instruction and data spaces. This creates significant flexibility by not forcing a fixed partitioning of memory between instructions and data. Additionally, there is no need to implement special load and store instructions for accessing constant data stored in instruction memory. The principle disadvantage of a unified memory—the so-called "von Neumann bottleneck" that occurs when instruction fetching is stalled to accommodate load or store operations—does not affect our design, because the banked memory architecture (described in Section 4.3) allows data and instruction accesses to occur simultaneously, as long as they target different memory banks.

The 16-bit datapath width of the DSP led us to adopt a 16-bit address space. This results in a maximum 64 kB of addressable memory, but eliminates the need for extended-precision registers and functional units in the processor for storing, generating, and manipulating address values.[2] The DSP provides 60 kB of combined program and data memory; the final 4 kB of the address space is used for the boot ROM and for memory-mapped control registers for the on-chip peripherals. Figure 4-1 illustrates the DSP's complete address space.

4.2 Instruction Caching

The impact of caching on memory system power is well known in PCs and embedded processors, but very few microcontrollers or micro-power processors to date incorporate caches. We surmise that this is because such systems generally contain only small,

[1]Program data can also be loaded using an RS-232 interface, or through the GPIO pins (for example, using a pattern generator).

[2]For an example of a comparable system which uses 24-bit addresses to address up to 16 MB of memory (mostly off-chip), see [35].

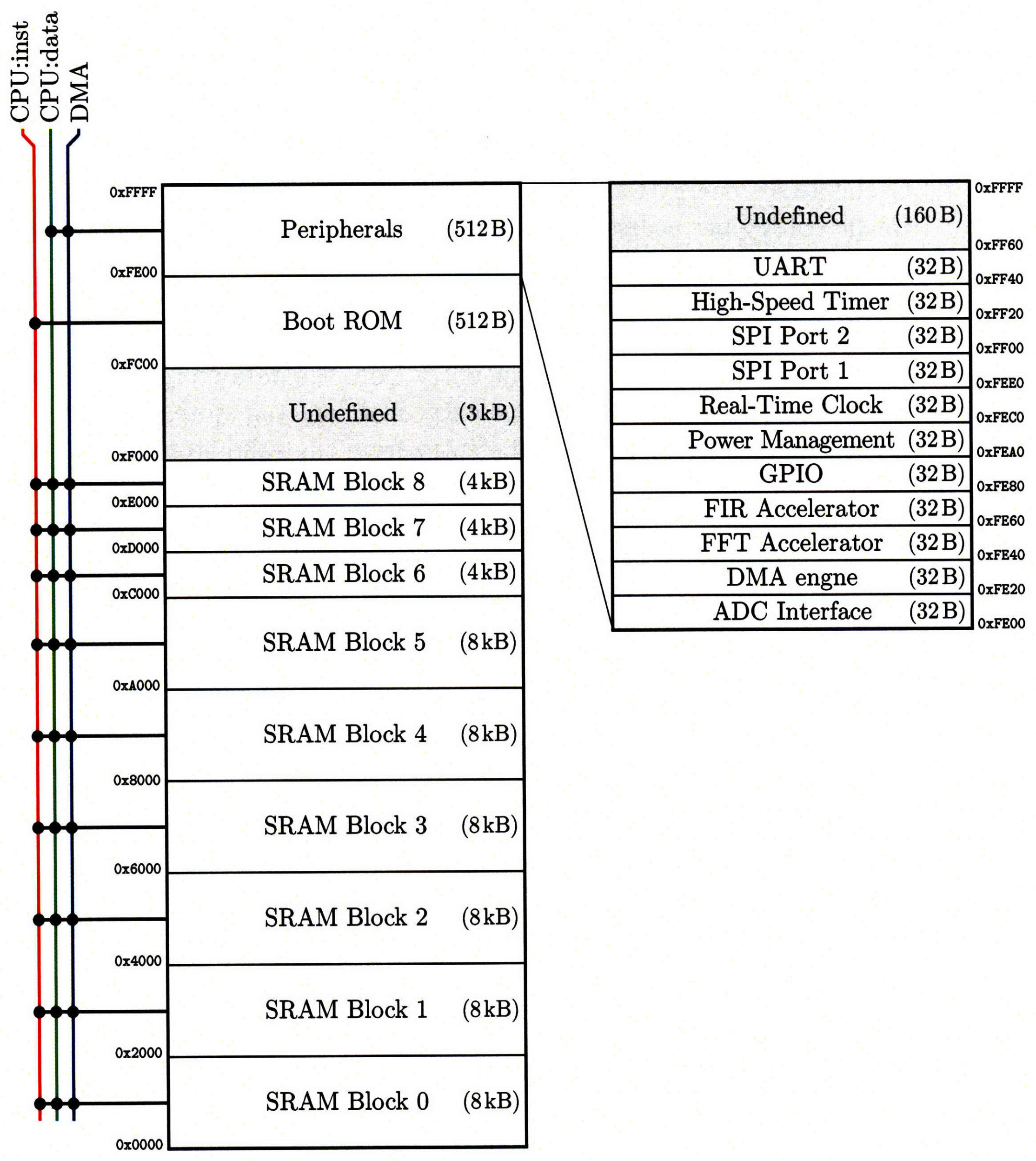


Figure 4-1: DSP memory map. Of the 64 kB address space, 60 kB is dedicated to general-purpose SRAM. The remaining addresses are assigned to the bootloader ROM and memory-mapped control registers for the on-chip peripherals.

on-chip memories, which already have single-cycle access times, and therefore derive no performance benefit from caching. As we will show, the lack of performance improvement does not mean we cannot achieve an energy improvement through caching. In fact, the lack of a performance impact permits the use of energy optimizations that would normally result in unacceptable performance decrease.

In this section, we specifically explore the design of a instruction caches for mi-

Table 4.1: Variables for memory architecture analysis

Name	*Description*
a_{read}	Ratio of the read energy of a memory to the length of the memory
E_{access}	Cache access energy for a cache hit
E_{cache}	Cache energy per access (average, including energy due to cache writes)
E_{fetch}	Instruction fetch energy
E_{mem}	Memory access energy
$E_{read}(\ell)$	Energy required to read one word from an ℓ-word memory
E_{data}^{read}	Energy consumed by reading one entry from the data store of a cache
E_{tag}^{read}	Energy consumed by reading one entry from the tag store of a cache
E_{valid}^{read}	Energy consumed by reading one entry from the valid store of a cache
$Etag$	The energy cost of changing the tag of one line of a cache, including the energy to update the tag and valid stores
$Ewrite$	Energy consumed writing a word of data into the cache (and updating the associated valid bits), assuming a tag match occurred
E_{data}^{write}	Energy consumed by writing one entry to the data store of a cache
E_{tag}^{write}	Energy consumed by writing to one entry in the tag store of a cache
E_{valid}^{write}	Energy consumed by setting one valid bit
f_a	Memory access rate (including instruction and data accesses)
k_{cycles}	Number of instruction fetches in a given benchmark
k_{misses}	Number of cache misses incurred for a given benchmark
$k_{replacements}$	Number of tag replacements incurred for a given benchmark
k_{writes}	Number of cache writes incurred for a given benchmark
ℓ	Number of lines in a cache
P_{leak}	Leakage power
P_{mux}	Memory bank multiplexer/demultiplexer power consumption
P_n	Average power consumption for an n-bank memory
r_{hit}	Cache hit rate
r_ℓ	The average number of sequential instruction fetches targeting the same cache line, before a different cache line is accessed
w	Cache line width, in words

cropower, single-chip processors. We focus on instruction caches because instruction fetches occur on every cycle, demonstrate a great deal of locality, and are read-only. In evaluating our designs, we consider principally the impact on net energy per cycle.

We find that a properly designed cache can reduce instruction fetch energy by up to an additional $50-90\,\%$, depending on the processor workload. We also find that the optimal cache size for this design space is surprisingly small: around 64 to 128 words. Because of the small capacity, we prefer a flip-flop, rather than SRAM, implementation, for reduced dynamic power (and possible lower voltage operation).

Because of the small cache capacity, preventing thrashing of the cache in the common case where the working set exceeds the cache capacity is critical. We describe a mechanism for preventing thrashing in small, direct-mapped caches. Along the way we also explore the theoretical performance bounds of small caches, including with optimal replacement policies and full associativity.

4.2.1 Related Work

Many ideas have been published for reducing memory system energy (though few have considered the micropower domain). We limit our discussion here to instruction caches.

In tagged cache architectures, such as direct-mapped or associative caches, the mapping between locations in the cache contents and locations in the backing memory is based on address. Each cache line is "tagged" with the common MSBs of the addresses that are mapped to that line. Storing these tags requires additional memory in the cache, and significant logic is required to compare the tags against instruction addresses from the CPU to determine if a hit occurs. Because each cache line is mapped independently, tag caches have the flexibility to cache multiple, non-sequential memory regions simultaneously (such as a subroutine and a loop which calls it).

Filter caches [36] are small, fast, tagged, L0 caches designed to reduce memory system power by reducing the access rate seen by a larger, higher access energy L1 cache. Because the L1 cache is almost as fast as the L0 filter cache, L0 misses do not increase memory latency significantly, while each L0 hit may save significant energy. However if the L0 hit rate is too low, then both performance and power may suffer. The µAMPS instruction cache is in essence a filter cache, with fast on-chip memory substituting for the L1 cache.

Loop caches [37,38] are non-tagged caches that a employ control-flow based mapping, instead of address-based mapping. The M·CORE loop cache [39,40], for example, monitors the instruction stream for backward branch instructions that indicate looping. When a loop is detected, the sequence of instructions that make up the loop is recorded in the cache, where it can then be played back for successive iterations of the loop. The loop cache is essentially a single-line cache and can can only be allocated to one loop at a time. Unlike normal tagged caches, the loop cache is not thrashed by a loop that is larger than the cache. In the event that a loop exceeds the size of the cache, the beginning of the loop (up to the capacity of the cache) is cached and the remaining loop instructions are fetched from main memory as normal. Loop caches are only effective at caching loops, and do not benefit other sources of repetitively executed code, such as subroutine calls and exception handlers.

Software caches [41] are another form of non-tagged cache, where frequently accessed instructions are mapped by the compiler to a special, small memory bank with low access energy. Multiple subroutines can share the cache, being copied in and out as needed, using DMA transfers (though this can result in stalling the software for an extended time while the cache contents are updated). Since the cache is allocated at compile time, software caching does not work well with dynamically loaded code. Implementing the hardware portion of a software cache is trivial, but the compiler support required is complex. The 4 kB memory banks of the µAMPS DSP could be considered software caches, since they have lower access energy than the other 8 kB banks. (We have not, however, developed the compiler support to automatically allocate code or data to these banks.)

A hybrid of tagged and software caches is the use of compiler-inserted hint instructions for improving cache replacement decisions. Examples include the keep/kill instructions in [42], as well as the PowerPC, UltraSPARC, and Cyrix architectures.

4.2.2 Benchmarks

A suite of five workloads was used to benchmark the cache designs considered in this work.[3] This a is a very small suite, and the workloads are all very different: our intent is not to attempt to characterize the "average" microcontroller workload, but rather to explore a range of different workloads. The benchmarks are as follows:

- crc16 computes a cyclic redundancy check on 160 words of random data. The algorithm is implemented as a short subroutine called within the main loop for each word of data. The code size is small and, correspondingly, the algorithm achieves high hit rates even with very small caches.

- swfft computes a 128-point real-valued Fourier transform. (The transform is computed in software, not using the DSP's hardware FFT accelerator core.) The algorithm primarily consists of several moderate-sized nested loops.

- tea encrypts and decrypts 128 bytes of data using the Tiny Encryption Algorithm [43]. The encryption and decryption algorithms are both relatively large, simple (non-nested) loops.

- irq consists entirely of interrupt handlers servicing an RS-232 UART, multiple DMA channels, and a timer system generating events at three different frequencies from a single hardware timer. Unlike the previous computational workloads comprised of long-running loops, the event-driven irq trace consists of many short blocks of code mixed together, with much less locality.

- swfft+irq combines the swfft workload running in the foreground with the irq interrupt handlers running in the background. The result is a mix of long running loops frequently interrupted by short interrupt handlers

The benchmarks are all written in .C and were compiled with a custom port of the GCC-4.0.3 compiler for our DSP. Table 4.2 contains basic statistics for each of the workloads. The benchmarks algorithms are all relatively small: hundreds of bytes

[3]As in Chapter 3, we are forced to create our own benchmark suite, since there are no established ones.

of code and tens of thousands of cycles. A real system would combine multiple of these size building block algorithms to form a full-scale application. Generally, the cache footprint of the benchmarks exceeds the capacity of the caches we will be considering. We therefore expect that a full-fledged application made up of multiple of these algorithms could reasonably be analyzed by considering each algorithm in turn. The algorithms will not interact much in the cache because each algorithm will occupy the entire cache when it is running.

4.2.3 Direct Mapped Sector Caches

We begin with an analysis of a simple direct-mapped cache, which will serve as a baseline for further optimization.

As shown in Figure 4-2, the cache consists of three memory arrays: a data store, used to hold the cached instruction words; a valid-flag store, containing one bit for each word in the data store indicating whether or not the respective location in the data store contains valid data; and a tag store, used to indicate which addresses are mapped to each data store location. Note that the data and valid-flag stores do not depend on the line width, the number of lines, or the replacement policy: all of that is determined by the implementation of the tag store (and associated tag-replacement

Table 4.2: Benchmark statistics

	crc16	swfft	tea	irq	swfft+irq
Runtime (cycles)	14 338	25 452	97 805	16 482	59 512
Code size (bytes)	104	806	620	880	1456
Mean block length[a]	6.46	42.2	89.6	9.87	19.0
Ideal hit rate[b]	0.996	0.984	0.997	0.978	0.989
Address toggle rate[c]	2.18	2.04	2.03	2.11	2.12
Instruction toggle rate[d]	6.62	6.62	6.14	6.25	6.60
Load operations[e]	0.02	0.23	0.28	0.23	0.30
Store operations[e]	0.01	0.11	0.05	0.15	0.15

[a]Average number of sequential instructions between changes in flow
[b]Hit rate with an arbitrarily large cache (cold-start misses only)
[c]Average number of address lines toggling per cycle
[d]Average number of instruction bits toggling per cycle
[e]As a fraction of the total instruction count

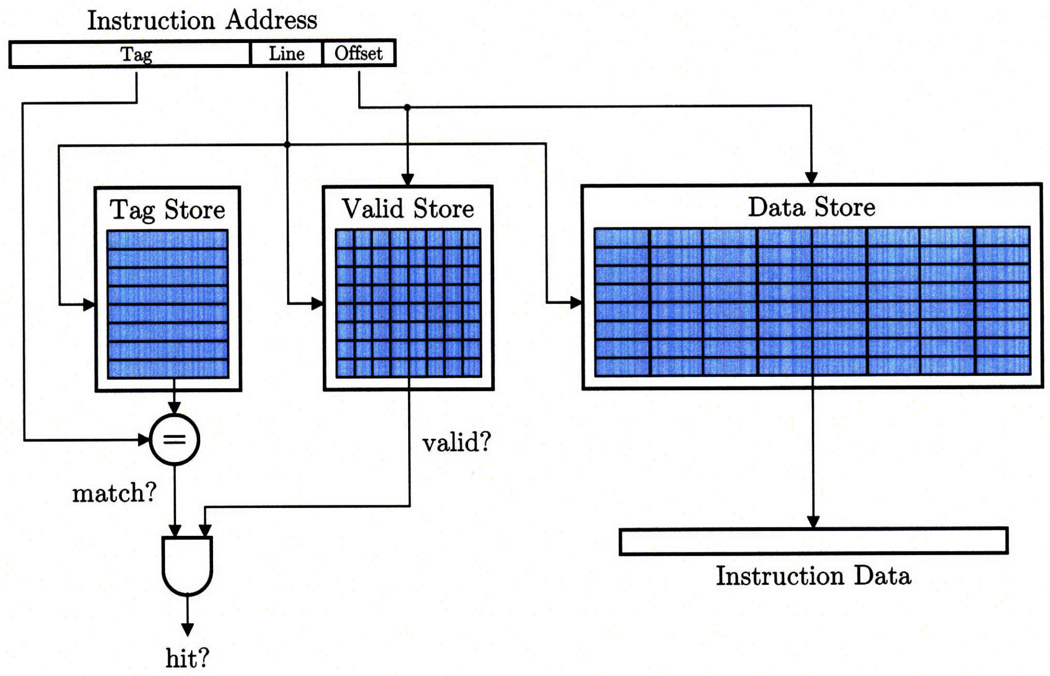


Figure 4-2: Structure of a direct-mapped cache

logic, which is not shown in the figure).

The data, tag, and valid stores would normally be implemented as SRAMs. However, in this work, we elect to use flip-flop based memories (synthesized using standard cells) instead. At the array sizes we are considering for caches (a few kilobits), SRAM and flip-flops provide about the same access energy.[4] Also, flip-flops will operate at lower voltages than the standard SRAM macros, allowing us to run the cache at the CPU voltage and further reduce the cache access energy.

In order to maintain single-cycle memory latency, even during cache misses, the tag and valid stores are read asynchronously. This allows the hit/miss decision to be made as soon as the instruction address from the CPU is valid, and in time to enable the main memory on the same clock cycle if a miss is detected. Since instruction fetch is part of the critical path on the DSP, this implementation results in a reduction in the maximum clock frequency. However, as we will show, the extra delay is a small percentage of the critical path.

[4]SRAM still has the decided advantage in leakage and area, however, neither of these is as important as dynamic power, in our system. As shown in Figure 1-6, the leakage energy per cycle is much less than the dynamic energy.

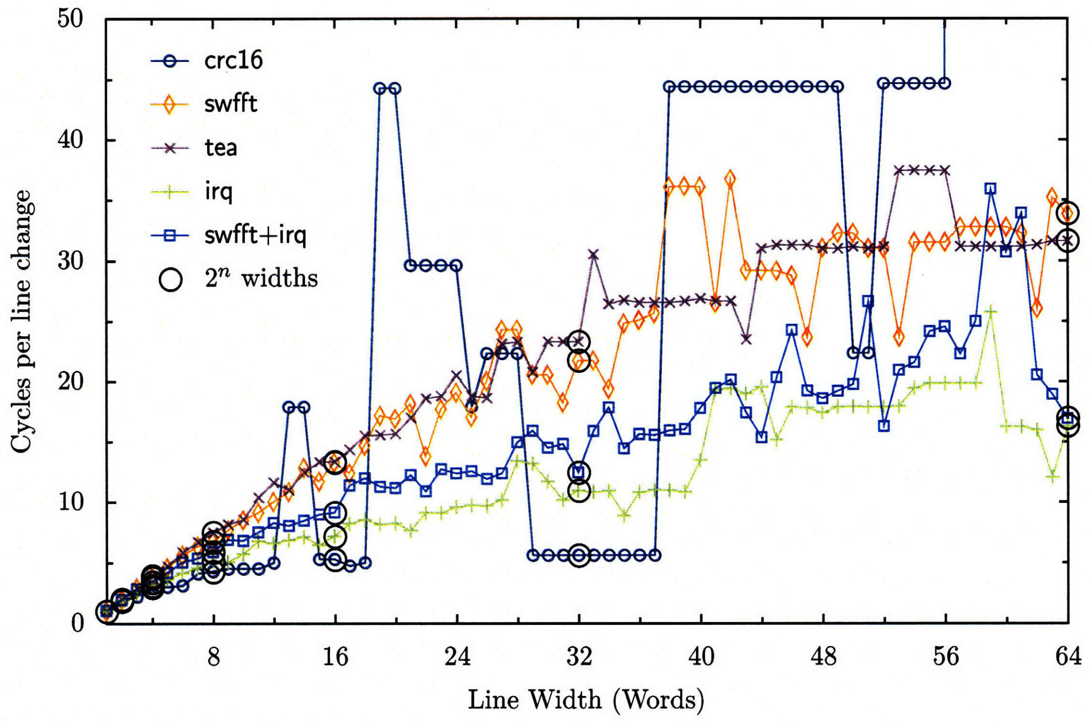


Figure 4-3: The rate at which a workload changes from line to line of a direct-mapped cache depends on the width of the line, but the relationship is not simple (or monotonic). Data is plotted for arbitrary line widths, but in practice, only the power-of-two (circled points) are efficiently implementable.

Implementing the tag store in static logic is particularly beneficial in reducing the tag comparison energy. Since the tag is the most-significant portion of the instruction address, the tag bits change relatively infrequently, resulting in low switching activity in the tag comparison logic. Furthermore, the tag store read logic only consumes dynamic power when the line number field of the current address changes. Figure 4-3 illustrates how the number of cycles between line changes varies with line widths from 1 to 64 words. While in practice, efficient hardware implementation requires that the line width be a power of two, in the figure we plot the line change frequency for arbitrary line widths to illustrate that the frequency is a rather chaotic function of the line width: small changes in line width have large, unpredictable effects on the line change rate. We infer that, correspondingly, small changes in the program code (e.g., loop length) would have a similar, unpredictable impact. Therefore, we do not attempt to construct a model for the line change frequency, but simply note that by only reading from the tag store when the tag field of the current address changes, we can expect to realize a several-fold reduction in tag store power. We will see that as a result, the tag logic consumes a relatively insignificant portion of the cache access energy—for all but very short line widths.

Because we are concerned with energy and not latency, we utilize a sector cache design, which reduces memory traffic. In a sector cache, each cache line is divided into multiple subblocks or sectors which can be loaded independently. When a miss occurs, instead of fetching an entire cache line, only the requested subblock is loaded into the cache. A subblock that is never accessed is never loaded from memory, thereby resulting in reduced memory traffic compared to a non sector cache. (See [44] for a general analysis of sector caches.)

To minimize memory traffic, we utilize a subblock size of one word, so each miss results in only fetching one word from main memory. No speculative fetching occurs (words are fetched only when they are immediately needed), and thus no advantage is made of spatial locality within the instruction stream. To support multiple subblocks per cache line, every subblock must have its own valid flag. This overhead is relatively small, however, as the valid flag store is still much smaller than the data store (one bit per location for the flags, compared to 16 bits per location to hold the actual instruction data).

An alternative to a sector cache would be to implement a multi-word wide main memory, so that entire cache lines can be fetched from memory in one read operation. The total number of words fetched from memory will be greater with this implementation than with a sector cache, because some of the lines fetched will contain words that are never actually executed. The multi-word memory implementation is advantageous, from an energy standpoint, only if the read energy per word is sufficiently less than for a single word wide memory to compensate for the memory traffic reduction of a sector cache. The relative reduction in memory traffic for a sector cache can be calculated as

$$\frac{(\textit{normal cache miss rate} \times \textit{line width}) - \textit{sector cache miss rate}}{\textit{normal cache miss rate} \times \textit{line width}}$$

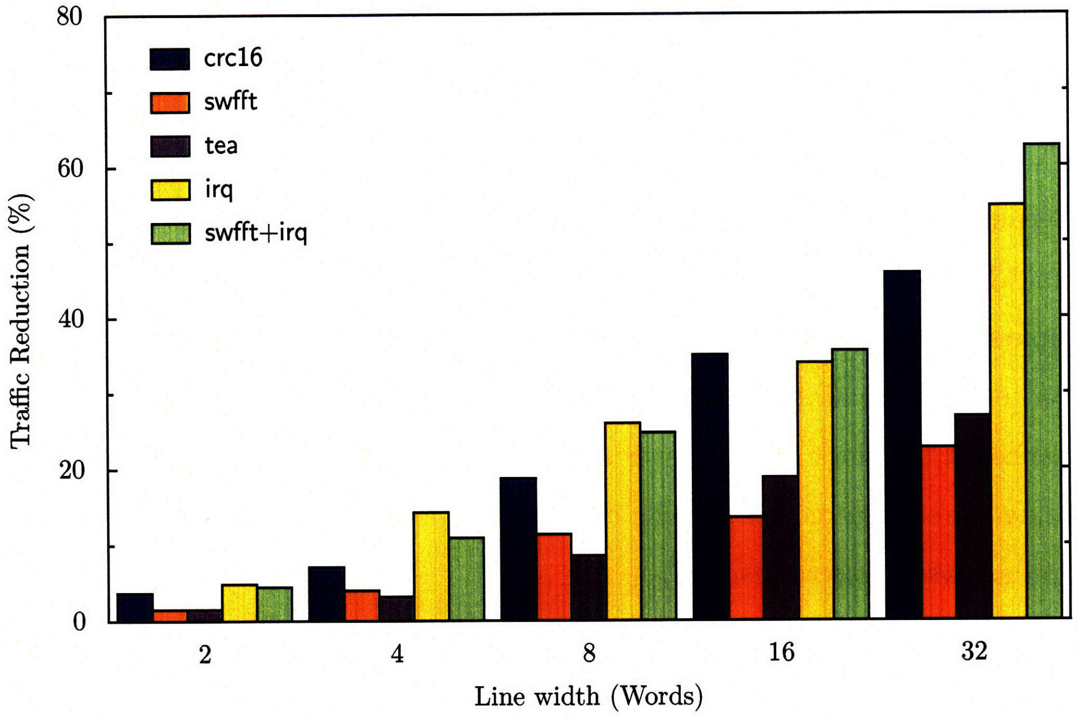


Figure 4-4: Memory traffic reduction ratio resulting from sector caching. (Data was obtained from simulating 64-word, direct-mapped caches with varying line widths.)

which is plotted in Figure 4-4 for line widths from 2 to 64 words and for each workload in our benchmark suite. For a line width of 4 (which we find to be roughly optimal), a sector cache is provides a memory traffic reduction of $5 - 15\,\%$.

Instruction traces taken from the benchmarks described previously were analyzed to determine the hit rate achieved by cache sizes n ranging from 16 to 256 words, as shown in Figure 4-5. Each cache size was simulated with varying line widths, ranging from one word (n lines per cache) up to half the total cache size (two lines per cache[5]). These simulations reveal that the hit rate depends predominantly on the total size of the cache, and only secondarily on the organization (number of lines) of the cache. The hit rate does increase somewhat for a fixed cache size when the number of lines in the cache is increased. The effect is most prominent in caches that are almost large enough to contain the entire working set (the hit rate is neither extremely high nor extremely low).

The direct-mapped cache was implemented as a parameterized Verilog model, and

[5]Due to the way the Verilog model was written, the number of lines must be at least two.

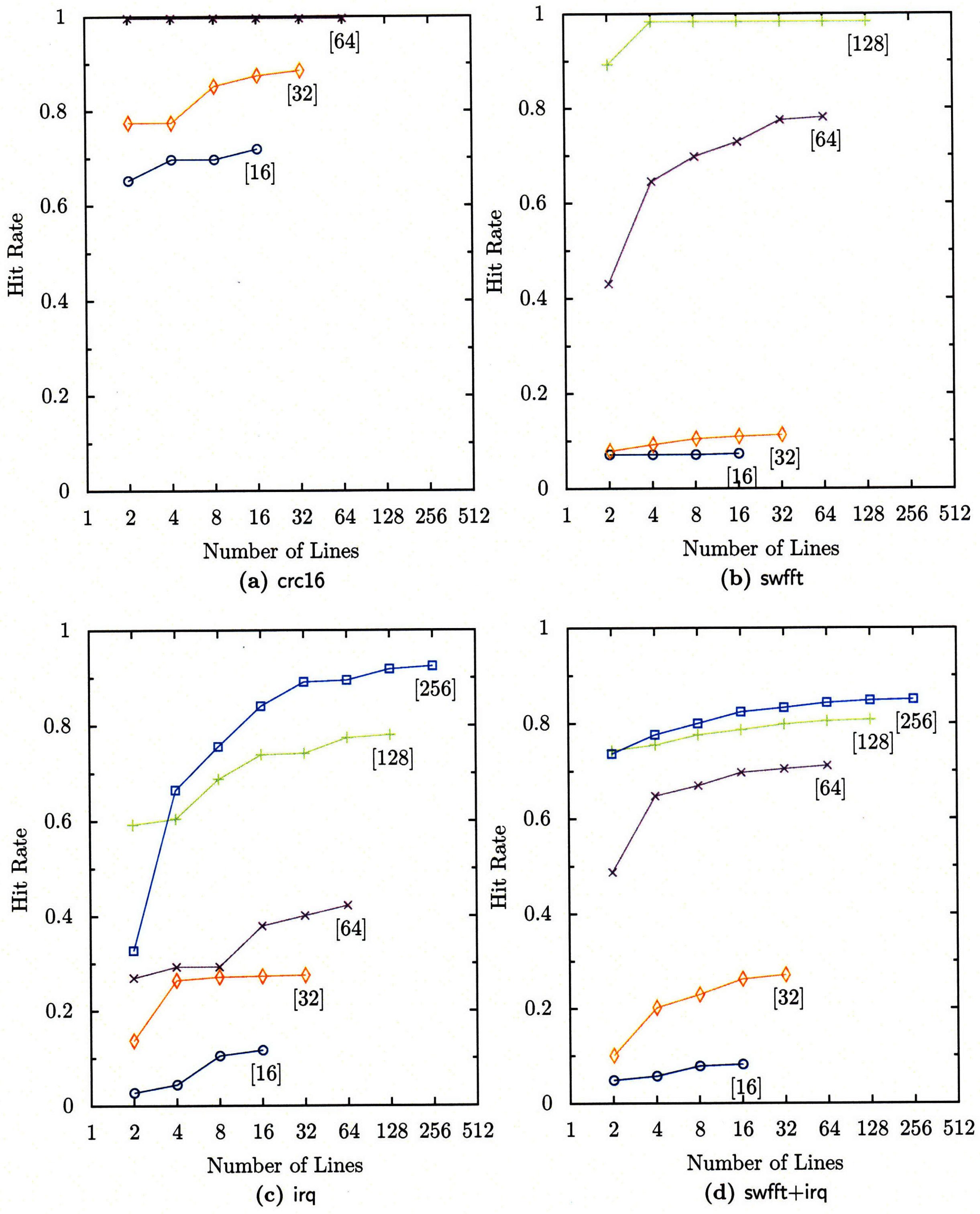


Figure 4-5: Hit rates for various cache configurations for four of the benchmarks: crc16, swfft, irq, and swfft+irq (a–d). Each line indicates a different total cache size as the number of lines (and correspondingly, the line width) is varied. The numbers in brackets indicate total cache size, in words.

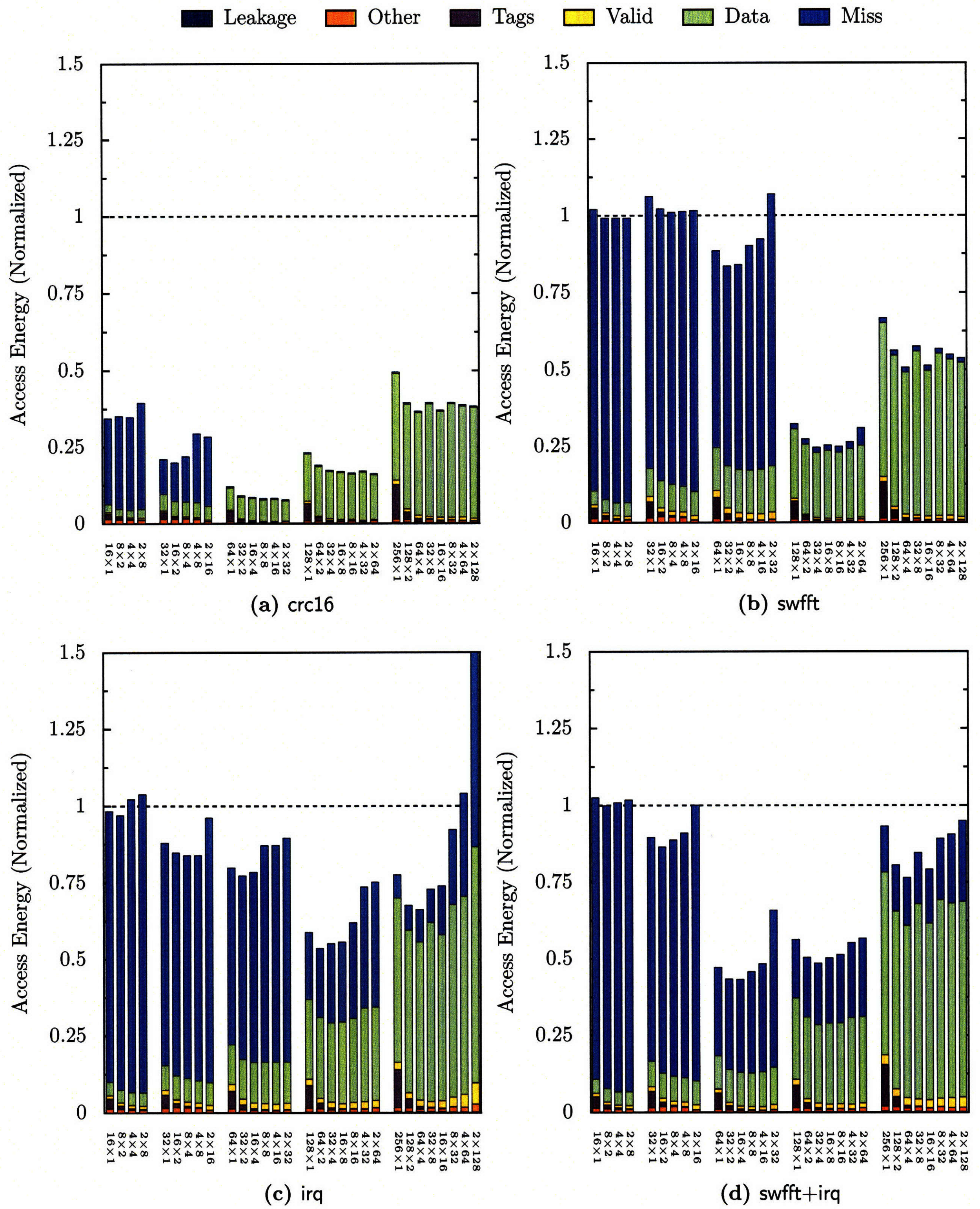


Figure 4-6: Net instruction fetch energy for the crc16, swfft, irq, and swfft+irq (a–d) benchmarks, for various cache configurations. Energy is normalized relative to a main memory access, so bars shorter than 1.0 represent a net energy savings. Cache configurations are denoted as $\ell \times w$, where ℓ is the number of lines and w is the line width in words. Configurations are grouped by total cache capacity: 16, 32, 64, 128, or 256 words.

synthesized for each of the configurations described in Figure 4-5. Power consumption was calculated using Synopsys PowerCompiler, using switching activity annotated from gate-level simulation of each cache configuration and interconnect parasitics estimated from physical prototyping with PhysicalCompiler. Finally, the effectiveness of each configuration at reducing instruction fetch energy was evaluated using the model

$$E_{fetch} = (1 - r_{hit})E_{mem} + E_{cache} \tag{4.1}$$

where r_{hit} is the hit rate, E_{mem} is the access energy for main memory (a single 64 kB SRAM), and E_{cache} is the cache access energy (including writes to the cache) computed from the dynamic and leakage powers reported by PowerCompiler for an access rate of 40 MHz.

Figure 4-6 illustrates the effective fetch energy for each cache configuration, normalized to E_{mem}, so values less than 1.0 represent net energy savings over having no cache. (Results from the tea benchmark are omitted, since they are very similar to the swfft results.) The upper bound on the size of the cache is 256 words: 512-word and larger caches achieve high hit rates, but provide no energy savings because $E_{cache} > E_{mem}$. A 128-word cache is the best all-around choice, at least for this benchmark suite.

In this low-power process technology, leakage accounts for only $1 - 4\,\%$ of the cache access energy. The valid store also represents a small portion of the access energy, $1 - 15\,\%$. Most of the access energy is consumed in the data store $(40 - 95\,\%)$. For configurations with a large number of very short lines, the tag store energy can be as much as $40\,\%$ of the access energy, but is typically $5\,\%$ or less for configurations with a line width of four words or more. Finally, it is worth noting that the cache power for each configuration varies depending on the hit rate. (This can be seen by comparing power for the same cache configuration on different benchmarks.) Lower hit rates result in higher cache power due to more frequent writes to the data, valid, and tag stores.

Synthesis of the Verilog cache model was done without any timing constraints, allowing the tools free rein to minimize power. The propagation delay of the cache (measured from when an address is received from the CPU to when the memory enable becomes valid) is $4 - 8\,\text{ns}$. Adding a timing constraint would likely reduce that significantly, with only a minor effect on energy. In the final DSP design (which includes an $16{\times}4$ cache), the cache accounts for $7\,\%$ of the critical path.

4.2.4 Energy Modeling and Analysis

Consider a cache with ℓ lines of w words per line. When a cache access occurs, the $\log_2 \ell w$ LSBs of the address are used to access the valid and data stores. The tag store is also accessed using the line-number field of the address, though as previously noted, these bits change less frequently than every access. If the tag matches and the accessed cache word is valid, then a hit has occurred and the data read from the data store are returned to the CPU as the requested instruction. The energy dissipated in this case is

$$E_{access} = E_{data}^{read} + E_{valid}^{read} + \frac{1}{r_\ell} E_{tag}^{read}$$

where r_ℓ is the line change rate illustrated in Figure 4-3.

In the event of a cache miss, it is usually advantageous to store the data returned by main memory into the cache so that future accesses to this address will result in hits. However, if the address is not reused quickly or frequently enough, it might be advantageous to not cache the address, thereby saving the energy required to write to the cache and avoiding evicting other addresses. Servicing a cache miss without modifying the state of the cache is known as bypassing.

Suppose a memory request results in a tag match, but the particular word (sub-block) is invalid. Adding the word to the cache requires writing to the data store and setting the corresponding flag bit in the valid store.

$$E_{write} = E_{data}^{write} + E_{valid}^{write}$$

The energy savings from each future access to this word that hits in the cache is the main memory access energy, E_{mem}. To achieve any net energy savings, we need to have at least k hits before this word is evicted from the cache, where

$$k = \frac{E_{write}}{E_{mem}} = \frac{E_{data}^{write} + E_{valid}^{write}}{E_{mem}}$$

If $E_{write} < E_{mem}$ (which is almost certain, since the data and valid stores together are much smaller than the main memory), then $k < 1$, and every instruction is worth writing to the cache if it will be used at least once more before eviction.

Suppose a memory access results in a tag mismatch. Caching the data then requires changing the selected line's tag, storing the new data in the data store, and setting the valid bit for that word while simultaneously clearing the valid bits for all the other words in that line. The cost of changing a tag, E_{tag}, is the energy required to change the tag field bits and clear the associated valid bits.

$$E_{tag} = E_{tag}^{write} + wE_{valid}^{write}$$

Measuring E_{access}, E_{write}, and E_{tag} is somewhat problematic, even for simulated designs. One approach is to simulate a large number of randomly generated traces and then perform a least-squares fit to the model

$$E_{cache} = k_{cycles}E_{access} + k_{writes}E_{write} + k_{replacements}E_{tag} \tag{4.2}$$

where k_{cycles}, k_{writes}, and $k_{replacements}$ are the number of cache accesses, words written to the cache, and tag replacements, respectively. In practice, however, this method does not work well because k_{writes} and $k_{replacements}$ are highly correlated ($\rho \approx 0.99$ for most cache configurations). One solution is to combine E_{tag} into E_{write} and use the simpler model

$$E_{cache} = k_{cycles}E_{access} + k_{writes}E_{write} \tag{4.3}$$

instead. Figure 4-7 illustrates the values of E_{access} and E_{write} obtained by fitting equation 4.3 to simulations of 100 randomly-generated 10,000-cycle instruction traces

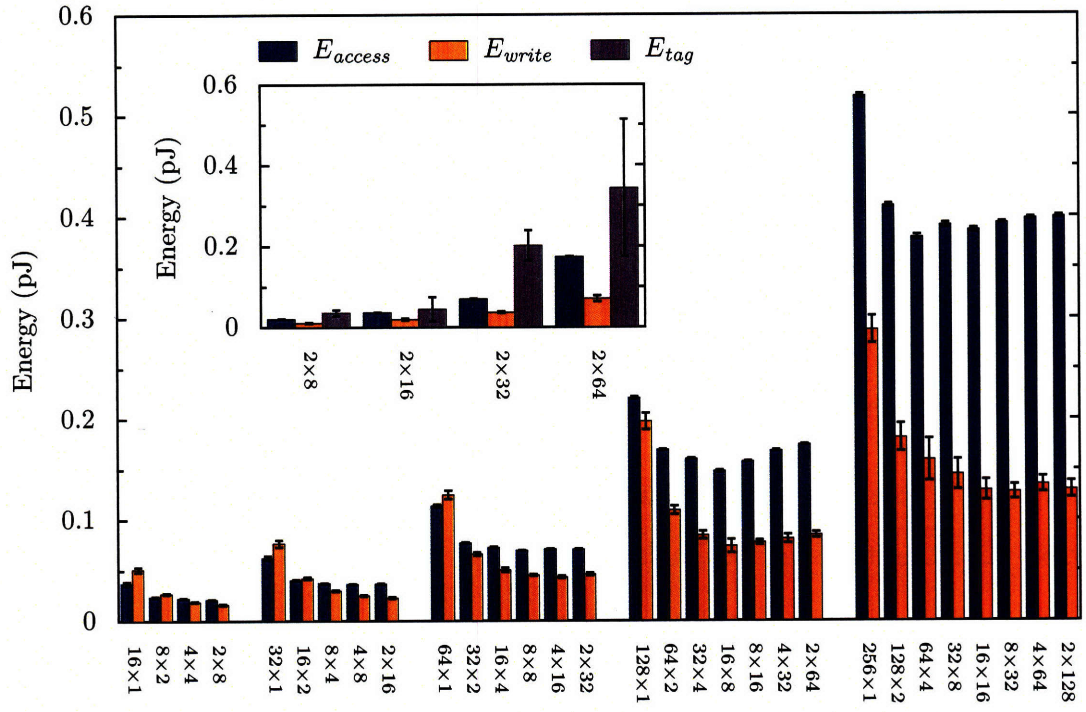


Figure 4-7: Fitted values for E_{access} and E_{write}. The inset graph shows E_{access}, E_{write}, and E_{tag} for the 2-line cache configurations, where E_{write} and E_{tag} were separable. Error bars indicate the 95 % confidence intervals. (The confidence intervals for E_{access} are nearly too small to see.)

for each cache configuration. (E_{access} and E_{write} are reported in units of E_{mem}.) E_{access} is roughly proportional to the cache capacity, but increases significantly for very short line widths, due to the increased cost and frequency of reading from the tag store. E_{write} follows a similar pattern of being relatively constant for moderate to large line widths and increasing for small line widths, where E_{tag} and the tag replacement rate become significant. (Small line widths result in more cache lines for a given cache capacity, and therefore, a larger tag store and correspondingly higher E_{tag}.)

For the purposes of designing an energy-optimal replacement algorithm we still wish to estimate E_{tag}. The degree of correlation between k_{writes} and $k_{replacements}$ decreases with increasing w. We were able to achieve a reasonable fit to equation 4.2 for those cache configurations having the minimal number of lines (two) for a given cache size (and thus the maximal line width for that size). Increasing w also increases E_{tag}, because w valid bits must be cleared each time a tag is changed. Thus these E_{tag} values represent worst-case bounds for all configurations with the same cache capacity. E_{access}, E_{write} and E_{tag} (normalized to E_{mem}) for these configurations are shown in the inset graph in Figure 4-7. E_{tag} is significantly larger than E_{write}.

Combining equations 4.1 and 4.3, the total instruction fetch energy is

$$E_{fetch} = k_{misses}E_{mem} + k_{cycles}E_{access} + k_{writes}E_{write} \tag{4.4}$$

The predictive accuracy of this model for four real benchmarks is illustrated in Figure 4-8. The relative error ranges from $-25\,\%$ to $14\,\%$. The error magnitude is generally smaller for smaller cache sizes.

4.2.5 Cache Management with Optimal Replacement

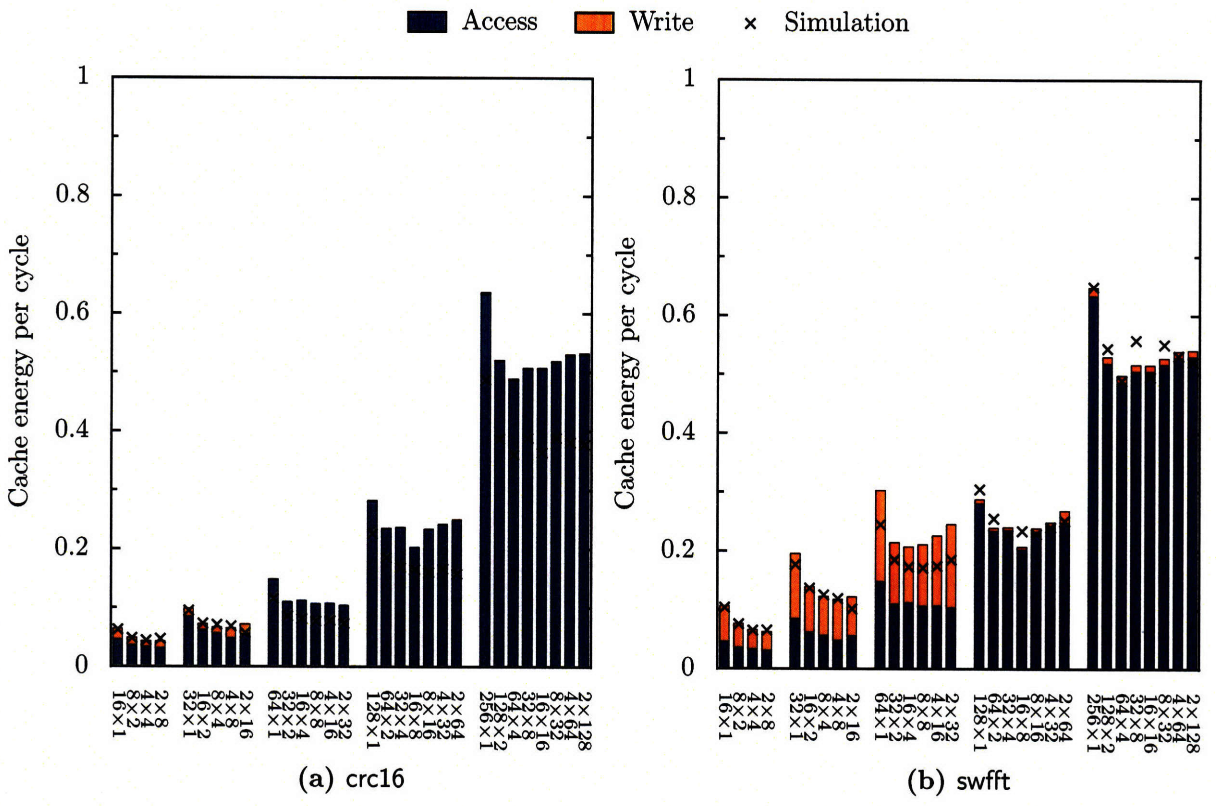
From Figure 4-6, the net energy savings obtained by instruction caching is limited by the hit rates that can be achieved with such small caches. Miss energy,

$$E_{miss} = (1 - r_{hit})E_{mem}$$

is the dominant energy drain for many small capacity configurations. If hit rate could be improved without increasing the cache size (by an improved cache management algorithm, for example), significant additional energy savings would result with these small caches.

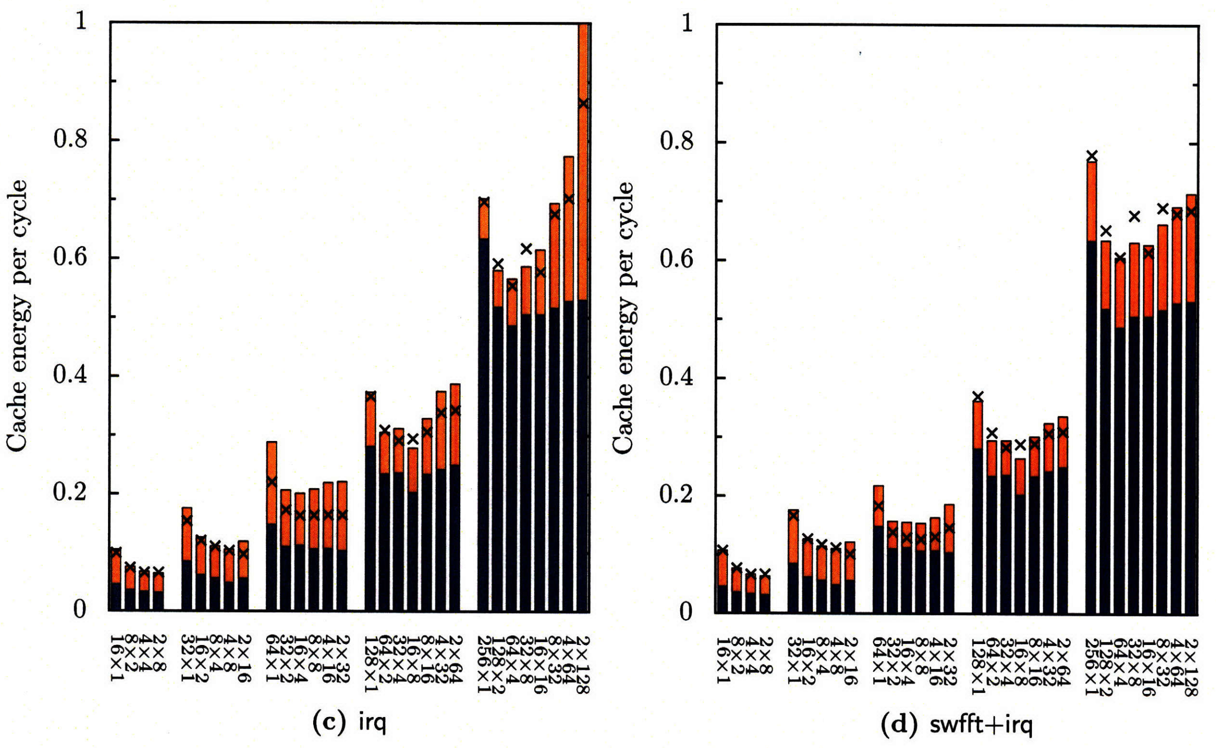
We now consider several off-line, optimal replacement algorithms. These algorithms are not implementable in hardware (because they make replacement decisions based on knowledge of all future accesses), but are nonetheless useful to study because they represent a theoretical upper-bound on the energy savings obtainable, and they can offer insight into the design of better, realizable replacement algorithms.

We begin by asking the question "What is the theoretical maximum energy savings obtainable with n words of cache?" For the moment, we will ignore the overhead of cache management entirely. Let us consider a fully-associative cache with a capacity of n words and a line width of one word. Any word in this cache can be independently allocated to any memory address at any time. If we can determine the optimal



(a) crc16

(b) swfft



(c) irq

(d) swfft+irq

Figure 4-8: Cache energy estimated with the model from equation 4.3 (bars) compared with actual simulated cache energy (crosses) for the crc16, swfft, irq, and swfft+irq benchmarks (a-d). Energy values have been normalized to E_{mem}.

set of allocations, this cache will therefore achieve the highest possible hit rate for an n-word cache. Belady's well-known MIN algorithm [45] is a provably optimal replacement algorithm for fully associative caches with mandatory replacement. MIN works as follows: when a miss occurs, look into the future in the instruction stream and determine the time of next use $t_{next}(i)$ for each element i in the cache. Replace the element i for which $t_{next}(i)$ is greatest. MIN is trivially extended to allow optional replacement (bypassing) as follows: in addition to calculating $t_{next}(i)$ for all cache elements, also calculate $t_{next}(a)$ for the address a causing the miss. If $t_{next}(a) > t_{next}(i)$ for all $i \in \{1 \ldots n\}$, do not evict any cache elements, and do not cache a. We will refer to this optional-replacement MIN extension as MIN-OPT.

Figure 4-9(a) illustrates the hit rates (r_{hit}) achieved on each of our benchmarks by MIN-OPT replacement with an ℓ-word capacity, one word per line, fully associative cache for $\ell = 1 \ldots 256$. Ignoring the overhead of cache management circuitry, we combine r_{hit} with a first-order read energy model for a flip-flop memory (where read energy is proportional to ℓ, the number of words in the memory: $E_{read}(\ell) = a_{read}\ell$) to obtain a lower bound on the fetch energy.

$$E_{fetch} = (1 - r_{hit}(\ell))E_{mem} + E_{read}(\ell)$$

$$\frac{E_{fetch}}{E_{mem}} = 1 - r_{hit}(\ell) + \frac{a_{read}}{E_{mem}}\ell$$

which is plotted in 4-9(b). Achieving any net energy savings ($\frac{E_{fetch}}{E_{mem}} < 1$) requires a hit rate of at least

$$r_{hit} > \frac{a_{read}}{E_{mem}}\ell$$

which is the dotted line plotted in Figure 4-9(a).

The minimum fetch energy occurs when $\frac{d}{d\ell}E_{fetch} = 0$, which in turn implies

$$\frac{d}{d\ell}r_{hit} = \frac{a_{read}}{E_{mem}}$$

The derivative $\frac{d}{d\ell}r_{hit}$ is plotted in Figure 4-9(c). The dashed line indicates the threshold a_{read}/E_{mem}. The circled points throughout Figure 4-9 indicate the minimal fetch

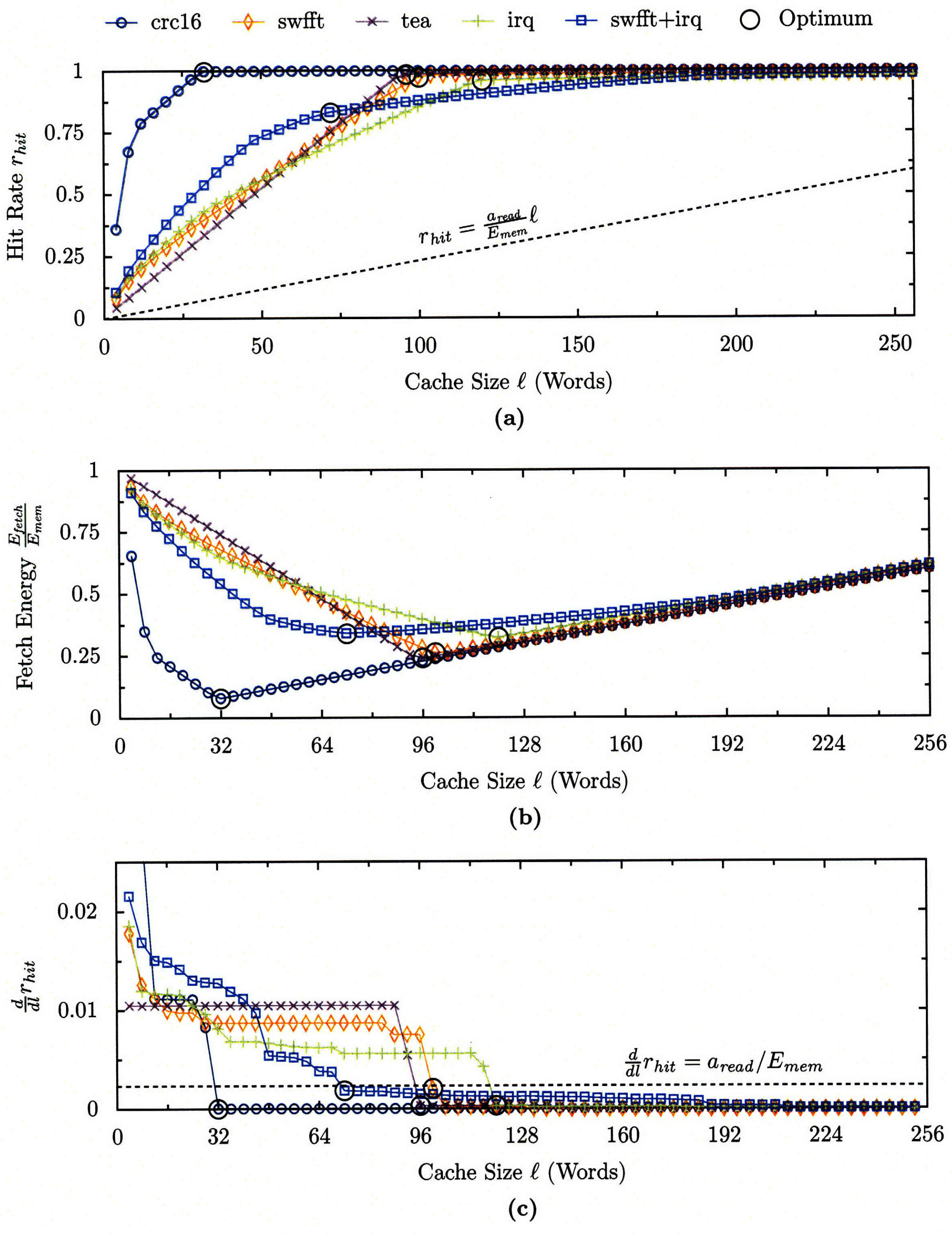


Figure 4-9: (a) Hit rate and (b) net instruction fetch energy (normalized to E_{mem}) for an idealized fully-associative cache model. Circled points indicate the optimum cache size for each benchmark. The dotted line in (a) indicates the minimum hit rate necessary to achieve any net energy reduction. The optimal cache size occurs when the derivative of the hit rate reaches a_{read}/E_{mem}, as shown in (c).

energy points for each benchmark. Note that for our E_{mem} and a_{read} values, the optimal hit rate for most of the benchmarks is essentially 1. In other words, the optimal cache size is just large enough to fit the entire benchmark. (There is no advantage to implementing a small cache with a lower hit rater but also lower access energy.) The optimal hit rate decreases as E_{mem} decreases or a_{read} increases—that is, as the difference between the cache and memory access energies shrinks.

For the loop-dominated benchmarks (**crc16**, **swfft**, **tea**), $\frac{d}{d\ell}r_{hit}$ has a distinctly stair-step shape, with each step corresponding to a different sized loop in the benchmark. If a loop is larger than the cache size, then incrementally increasing the cache size will increase the hit rate proportional to the number of iterations of that loop. Once the cache becomes larger than a loop, that loop will be completely cached and will not contribute to $\frac{d}{d\ell}r_{hit}$. The interrupt-intensive benchmarks have somewhat smoother derivatives.

Next we consider optimal management caches with multiple words per line. The MIN algorithm is not applicable to sector caches, because it does not account for partially valid cache lines. In essence, with a sector cache, the cost of changing a cache line is not constant, but depends on the current number of valid words in that line. Consider, for example, a 2-line, 2-word-per-line associative cache. Suppose we flush the cache, and then fetch the following addresses: 1, 2, 3, 5, 5, 3, 1, 2. As illustrated in Figure 4-10, Belady's algorithm will opt to replace the line containing $\{1,2\}$ in order to cache 5; however, replacing the line containing $\{3\}$ results in an additional hit. The non-optimality of MIN in general with variable replacement cost is shown in [46].

We do not currently know of an efficient algorithm for determining the optimum replacement decisions for a sector cache. Hosseini [46] proved that the optimal replacement problem is NP-complete when both cache element sizes and replacement costs are nonuniform, and conjectures that it is also NP-complete when either element size or replacement costs are nonuniform. Brehob et al. [47] compare the replacement decision problem with interval scheduling and imply that an efficient algorithm exists, though we did not find it in the reference they cite.

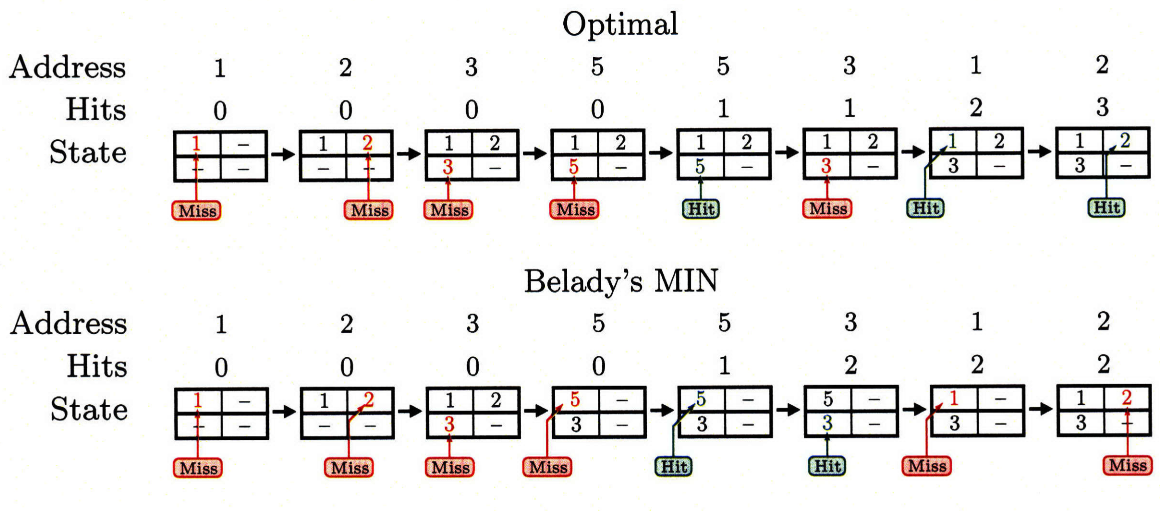


Figure 4-10: Belady's algorithm is not valid for sector caches.

For small enough benchmarks, the optimal replacement problem can of course be solved by an exhaustive search of the decision tree. Hosseini [46] describes such an approach for a similar problem (web caching), using best-first search and dynamic programming (saving solutions to subproblems that may be needed again later in the search). When we applied their method to the sector caching problem, we found the memory requirements were too great. The extra valid bits in a sector cache greatly increase the number of possible cache states, and as a result paths in the search space are less likely to converge, reducing the benefit of dynamic programming. A depth-first search was used instead, which requires less memory because it only explores one branch of the search space at a time.

By restricting ourselves to direct-mapped sector caches, we were able to compute the optimal replacement decisions for each of our benchmarks. Direct mapping shrinks the search space in two ways. First, the branching factor is only two (each miss can either be bypassed or cached in one specific line of the cache). Second, there is no interaction between replacement decisions for different cache lines. The lines (and the addresses mapping to them) can therefore be analyzed independently, reducing the depth of the search tree.

In order to determine the optimal cache replacement decisions to minimize energy,

Table 4.3: Comparison of optimization for maximum hit rate versus minimum energy for a 2×16 cache.

	crc16	swfft	tea	irq	swfft+irq
Optimized for hit rate					
Hits	13014	9893	32706	5165	30116
Cache writes	31	127	68	1164	3829
Tag replacements	3	9	5	82	254
Optimized for energy					
Hits	13014	9893	32706	5163	30116
Cache writes	31	127	68	1035	3813
Tag replacements	3	9	5	77	253

and not just maximize hit rate, we use

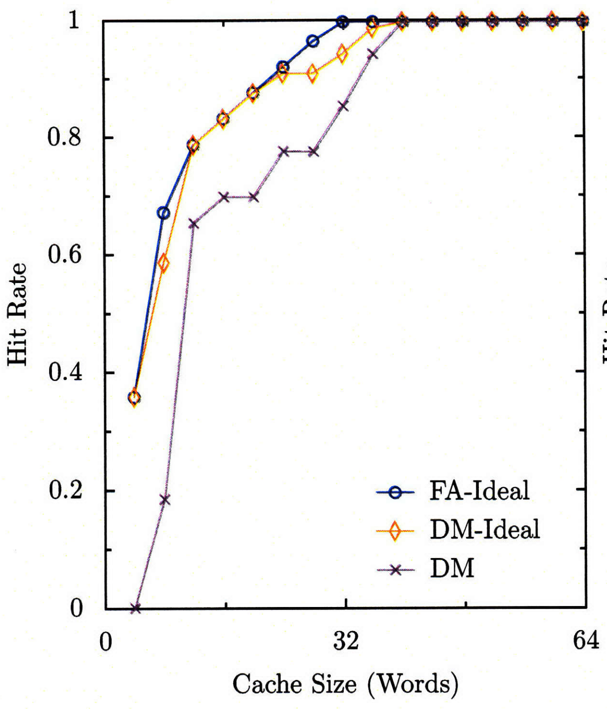
$$E_{fetch} = k_{misses}E_{mem} + k_{cycles}E_{access} + k_{writes}E_{write} + k_{replacements}E_{tag}$$

as a cost function when evaluating replacement decisions. However, because E_{write} and E_{tag} are less than E_{mem}, in practice there is little difference between optimizing for energy versus hit rate. Table 4.3 compares hit rate, word write, and tag replacement statistics with and without accounting for E_{write} and E_{tag} for a 2×16 cache. Only the interrupt-intensive benchmarks (where the tag replacement rate is much higher) show any difference at all, and even there, the differences are trivial.

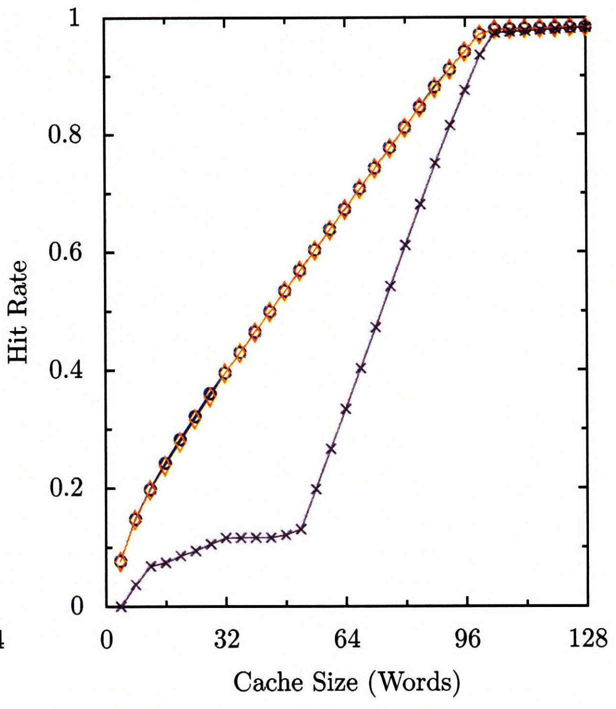
Figure 4-11 compares the hit rates achieved by a fully-associative 1-word-per-line cache and a direct-mapped, 4-words-per-line cache, both with optimal, off-line management.[6] A direct-mapped cache with mandatory replacement is also shown, for comparison. For small cache sizes (less than ≈128 words), there is relatively little difference between the fully-associative and direct-mapped caches, when both are managed optimally. With such small cache sizes, capacity misses dominate over conflict misses. This justifies our decision not to explore hardware implementations of associative caches.

As cache size decreases, the number of replacement opportunities that are declined by the optimal algorithm increases dramatically, as shown in Figure 4-12. An

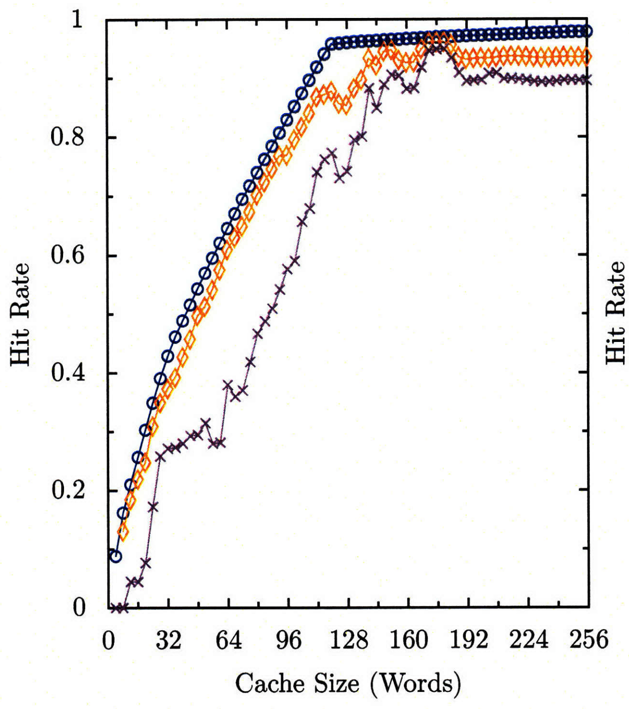
[6]A fairer comparison would have 4 words per line for the fully associative cache, but we are unable to determine the optimal replacement decisions for this configuration.



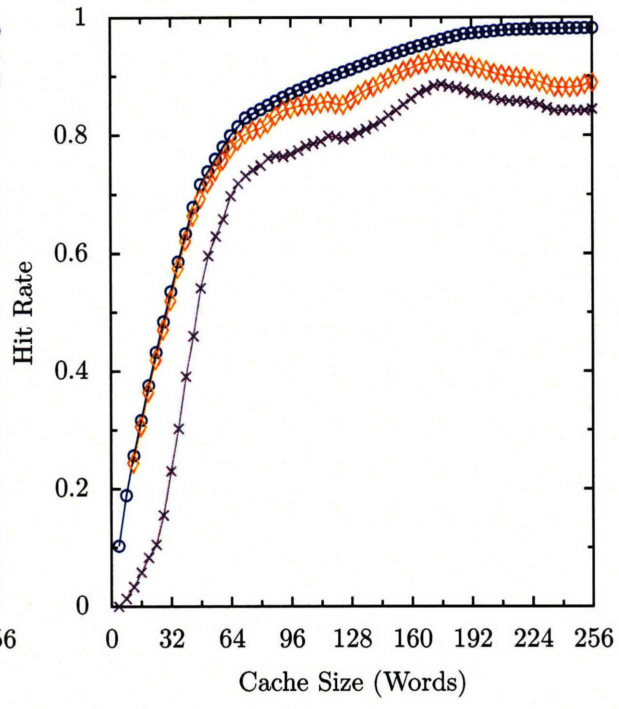
(a) crc16



(b) swfft



(c) irq



(d) swfft+irq

Figure 4-11: Comparison of replacement policies on four different benchmarks. The policies are fully-associative cache with MIN-OPT (FA-Ideal); direct-mapped with ideal replacement (DM-Ideal); and direct-mapped with mandatory replacement (DM). Note the different x scales for different benchmarks.

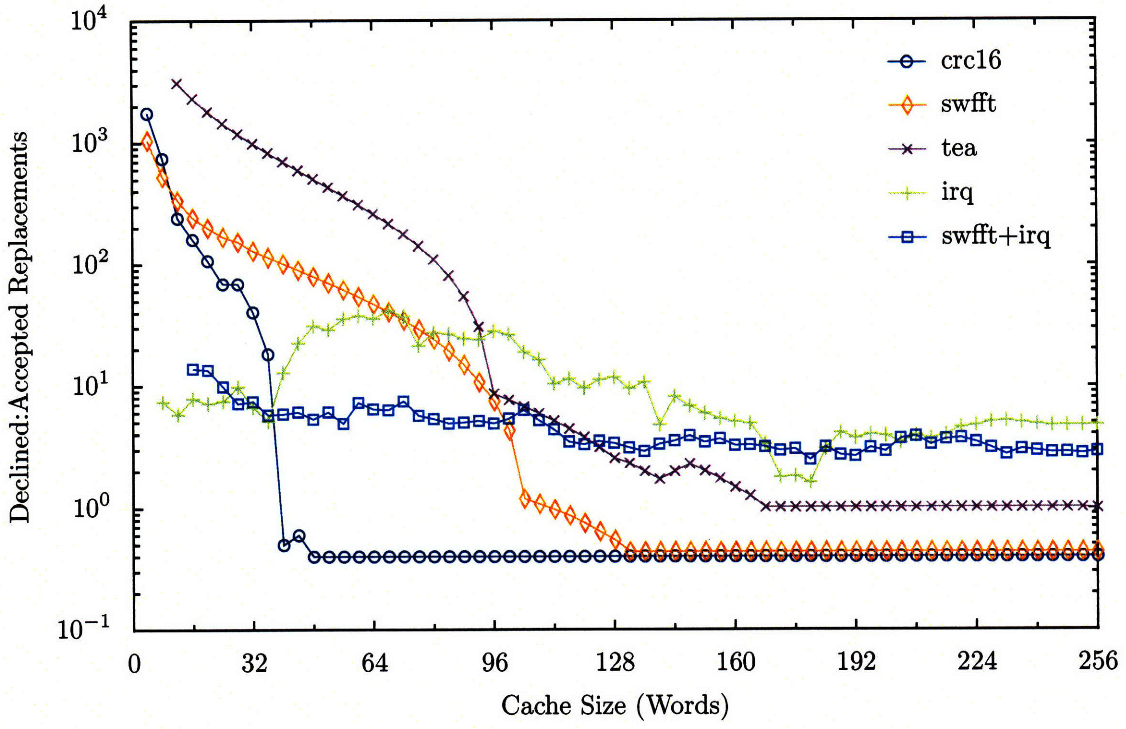
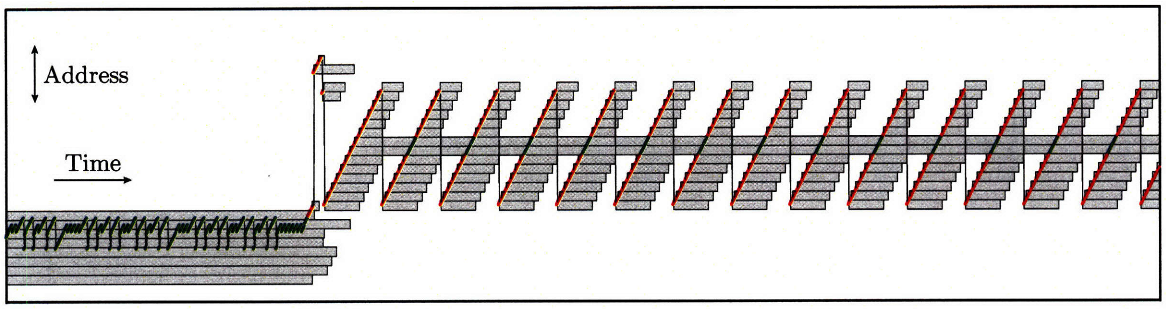


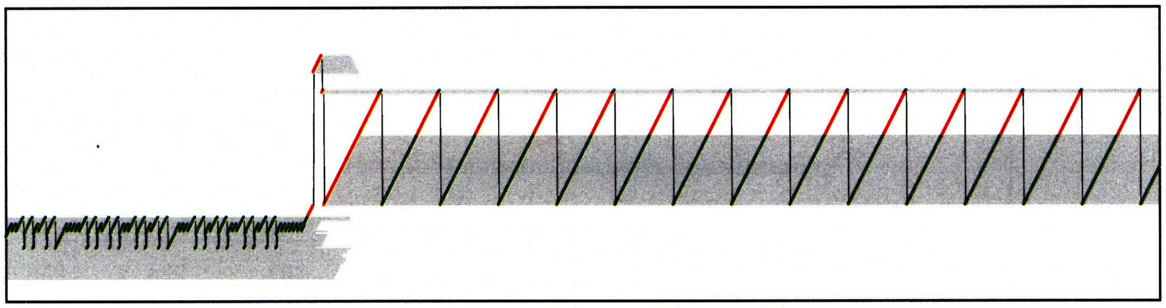
Figure 4-12: Ratio of declined to accepted line replacements for a direct-mapped cache (line width 4) with optimal replacement. (Note the logarithmic y-axis.)

optional-replacement policy is therefore doubly important for energy efficiency: it increases hit rate and also decreases energy spent changing the cache contents.

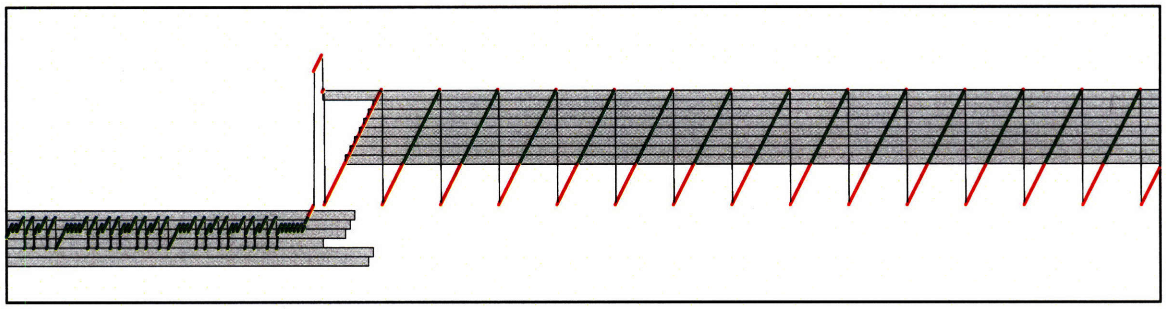
Figure 4-13 illustrates the allocation of cache lines over a small portion of the swfft benchmark, for multiple cache-management algorithms. In (a), a direct-mapped cache with mandatory replacement suffers from severe thrashing, because the inner loop of the benchmark is significantly larger than the cache. Only the cache lines that are not mapped to multiple locations in the loop contribute any hits. The optimal off-line algorithms (b, c) allocate the available cache capacity to a portion of the loop, caching as much of the loop as possible and ignoring the remainder of the loop once the cache is full. (The loop-protection algorithm in (d) is shown here for completeness: it will be described in the next section.)



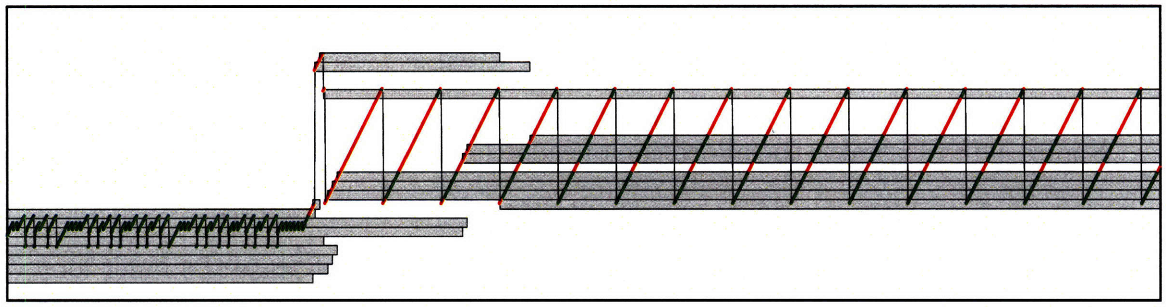
(a) 8×8, DM



(b) 64×1, FA-Ideal



(c) 8×8, DM-Ideal



(d) 8×8, DM-LP

Figure 4-13: Traces of a portion of the **swfft** benchmark using a 64-word cache, showing cache allocations and hits achieved by (a) a direct mapped cache with 8 words per line, (b) a fully associative cache with one word per line and optimal, optional replacement, (c) a direct mapped cache with 8 words per line and optimal, optional replacement, and (d) a direct mapped cache with 8 words per line with optional replacement controlled by a loop protection queue of length 4 (described in Section 4.2.6). Green dots represent hits, red dots represent misses, and the gray bars indicate the allocations of cache lines.

4.2.6 Realizable Allocation Algorithms

One approach to designing an implementable optional-replacement policy is to target thrashing. We propose here an entirely automatic mechanism, known as loop protection, which does not require modifying or otherwise annotating the instruction stream. Loop protection implements the following heuristic: lines that are part of a recently executed loop should not be evicted.

Loops are automatically detected by monitoring the instruction stream for taken backwards branch (TBB) instructions. (This idea was taken from the M·CORE loop cache [39].) When a TBB is detected, the upper bound U of the loop is indicated by the address of the branch instruction, and the lower bound L of the loop by the branch destination. In order to optimally handle nested loops (inner loop instructions should not be evicted by outer loop instructions), it is necessary to keep track of the last several sets of loop bounds. This is accomplished with a loop queue, which consists of two m-element FIFOs, into which the upper and lower loop bounds are written each time a TBB is detected. The queues only need to be wide enough to store the tag and line number portions of the bounds.

When a cache miss due to a tag mismatch occurs, the current tag t_i associated with the selected cache line is concatenated with the line number i, and compared with each pair of bounds L_j, U_j within the queue. If for any queue position $j \in \{1 \ldots m\}$, $L_j \leq t_i \| i \leq U_j$, then the cache line i is currently mapped to a portion of loop j, and should not be evicted.[7]

Consider a simple loop which is larger than the cache capacity. During the first iteration, instructions will be loaded into the cache as they are executed. Once the cache capacity is reached, later instructions in the loop will begin replacing earlier ones. At the end of the first iteration, the cache will be full of loop instructions, ranging from the end of the loop back as far as the cache capacity allows. The TBB at the end of the first iteration will load the bounds of the loop into the loop queue, protecting any loop instructions that are currently in the cache. During the second iteration, instructions at the beginning of the loop will result in misses, but will

[7]We use the symbol $\|$ to represent bitwise concatenation. Thus, $t_i \| i = \ell w t_i + i$.

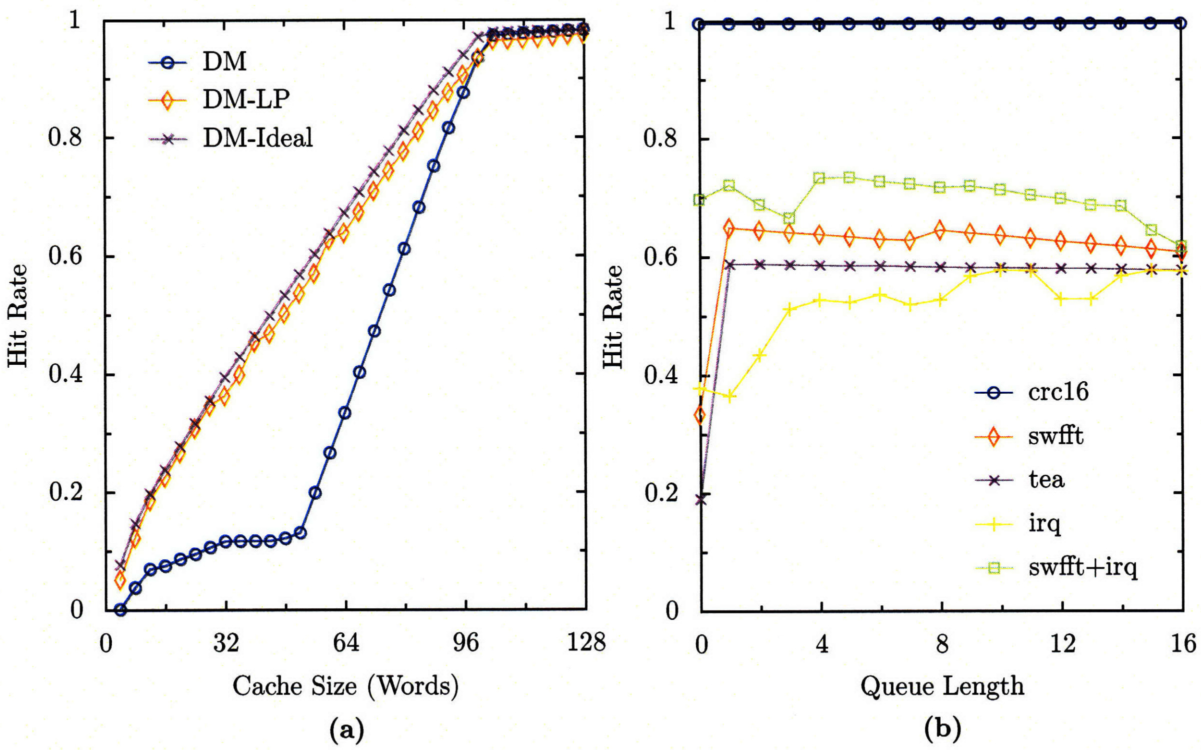


Figure 4-14: (a) Comparison of hit rate as a function of cache size for loop-protected (DM-LP), ideal (DM-Ideal) and mandatory (DM) replacement policies. The line width was fixed at 4 words, and the loop queue length was also 4. (b) Effect of loop queue length on hit rate for a 16×4 cache.

not be loaded into the cache because doing so would require evicting lines already holding instructions that are part of this loop. If the initial contents of the loop queue prevented some instructions from being cached during the first loop iteration, after a few iterations these old loop bounds will be pushed out of the queue, enabling all of the cache lines to be mapped to the current loop.

In the case of nested loops, after the inner loop completes, outer loop instructions can be cached as long as they do not evict inner loop instructions. While only loops formed with TBB instructions are automatically protected using this scheme, it is easy to implement a special hint instruction which manually places arbitrary entries in the loop queue. Hints can then be used to protect subroutines, exception handlers, and other repetitively-executed non-loop code.

Figure 4-14(a) compares the hit rates (as functions of cache size) achieved on the swfft workload by three different replacement strategies: mandatory replacement, loop protection, and optimal replacement with an off-line algorithm. For workloads composed primarily of loops, particularly loops large relative to the cache size, loop protection achieves hit rates almost equal to the theoretical maximum for a direct-mapped cache.

In Figure 4-14(b), we consider the effect of changing the loop queue length. A queue of length one is sufficient to achieve most of the benefit of loop protection, at least for large-loop-intensive workloads. Larger queue lengths benefit small-loop-, subroutine- and interrupt-intensive workloads like irq, where multiple code segments can fit in the cache at once.

The Verilog cache model from Section 4.2.3 was extended to include a loop queue. Based on Figure 4-14(a), we elected to implement a queue of length 4. The fetch energy simulations of Figure 4-6 were repeated for this improved cache, and the results are shown in Figure 4-15. The energy overhead of the loop queue is negligible, accounting for less than 1 % of the fetch energy on average and 5 % in the worst case. The improved hit rate with loop protection therefore translates directly to net energy savings. Up to a 58 % fetch energy reduction is achieved, with the typical reduction being about 10 % to 40 % across 16-, 32- and 64-word caches. For larger caches, where the miss rate without loop protection is near zero, loop protection results in a roughly 10 % increase in fetch energy, because the miss rate is slightly increased.

In conclusion, loop protection succeeds in increasing hit rate for loop-intensive workloads, and also decreases tag replacement frequency (reducing cache energy). The overhead of the added circuitry (the loop cache) is minimal.

4.2.7 Measured Cache Performance

The cache design actually implemented on the µAMPS DSP chip is a 64 word with a line width of four words and a simple direct-mapped replacement policy. (The DSP design was finalized before the loop protection policy had been developed.) The cache is part of the CPU power domain, so it is power gated with the CPU and operates at the same voltage as the CPU.

Figure 4-16 illustrates the measured impact of the instruction cache on total DSP

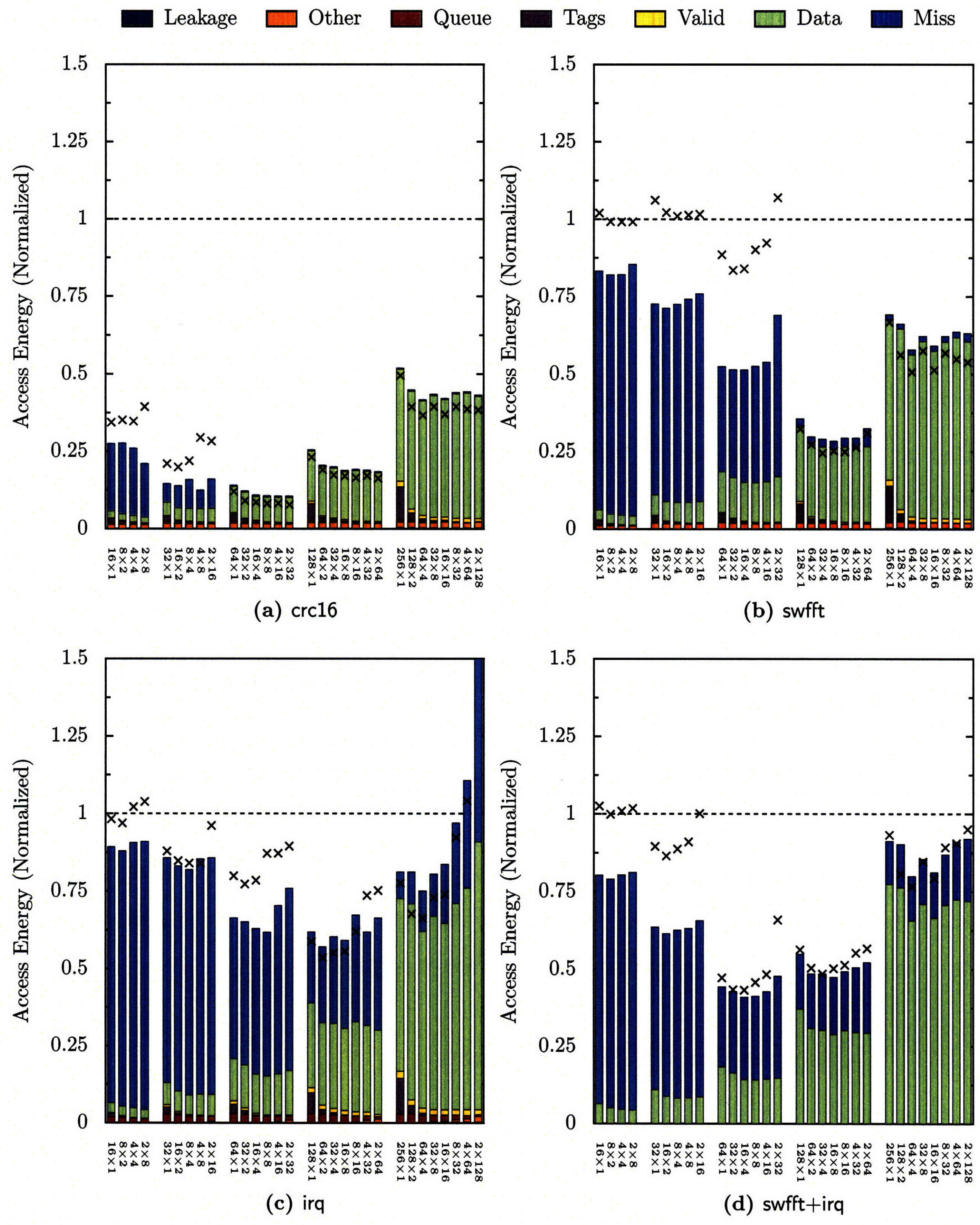


Figure 4-15: Simulated net instruction fetch energy for a direct mapped cache with a 4-element loop protection queue. For comparison, the total instruction fetch energies for the original direct mapped cache configurations are indicated with crosses.

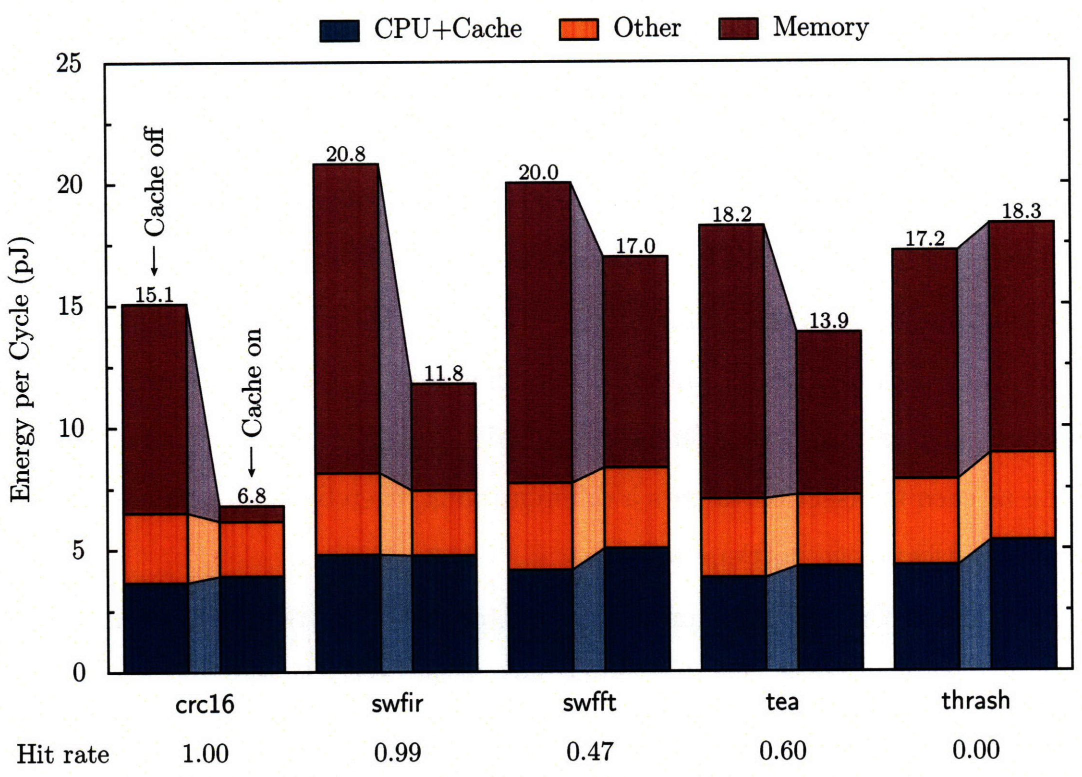


Figure 4-16: Measured impact of instruction caching on energy per cycle. Memory energy is reduced by 1.4 – 14 × with the cache enabled. The DSP was operating at 5 MHz with the logic at 0.5 V and the memory at 0.8 V.

energy per cycle for five different benchmarks. Enabling the cache reduces the average memory energy per cycle for the first four benchmarks by 3.7–8.4 pJ, a factor of 1.4–14 ×. Note that the memory energy measurements include data memory accesses, which are not cached. The variation in memory energy reduction between the benchmarks is due both to differences in hit rate, and differences in data memory access rate. The CPU energy (including the cache) increases by roughly 0.5 pJ per cycle when the cache is enabled. Disabling the cache does not entirely eliminate switching activity in the cache, however, so the cache access energy is actually somewhat higher than 0.5 pJ.

The fifth benchmark (thrash) in Figure 4-16 was specifically designed to achieve a hit rate of zero, in oder to demonstrate the worst-case impact of the instruction cache. Enabling the cache for this benchmark causes a 1 pJ increase in the combined cache and CPU energy per cycle. The memory and core energies are not affected.

4.3 Banked Memory

The primary motivation for implementing a banked memory system is reducing average power consumption in the memory. Banking provides two mechanisms for power reduction: smaller SRAM arrays have lower access energy, reducing dynamic power, and multiple banks permit power gating of idle memory segments. (As previously discussed, a third advantage of banking is that it permits multiple memory accesses—such as instruction fetches and DMA transfers—to occur simultaneously, as long as each transaction targets a different bank, thereby eliminating a significant potential performance bottleneck.)

Determining the optimal number and size of the banks requires balancing access energy and leakage power. Smaller banks have lower access energy, but having a large number of banks increases leakage power, due to duplicated periphery logic (address decoders, sense amps, etc.). Too many banks can also increase access energy, if the multiplexing/demultiplexing logic for selecting between banks becomes too complex. The optimal bank configuration therefore depends on the expected average access rate.

The key parameters for the SRAM macros used to implement each bank are leakage power (P_{leak}) and the read (E_{read}) and write (E_{write}) access energies. Average power for the entire memory can then be expressed as

$$P_n(f_a) = nP_{leak}(\ell/n) + f_a\alpha E_{write}(\ell/n) + f_a(1-\alpha)E_{read}(\ell/n) + P_{mux}(n, f_a) \tag{4.5}$$

where ℓ is the total memory size, n is the number of banks, f_a is the access rate, α is the fraction of accesses that are writes, and $P_{mux}(n, f_a)$ is the power consumed by the bank multiplexer/demultiplexer logic. The exact α value is not particularly critical, as E_{read} and E_{write} are generally similar. We assume $\alpha = 0.1$ (10 % of accesses are writes), based on simulations of several benchmark programs. (Note that if an instruction cache is used, this will increase α, by filtering out many instruction fetch reads.) We also neglect the multiplexer/demultiplexer power for simplicity: $P_{mux}(n, f_a) = 0$. Simulation suggested that this circuitry would contribute only about 5 % to the overall

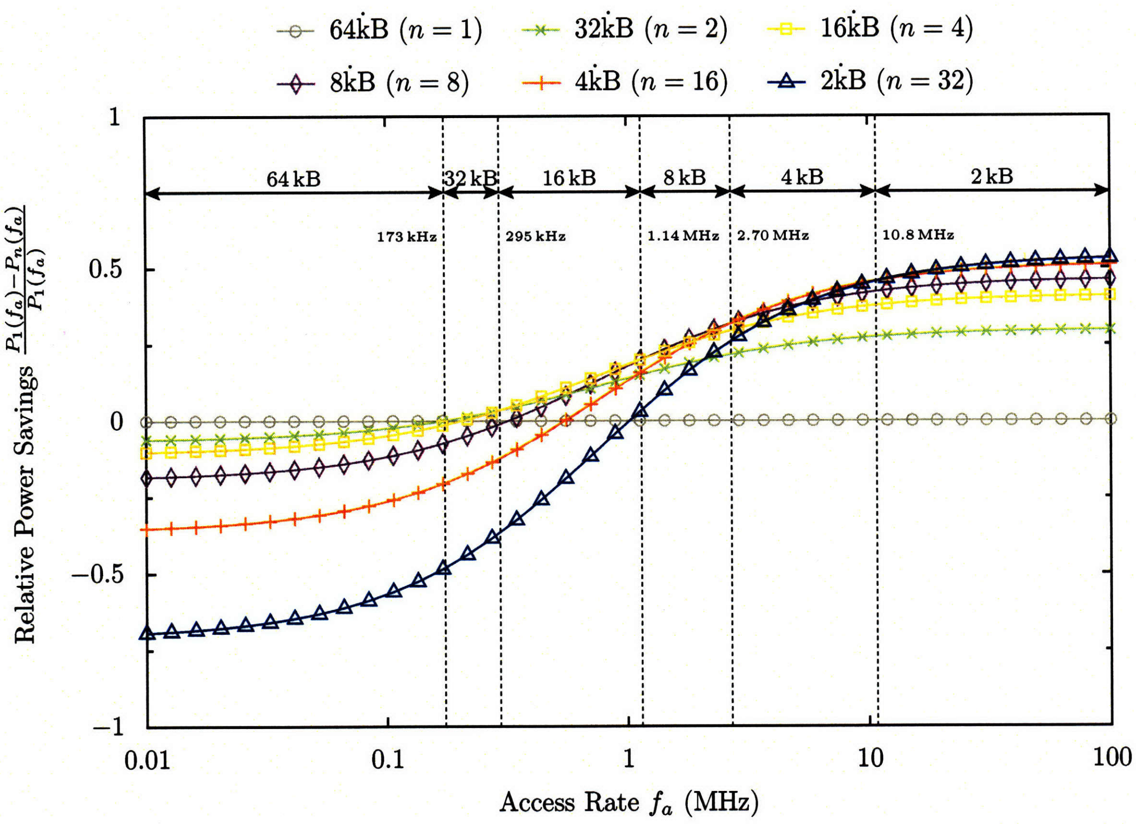


Figure 4-17: Power consumption for a 64 kB memory, implemented using various bank sizes. Power for each implementation has been normalized relative to a single 64 kB bank. The single-bank implementation is efficient only for very low access rates (less than 170 kHz). For higher access rates, arrays composed of 8 kB, 4 kB, or even 2 kB banks are more efficient.

memory access energy. Benini et al. [48] similarly found for their banked memory design the decoder logic accounted for 1.1 % of energy on average, and the wiring between banks for 7.1 %.

We can evaluate various bank configurations by plotting the relative power savings of an n-bank memory over a monolithic ($n = 1$) design, $(P_1 - P_n)/P_1$, as shown in Figure 4-17. The leakage-minimizing monolithic configuration is only optimal for very low access rates (less than 170 kHz). As access rate increases, the optimal number of banks also increases. For the highest access rates, a bank size of 2 kB provides an overall 2× energy savings relative to a monolithic 64 kB memory.

Picking an optimal memory configuration would be as simple as determining the proper value of f_a—except that no single f_a value will accurately characterize a real system at all times. Instruction fetching accounts for the majority of memory accesses, since a new instruction is fetched on essentially every clock cycle when the CPU is active. Most instruction fetches will hopefully be handled by the instruction cache, but the hit rate can vary dramatically depending on the application. The clock speed of the DSP ranges from 50 MHz at 0.8 V down to 4 MHz at 0.45 V. The duty cycle of the processor can also vary, from 100 % down to less than 1 %. Therefore, f_a can range over three orders of magnitude, and can be expected to change dramatically during the course of operation.

In response to the unpredictability of f_a, a heterogeneous memory architecture was settled on for our DSP, initially with seven banks of 8 kB and two banks of 2 kB. The main 8 kB banks would provide optimal or near-optimal storage for access rates below 10 MHz, while the 2 kB banks provide low access-energy storage for frequently accessed data and instructions (such as the program stack, or commonly used subroutines). Before tapeout, the foundry released a new memory compiler with reduced access energy, and as a result, we changed the bank arrangement to six banks of 8 kB and three banks of 4 kB. We have described the bank size analysis using data from the older memory compiler, because we do not have full data from the new compiler. Figure 4-18 shows the actual measured access energy for each memory bank. A side effect of using the improved memories with reduced access energy is that the memory arbiter energy makes up a higher percentage of the total access energy ($20 - 25$ %) than planned.

4.3.1 Memory Arbiter

On our DSP, memory accesses have three possible origins: the CPU instruction bus, the CPU data bus, or the DMA engine. As previously described, the arbiter permits multiple simultaneous memory accesses, as long as the destination of each access is a different memory bank. The access origins are prioritized, so that if multiple transactions are attempted on the same cycle for the same memory bank, one transaction will be allowed to proceed, and the other transactions will receive wait signals, stalling their respective processor core. The arbiter is hardwired to give lowest priority to the DMA engine (DMA operations occur in the background) and highest priority to the

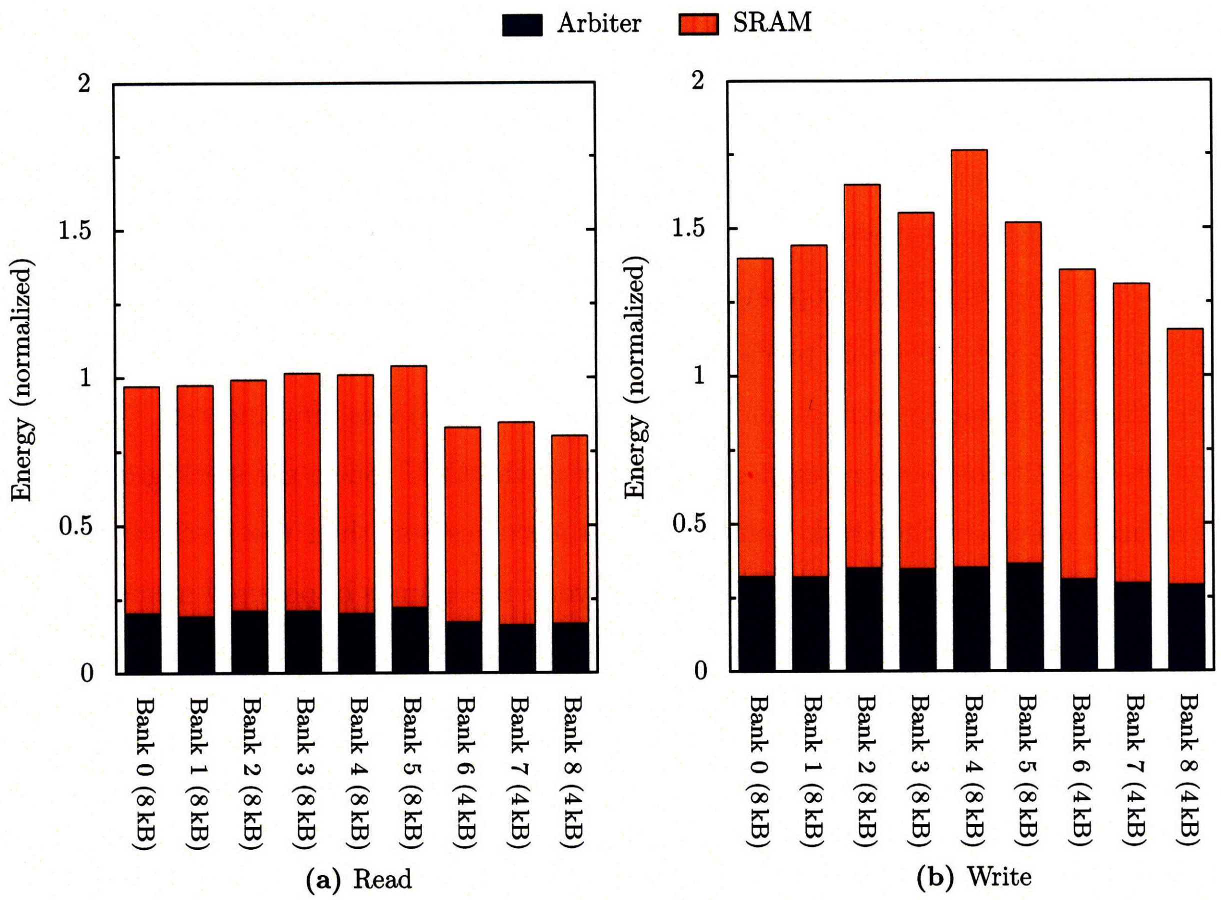


Figure 4-18: Measured SRAM access energy for each bank: (a) reads, (b) writes. Energy has been normalized relative to the average read energy for an 8 kB bank. The average read and write energies of the 4 kB banks are 17 % less than for the 8 kB banks. The write energy varies much more between banks than does the read energy. The variations roughly correlate with the physical distance between the bank and the CPU. The logic overhead (including address and data multiplexing and access arbitration) for each access is 20 − 25 %.

CPU data bus (Giving the CPU databus highest priority allows some simplification in the CPU logic, because load and store operations never stall). Other prioritizations could also be justified, however.

In order to simplify the arbiter logic and thus reduce its energy overhead, some access paths were omitted. The boot ROM cannot be accessed by the DMA engine or the CPU data bus. The peripheral control registers cannot be accessed by the CPU instruction bus. Also, the peripheral registers are treated as one memory bank by the arbiter, so only one peripheral register access is allowed each cycle.

4.3.2 Stack Pointer Limit Exceptions

An important design goal was to provide a mechanism for powering memory banks on and off as the demands of the software change, in real time. The original plan was for all accesses to powered down banks to generate exceptions in the CPU, alerting the processor to power up the appropriate memory bank before continuing. Implementing these memory abort exceptions precisely proved to be overly complicated, so a simpler, more efficient scheme (requiring much less logic in the CPU) was implemented.

A stack limit comparison register was added to the CPU. Since the stack grows downward, whenever the stack pointer register becomes less than the stack limit register (indicating that the stack has grown beyond the bounds of powered memory banks), an exception is generated. The ABI for our processor stipulates that all stack accesses must use a non-negative offset from the stack pointer. The stack pointer register therefore always represents a lower bound on stack addresses.

The stack limit exception is easier to implement than a memory exception, because it occurs after a successful arithmetic operation (modifying the stack pointer register), rather than after an unsuccessful memory operation. There is no need to backup and restart an instruction after the exception.

The disadvantage of this mechanism, compared with complex general memory abort exceptions, is that it does not protect non-stack memory accesses. Thus, it is up to the operating system or application code to manage power gating for the heap and for statically allocated data.

4.4 Combining Banked Memory and Caches

We have seen that instruction caching and memory banking both promise significant memory power savings of roughly the same magnitude. It is possible to combine both techniques in the same system for even greater power reduction. However, interactions between the two techniques reduce their effectiveness when combined, so that the total energy reduction is less than the product of the reductions when each technique is used individually. The filtering effect of a cache reduces the main memory access rate,

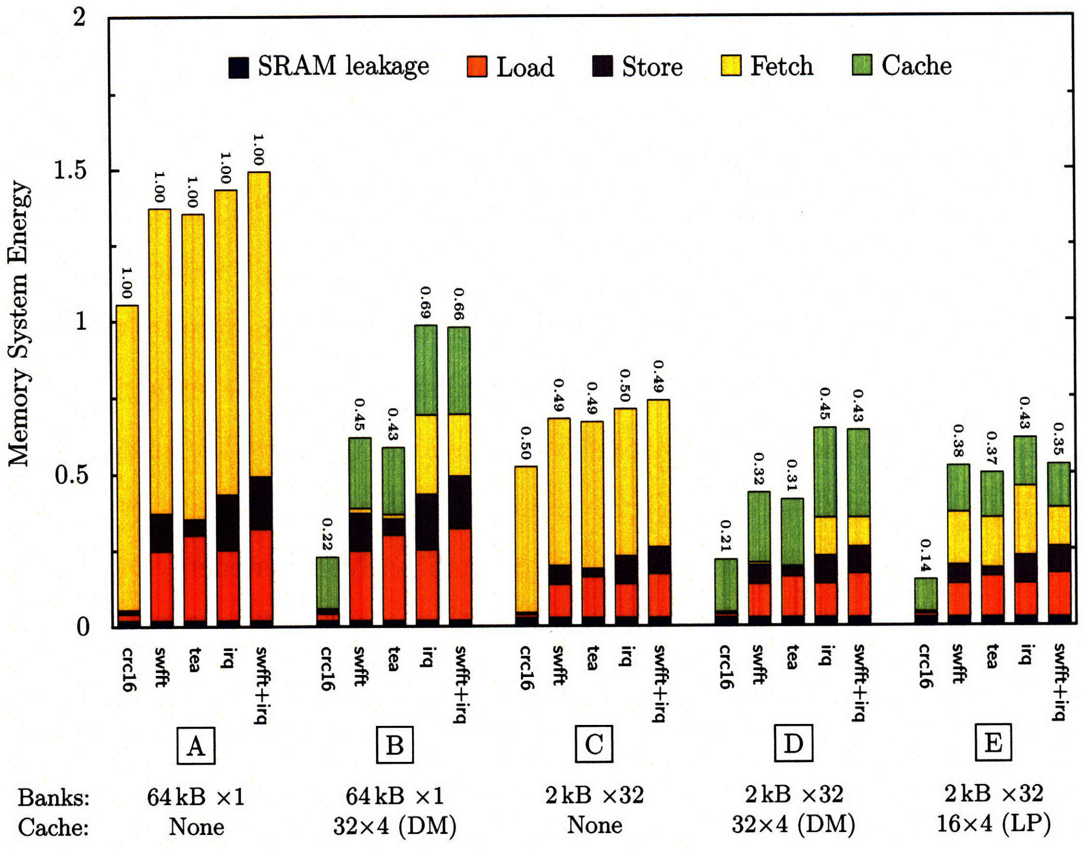


Figure 4-19: Comparison of total memory energy across the benchmark suite for selected memory architectures. Configuration A: monolithic 64 kB SRAM with no cache, B: with instruction caching, C: with banking but no cache, D and E: with caching and banking. The numbers at the top of each bar indicate total memory energy per cycle relative to the baseline configuration (A) for that benchmark. y-axis units are the read access energy for a 64 kB SRAM.

thereby increasing the optimal bank size and decreasing the energy savings available from banking. Conversely, the reduced access energy provided by banking drives the optimal cache size smaller (in order to reduce the cache access energy), resulting in a lower hit rate.

The individual and combined effect of instruction caching and memory banking are illustrated in Figure 4-19, which compares total memory system energy per cycle for the best-in-class configurations of all the techniques discussed so far. Memory system energy is broken down into SRAM leakage, access energy due to load and store operations, cache energy, and instruction memory energy (due to cache misses).

In the baseline configuration (configuration A, a single 64 kB SRAM and no cache), memory energy is dominated by instruction fetching (67 − 95 % of the energy per cycle). Adding a cache (configuration B) dramatically reduces the instruction fetch energy, reducing the total memory energy by 31 − 78 %. The cache configuration shown (32 lines of 4 words) represents the best all-around choice for the benchmark suite. Loop protection was not employed for this configuration. Implementing a banked memory architecture (configuration C) with 32 2 kB banks results in a 50 % memory energy reduction relative to the baseline configuration. The reduction does not depend on the actual memory access pattern and is therefore the same for all benchmarks. This workload ambivalence is an advantage of banking over caches, though for some workloads caches may achieve a significantly higher energy reduction.

Combining caching and multi-banking, as in configurations D and E results in additional savings relative to either technique alone, although the additional savings is not as dramatic as the original factor of roughly 2 ×. In configuration D, a 32×4 cache (with mandatory replacement), combined with a 2 kB bank size results in overall 55 − 79 % energy savings. Finally, configuration E employs loop protection to boost the hit rate of a smaller (16×4) cache to achieve lower total memory energy for several of the benchmarks (though the energy for the other benchmarks increases slightly, due to an increase in fetch energy caused by a reduced hit rate).

4.5 Conclusions

The measured results in Figure 4-16 show conclusively that instruction caching can significantly decrease memory energy per cycle in a micropower processor, achieving up to a 14 × reduction. We are unable to measure the effectiveness of memory banking (since we did not fabricate a chip with a monolithic memory to compare to), but our analysis suggests banking contributes a further 2 × reduction in memory access energy.

Since the access energy for on-chip memories is relatively small, achieving any energy savings requires using a very small cache with an even lower access energy. Determining the optimal cache size (for maximum instruction fetch energy reduction) is a matter of balancing hit rate and cache access energy, both of which increase with cache size. The optimal instruction cache size for our DSP is around 64–128 words. The cache cannot be made much larger than this, or the cache access energy will exceed the memory access energy.

Since the cache is so small, capacity misses are more common than conflict misses, and there is little advantage to implementing an associative cache over a direct-mapped one. Thrashing is often problem for such a small cache, however. The proposed loop protection algorithm is very effective at preventing thrashing, giving near-optimal hit rates for loop-based workloads.

While instruction caching benefits only instruction fetch memory accesses, reducing memory access energy by dividing the memory into smaller banks benefits both instruction and data memory accesses. The optimal bank size depends on the average memory access rate, and is a tradeoff between increasing leakage power (due to duplicated peripheral circuitry for each bank) and decreasing access energy (due to shorter word and bit lines), as the number of banks increases.

Chapter 5

Accelerator Cores

Common data manipulation algorithms frequently involve inherently simple operations not efficiently implementable on general-purpose processors. For example, the fast Fourier transform (FFT) requires sorting either its inputs or outputs into bit-reversed order. In addition to the cost of computing the bit-reversed addresses (for which most general-purpose processors lack a dedicated instruction), this entails extensive (and, as has been shown in previous chapters, energy-expensive) copying between memory locations. The energy cost is particularly regrettable because no essential computation is being performed: with custom hardware the reordering could be accomplished at zero cost merely be rearranging the address lines when the input data is read.

Signal processing operations, such as Fourier transforms, are prime candidates for acceleration using dedicated, algorithm-specific hardware. Other common microsensor tasks which are obvious candidates for acceleration include encryption, data compression, network protocol processing, and operating-system tasks (such as event handling and task scheduling), all of which have been addressed in the literature [12, 14, 49–51].

The µAMPS DSP, being designed for acoustic sensing applications, incorporates accelerators for both FIR filtering and FFTs. The accelerators are implemented as memory-mapped devices. Additional accelerators could be easily added to support other algorithms.

In this chapter, beyond describing the accelerators that were implemented on the DSP, we consider where the energy savings associated with using an accelerator comes from, and how the raw savings from an accelerator translates into net energy savings for a real application. After summarizing related work, we develop an algorithm-generic framework for evaluating accelerator cores. The accelerator cores implemented on the µAMPS DSP are then examined in the context of this framework.

How much energy reduction is possible by employing an accelerator depends as much on the efficiency of the baseline processor as it does on the efficiency of the accelerator. While energy reduction ratios of multiple orders of magnitude have been posted for accelerators in high-performance systems (e.g., desktop PC class), we find that in the domain of energy-optimized micropower processors, energy reduction ratios of up to about $10\times$ are more realistic, at least for numerical algorithm accelerators, such as those implemented in the µAMPS DSP. In the energy-constrained world of microsensors, this nonetheless represents an important savings.

5.1 Related Work

Hardware accelerators are far too ubiquitous to discuss in general. (Virtually every PC, for example, contains an accelerator in the form of a graphics processor.) We therefore limit our consideration to low-power systems, particularly where accelerators are employed for energy efficiency reasons.

A fundamental design consideration is whether to integrate an accelerator into the main processor core (as a coprocessor, for example, where it may share an instruction stream with the main processor), or to keep the accelerator independent (implemented as a memory-mapped peripheral, for example). Accelerators for high-performance systems are often implemented as coprocessors, tightly coupled with the main processor core, and controlled by instructions in the main processor's instruction stream—for example, the speech recognition accelerator in [52] (which achieved an energy improvement of $104\times$ compared to a baseline Pentium 4 software implementation). Lower performance systems, where memory latency is less problematic, are more likely to forgo the complexity of a coprocessor interface. Memory mapped accelerators have the advantage of greater independence from the main processor core (which may facilitate power gating, for example), but may need to duplicate resources already available in the processor (register files, for example).

Hodjat and Verbauwhede [53] implemented an AES encryption accelerator for a LEON (SPARC V8 clone) based system, and compared both memory-mapped and coprocessor-interface versions of the accelerator. They found the coprocessor interface to be 70 % faster and 35 % more energy efficient than the memory-mapped interface, but much of this difference seems to be attributable to the lower I/O bandwidth of the memory-mapped interface. They report a speedup of 64 × and an energy reduction factor of 49 × for their accelerator, relative to software, when implemented on an FPGA.

In an example of an accelerator integrated into the processor of a true micropower system, Kim et al. implemented a lossless data compression accelerator [51] as part of a 16-bit, 24 µW, 4 MHz micropower RISC processor intended for biomedical (EKG) sensing. The accelerator is highly integrated into the CPU core, and consists of an auxiliary 16×16 register file with built-in logic for doing bitwise operations on the entire register file at once, as well as a few special purpose instructions. When not being used for compression, the auxiliary register file is usable as general purpose registers. The authors claim a 20 × speedup using the accelerator hardware, but unfortunately do not report the energy savings.

Some models of the commercial MSP430 processor by Texas Instruments include a 16-bit hardware multiplier, which is implemented as a memory-mapped peripheral [54]. Multiplication operands are written to memory-mapped registers, and the results are read out a few cycles later. Since multiplication even with the accelerator takes several cycles, the overhead of the load and store instructions (compared to a single multiply instruction) is minor. No energy measurements appear to be documented, but the hardware multiplier speeds up 16×16-bit multiplications by 6.4 ×, from 77 to 12 cycles. [54,55] Implementing multiplication outside of the processor core is an unusual design, but has several apparent advantages. Not adding additional instructions to the CPU instruction set not only avoids backwards-compatibility issues, but also prevents the logic that would have been required to decode those additional (infrequently used) instructions from increasing the energy consumption per instruction for all other instructions. An external multiplier is also easily power gated (though the MSP430 documentation does not indicate that they have actually implemented this). While the MSP430 only supplies a multiplication accelerator, the "functional unit as memory mapper peripheral" model could easily be applied to other uncommon but expensive mathematical operations, such as division, square root, floating point operations in general, and trigonometric functions.

5.2 Characterizing Energy Savings

The benefit of an accelerator core, from an energy standpoint, can be quantified as the energy reduction factor (*ERF*), comparing the energy costs of performing a given computation with and without the accelerator.

$$ERF = \frac{\textit{software (unaccelerated) implementation energy}}{\textit{accelerator implementation energy}} = \frac{E_{sw}}{E_{acc}}$$

If we consider a complete system (including processor core, memory, and other peripherals), then the energy savings obtained by using a hardware accelerator can come from two places: intrinsic savings in performing the actual computation (e.g., reduced cycle count and control logic overhead), and extrinsic savings from reduced utilization of global resources (e.g., reducing the number of main memory accesses). Specifically, we define the intrinsic energy (E_{int}) for a given computation to be the energy consumed by the relevant processor core (either the CPU or an algorithm-specific accelerator) during that computation, and the extrinsic energy (E_{ext}) to be the energy consumed by the rest of the system, but which is caused by the computation in question.

For example, computing a 128-point FFT with the µAMPS CPU (not using the accelerator) requires 4.59 ms (22 949 cycles at 5 MHz). If FFT operations are computed

Table 5.1: Variables for accelerator core analysis

Name	*Description*
α_{cyc}	Runtime reduction (number of cycles) achieved by an accelerator
α_{ext}	Extrinsic energy reduction factor for an accelerator
α_{int}	Intrinsic energy reduction factor for an accelerator
α_{pwr}	Power reduction factor for an accelerator
β	Ration of extrinsic energy to intrinsic energy for a given computation
E_{acc}	Energy consumed to perform a given computation, when an accelerator core is utilized
E_{buf}	Energy required to store the input data to a computation in memory, so that the computation can be performed later (e.g., when an accelerator core is available)
E_{ext}	Extrinsic energy for a computation: energy consumed outside of the CPU or accelerator, but which is attributable to the computation in question (e.g., memory access energy)
E_{int}	Intrinsic energy: energy consumed by the CPU or accelerator core while performing a given computation
$E_{startup}$	Energy required to prepare a powered-off accelerator core for use, including power-up energy and initialization energy
E_{sw}	Energy consumed to perform a given computation in software (no accelerator)
ERF	Energy reduction factor
f_c	The rate at which a given computation is performed
$f_{c,min}$	The minimum rate at which an accelerator must be utilized in order to offset the idle power of the accelerator
L	The latency of a system, defined as the time from the moment when the inputs to a computation are ready, to the moment when the computation is actually started
n	The length of a buffer used to hold the inputs of computations which are being deferred until an accelerator core is available. Also the number of points in an FFT.
n_b	The number of butterfly operations required for an FFT
P_{idle}	Power consumed by an accelerator when idle between computations
t_{idle}	Idle time between computations
$t_{idle,max}$	The maximum average time an accelerator can remain idle between computations before its idle power exceeds any energy savings from utilizing the accelerator.

back-to-back, continuously, the average power consumption of the CPU, memory, and core domains is 24.6 µW, 42.7 µW, and 16.6 µW, respectively. The intrinsic energy (energy consumed by the CPU) for each FFT operation is

$$E_{int} = 24.6\,\mu\text{W} \times 4.59\,\text{ms} = 113\,\text{nJ}$$

If all of the memory and core power is caused by the FFT operations (e.g., there are no concurrent DMA transfers occurring), the extrinsic energy per FFT operation is

$$E_{ext} = (42.7\,\mu\text{W} + 16.6\,\mu\text{W}) \times 4.59\,\text{ms} = 272\,\text{nJ}$$

An interesting characteristic of a given computation is the ratio of extrinsic to intrinsic energy.

$$\beta = \frac{E_{ext}}{E_{int}}$$

The β value for an unaccelerated computation can be a guide for designing an accelerator. If $\beta \ll 1$ (intrinsic energy dominates), the accelerator design should focus on reducing the actual computation energy—by exploiting parallelism, using specialized functional units, for example. If $\beta \gg 1$ (extrinsic energy dominates), minimizing external activity—using large local memories, for example—is most important. For our 128-point FFT,

$$\beta = \frac{272\,\text{nJ}}{113\,\text{nJ}} = 2.4$$

An accelerator may reduce intrinsic energy, extrinsic energy, or both. If α_{int} and α_{ext} represent the scaling of intrinsic and extrinsic energy obtained by using an accelerator,

$$E_{int(accelerator)} = \alpha_{int} E_{int(software)}$$

$$E_{ext(accelerator)} = \alpha_{ext} E_{ext(software)}$$

then the overall energy reduction factor can be expressed as

$$ERF = \frac{E_{int} + E_{ext}}{\alpha_{int} E_{int} + \alpha_{ext} E_{ext}} = \frac{1 + \beta}{\alpha_{int} + \alpha_{ext}\beta}$$

Accelerators commonly reduce not only the energy required for a computation, but also the time. We define the speedup factor α_{cyc} for an accelerator as

$$\alpha_{cyc} = \frac{\mathit{accelerator\ runtime}}{\mathit{software\ runtime}}$$

Factoring the change in runtime out of α_{int} leaves

$$\alpha_{pwr} = \frac{\alpha_{int}}{\alpha_{cyc}}$$

where α_{pwr} represents scaling of the average power consumption of the processing module. Generally, minimizing α_{pwr} is not of primary importance; α_{pwr} may be greater than one (for example, if the accelerator exploits parallelism to perform the computation faster) if α_{cyc} is low enough that $\alpha_{int} < 1$. It is conceivable, though uncommon, for an accelerator to have $\alpha_{int}<1$ but $\alpha_{cyc}>1$: the accelerator would be slower than a software implementation, but would use sufficiently less energy per cycle to still yield a net internal energy savings. An example might be an accelerator operating in the subthreshold regime to compute a non-time-critical operation slowly, but very efficiently, in the background.

Continuing our FFT example, using the µAMPS FFT accelerator, a 128-point FFT requires 157 µs (47 µs for the actual transform, plus 110 µs to copy the data into and out of the accelerator). The accelerator power is 177 µW, the CPU power 4.9 µW, the memory power 23.3 µW, and the core power 23.1 µW. The intrinsic energy for the accelerator is thus

$$E_{int} = 177\,\mu\text{W} \times 157\,\mu\text{s} = 27.8\,\text{nJ}$$

and the extrinsic energy is

$$E_{ext} = (4.9\,\mu\text{W} + 23.3\,\mu\text{W} + 23.1\,\mu\text{W}) \times 157\,\mu\text{s} = 8.0\,\text{nJ}$$

The speedup achieved by the accelerator is

$$\frac{1}{\alpha_{cyc}} = \frac{4.59\,\mathrm{ms}}{157\,\mu\mathrm{s}} = 29\,\times$$

(or 97 ×, if we do not count the data copying time). Intrinsic energy has been reduced by

$$\frac{1}{\alpha_{int}} = \frac{133\,\mathrm{nJ}}{27.8\,\mathrm{nJ}} = 4.7\,\times$$

and extrinsic energy is reduced by

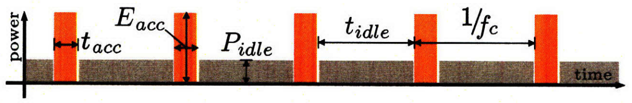
$$\frac{1}{\alpha_{ext}} = \frac{272\,\mathrm{nJ}}{8.0\,\mathrm{nJ}} = 34\,\times$$

Together the factors α_{cyc}, α_{pwr}, and α_{ext} describe where an accelerator's energy savings comes from. Does it perform the computation faster than a software implementation (α_{cyc}<1), perhaps by using more hardware to exploit parallelism (α_{pwr}>1)? Does it perform the computation using simpler hardware than a full general-purpose CPU (α_{pwr}<1)? Or does it employ extra local memory in order to reduce memory traffic or other system resource use (α_{ext}<1)? Usually, some combination of these factors are true.

For ease of interpretation, we will generally report accelerator performance in terms of the inverses of the α values. For example, if $\alpha_{int} = 0.5$ for an accelerator, than the intrinsic energy reduction is reduced by a factor of $1/\alpha_{int} = 2\,\times$.

5.3 Minimum Duty Cycle and Power Gating

Adding accelerator hardware increases the overall power of the system, even when the accelerator is not being used. If the accelerator is well designed, its idle power (P_{idle}) will be almost entirely due to leakage. Consider a stereotypical workload, shown below, where computations are triggered at a fixed rate f_c.



Each time a computation is performed using the accelerator instead of software, $\Delta E = E_{sw} - E_{acc}$ energy is saved; however $P_{idle} \cdot t_{idle}$ energy is lost during the idle time between computations. The maximum average idle time for the accelerator in order to maintain any net energy savings is

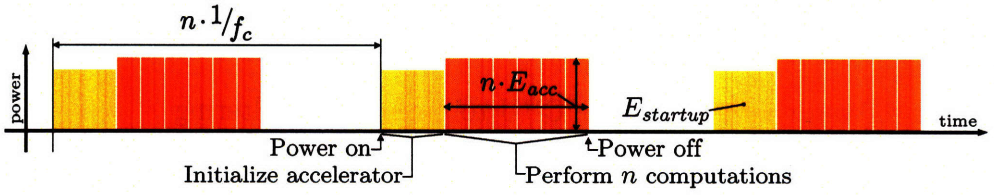
$$t_{idle,max} = \frac{E_{sw} - E_{acc}}{P_{idle}}$$

If the accelerator is effective (*ERF* is reasonably large) and is well designed so that its idle power is much less than its active power (through clock gating, for example), then $t_{acc} \ll t_{idle}$ and the minimum rate at which the accelerator must be used is[1]

$$f_{c,min} \approx \frac{1}{t_{idle,max}} = \frac{P_{idle}}{E_{sw} - E_{acc}} \tag{5.1}$$

Power-gating an accelerator between computations can mitigate the energy wasted due to accelerator idle power. Due to the energy overhead required to power up the accelerator, it is often beneficial to buffer computations in a queue so that when the accelerator is powered on multiple computations can be performed back-to-back, with no idle time. The power-up energy is then amortized over multiple computations.

The number of computations that can be queued may be limited by available memory, or by real-time responsiveness constraints. If computations are generated at a rate f_c, and the latency (defined as the time between when a computation is generated and when it is actually started) is required to be less than L, then the maximum queue length n is limited to $n_{max} = \lfloor L f_c \rfloor$ computations. An example buffered workload is shown below.



The cost of powering up and initializing the accelerator, $E_{startup}$, is the sum of

[1]The precise condition for the approximation to be valid is that $P_{idle}/P_{acc} \ll ERF-1$. To see this, if $t_{idle} \leq (E_{sw} - E_{acc})/P_{idle}$, then $t_{idle}P_{idle} \leq E_{sw} - E_{acc} = E_{acc}(ERF-1)$. Since $E_{acc} = t_{acc}P_{acc}$, then $\frac{t_{idle}}{t_{acc}} \leq \frac{P_{acc}}{P_{idle}}(ERF-1)$. If $\frac{P_{idle}}{P_{acc}(ERF-1)} \approx 0$, then $t_{acc} \approx 0$ and $f_c \geq \frac{1}{t_{acc}+t_{idle}} \approx 1/t_{idle}$.

power switch activation energy E_{switch}, the energy $E_{recharge}$ required to charge up the power supply rails in the accelerator, and any additional energy E_{init} required for re-initializing the accelerator after power-up (i.e., restoring state that was lost during power down). The total power for the power-gated accelerator scenario is

$$P(f_c, n) = f_c \left(\frac{E_{startup}}{n} + E_{acc} + E_{buf} \right)$$

where E_{buf} is the energy associated with storing a computation in the queue and retrieving it later. We can now compute the net power reduction from using the power-gated accelerator

$$\Delta P = f_c \left(E_{sw} - E_{acc} - E_{buf} - E_{startup}/n \right)$$

and the minimum buffer size necessary to achieve any net energy savings

$$n_{min} = \frac{E_{startup}}{E_{sw} - E_{acc} - E_{buf}} \tag{5.2}$$

A simple test program was developed to demonstrate these models. The computation of interest in this example is a 16-tap FIR filter, which can be performed on the µAMPS DSP either in software, or using the filter accelerator (which will be described in detail in Section 5.4.1). In this test application, input data samples for the filter are generated at a programmed rate by the DSP's ADC interface. The outputs of the filter are simply written to an output queue in main memory, where in a real application, they could be processed further.

Four different versions of the application were tested.

1) Using software running on the CPU to perform the filtering

2) Using a hardware accelerator for the filtering, with the accelerator powered continuously

3) Using a power-gated filter accelerator

4) Without any filtering (the ADC samples were written directly to memory)

The code for the four versions is identical, except for the implementation of the filter. The fourth version served as a baseline to isolate just the power associated with the filtering operation in each of the other versions.

Figure 5-1 shows an oscilloscope screen capture of the power-gated accelerator implementation in operation. The ADC interface logic on the DSP automatically triggers ADC conversions (top trace). When a conversion is complete, the CPU is woken from the idle mode: it copies the conversion result from the ADC into the queue in memory, and immediately returns to idle mode (clocks stopped in the CPU). When the queue becomes full, the CPU remains active to power up the FIR accelerator. At a clock speed of 5 MHz, the accelerator is ready to be initialized within one clock cycle of the power being turned on. Because no state is retained in the accelerator during power down, the CPU loads the tap coefficients, as well as the previous 15 input samples (to prime the filter) into the accelerator. The sample points in the buffer are then written to the accelerator, and the filtered output points are written back to memory. (The queue length at the 5 kS/s sample rate shown is $n = 50$.) The CPU is able to return to idle mode for a few clock cycles during each filter operation. When the queue has been emptied, the accelerator is immediately powered down.

Because the four versions of the test program are identical except in the filter implementation, the power associated with the filtering operation can be isolated by subtracting the measured total system power for the fourth (non-filtering) version from the measured power for each of the first three versions. Figure 5-2 shows the filter operation power as a function of the ADC sampling rate f_c, from 0.1 kS/s to 12 kS/s, for each of the three filter implementations.

Each implementation is optimal (lowest power) over some range of sampling rates. The software implementation has the highest energy per filter operation (and thus the highest slope in Figure 5-2), but has no overhead ($P(f_c{=}0) = 0$), and is thus the optimal implementation for the lowest sampling rates. The filter power for this

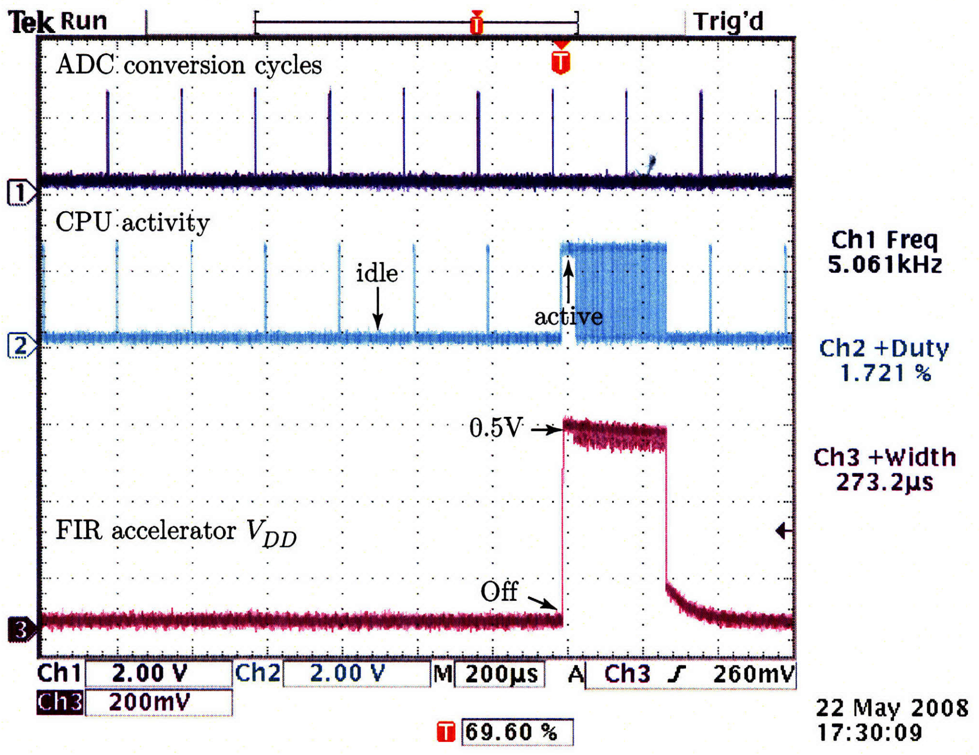


Figure 5-1: Oscilloscope screen capture illustrating power-gated operation of an FIR filter accelerator.

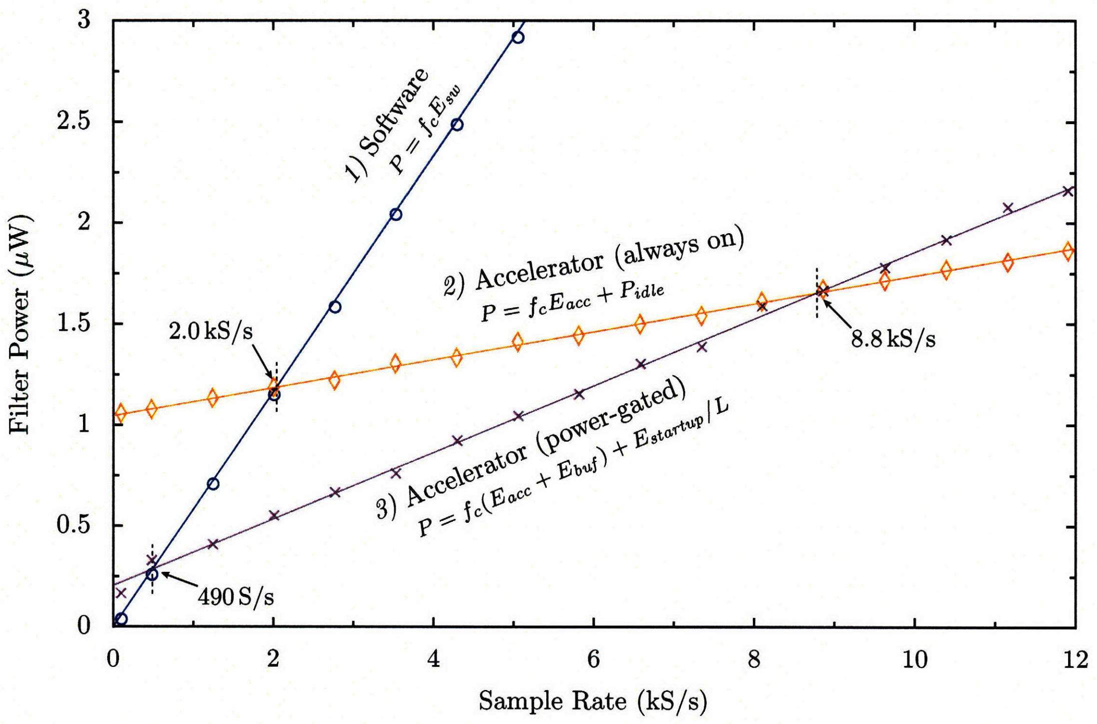


Figure 5-2: Power consumed by a 16-tap symmetric filter as a function of input sample rate. Three implementations are shown: 1) software (no accelerator), 2) hardware accelerator without power gating, and 3) hardware accelerator with power gating.

implementation is directly proportional to the sampling rate.[2]

$$P_{(1)} = f_c E_{sw}$$

The always-on accelerator implementation has the lowest slope, but a significant constant overhead due to the additional idle power of the accelerator.

$$P_{(2)} = f_c E_{acc} + P_{idle}$$

For the power-gated accelerator implementation, the filter power depends on both the sampling rate and the buffer length or latency.

$$P_{(3)} = f_c(E_{acc} + E_{buf} + \frac{E_{startup}}{n}) = f_c(E_{acc} + E_{buf}) + \frac{E_{startup}}{L}$$

In Figure 5-2, the buffer length changes with the sampling frequency, so as to maintain a constant latency of $L = 10\,\text{ms}$.

The model parameters, obtained by least-squares fitting the measurements, are as follows.

Software energy	E_{sw}	580 pJ
Accelerator energy	E_{acc}	69 pJ
Accelerator idle power	P_{idle}	1.0 µW
Accelerator startup energy	$E_{startup}$	2.0 nJ
Buffer energy	E_{buf}	96 pJ

The break-even point between the software implementation and the always-on accelerator implementation occurs at $f_c = 2.0\,\text{kS/s}$, as given by equation 5.1. The break-even point between software and the power-gated accelerator occurs at

$$f_c = \frac{E_{startup}}{L(E_{sw} - E_{acc} - E_{buf})} = 490\,\text{S/s}$$

[2]The idle power of the CPU or other system components between filter operations is not included, because we are considering only the energy involved with the filter computation. In a real application, the CPU would likely have additional work to do between filter operations, after which the CPU could be idled or powered down entirely. We assume that offloading the filter operation to an accelerator will not significantly change the CPU workload or how the CPU is power-managed between filter operations.

which also corresponds to the minimum buffer length ($n_{min} = 4.9$) given by equation 5.2.

The always-on accelerator implementation has a large overhead, $P(f_c{=}0) = P_{idle}$, but requires the least energy per filter operation and is thus the optimal implementation at the highest sampling rates. It becomes inefficient to power-gate the accelerator when the sampling rate reaches

$$f_c = \frac{P_{idle} - E_{startup}/L}{E_{buf}} = 8.8\,\mathrm{kS/s}$$

Increasing the latency would move the line for the power-gated implementation downward, decreasing the sample rate below which the software implementation is advantageous, and increasing the sample rate above which keeping the accelerator always on is advantageous.

If instead of fixing the latency we had fixed the queue length, then the line for the power-gated filter implementation would have increased slope (depending on the queue length) and pass through the origin. The power-gated accelerator implementation would then be preferable to the software implementation at all sampling rates, though the latency would be very large at very low sampling rates.

Although the raw energy reduction factor per filter operation for the accelerator is $ERF = \frac{E_{sw}}{E_{acc}} = 8.4\times$, the effective power reduction depends on the sampling rate, as shown in Figure 5-3. The full *ERF* may not be achievable in practice: for example, at 5 MHz this sample application has a maximum throughput of 43 kS/s (limited by the ADC interface), which corresponds to a filter power reduction of only 6.2 ×.

5.4 Characterizing the μAMPS Accelerators

We have identified a number of parameters describing the effectiveness (ERF, α_{cyc}, α_{pwr}, α_{ext}) and utility (f_c, n_{min}) of a generic accelerator core. This section will detail the implementation of the accelerators on the μAMPS DSP, and examine their effectiveness and utility in terms of those parameters.

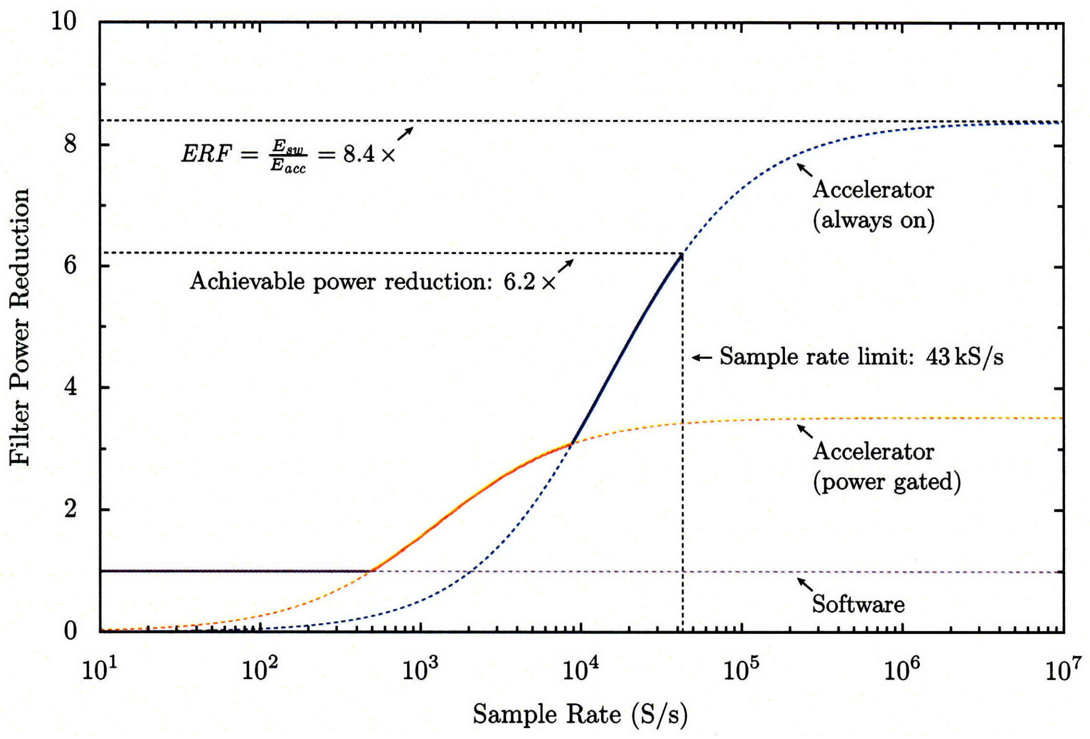


Figure 5-3: Dependence of actual accelerator power reduction on the sampling rate.

5.4.1 FIR Filtering

Filtering is a nearly ubiquitous component in any signal processing application. FIR filters are commonly used because of they are inherently stable and can easily be designed to prevent phase distortion. An N^{th}-order FIR filter is simply an $N+1$ point convolution

$$y[k] = (x * h)[k] = \sum_{i=0}^{N} h_i x[k-i]$$

where $x[k]$ are the input samples, $h_0 \ldots h_N$ are the filter tap coefficients, and $y[k]$ are the filtered output samples. A symmetric filter has $h_i = h_{N-i}$, so the summation can be factored to

$$y[k] = \sum_{i=0}^{(N-1)/2} h_i(x[k-i] + x[k-N+i])$$

(assuming N is odd), thereby reducing the number of multiplications from $N+1$ to $(N+1)/2$. A similar simplification for anti-symmetric filters ($h_i = -h_{N-i}$) is obvious, and hereafter we will use the term "symmetric" to refer to both symmetric and anti-symmetric filters. Symmetric FIR filters are very common in practice, because they have linear phase.

A baseline software implementation of a 15th-order symmetric FIR filter was hand coded in assembly, using fixed-point math. The symmetry of the filter taps is exploited, so only eight multiplications are required to compute each output point. Each pair of taps is computed in five instructions, with an additional five instructions to rescale (by right-shifting) the output value, plus four instructions of loop overhead, for a total of 49 instructions per sample point. Input sample data and tap coefficients are all kept in main memory, so computing each filter output value requires reading 24 words (16 input samples and 8 unique tap coefficients), and writing one word (the output sample). The entire filter loop code fits within the 64-word instruction cache of the CPU, so the cache hit rate is nearly perfect (98 % after processing 128 samples). Measured on the actual chip (at $f = 5\,\mathrm{MHz}$ and $V_{DD} = 0.5\,\mathrm{V}$), the software filter implementation consumes 550 pJ per sample point, with an extrinsic/intrinsic ratio of $\beta = 1.6$. With β close to one, an accelerator for this computation should address both intrinsic and extrinsic energy consumption.

We evaluated three FIR accelerator architectures, which are shown in Figure 5-4. Architecture (a) employs a single multiply-accumulate (MAC) structure that is reused for each unique tap coefficient. An N^{th}-order filter thus requires $N + 1$ cycles per data sample, or $(N + 1)/2$ cycles for a symmetric filter. Two small flip-flop memory arrays hold up to 16 input sample values (in a circular queue), and up to 8 tap coefficients, allowing up to 7th-order asymmetric and 15th-order symmetric filters to be implemented. Control logic computes, on each clock cycle, the appropriate addresses for the input data and coefficient memories. The filter order is determined by the control logic, and can be easily and efficiently scaled.

Architecture (b) is the classic transposed form for a symmetric FIR filter, with one multiplier per unique tap. The transposed form results in a short critical path (one MAC operation) that does not depend on the filter order. The architecture can be programmed for a reduced filter order by loading zeros into some of the coefficient registers. Operand isolation logic could be employed to further reduce switching activity in the unused multipliers and adders, but the leakage and area of the unused

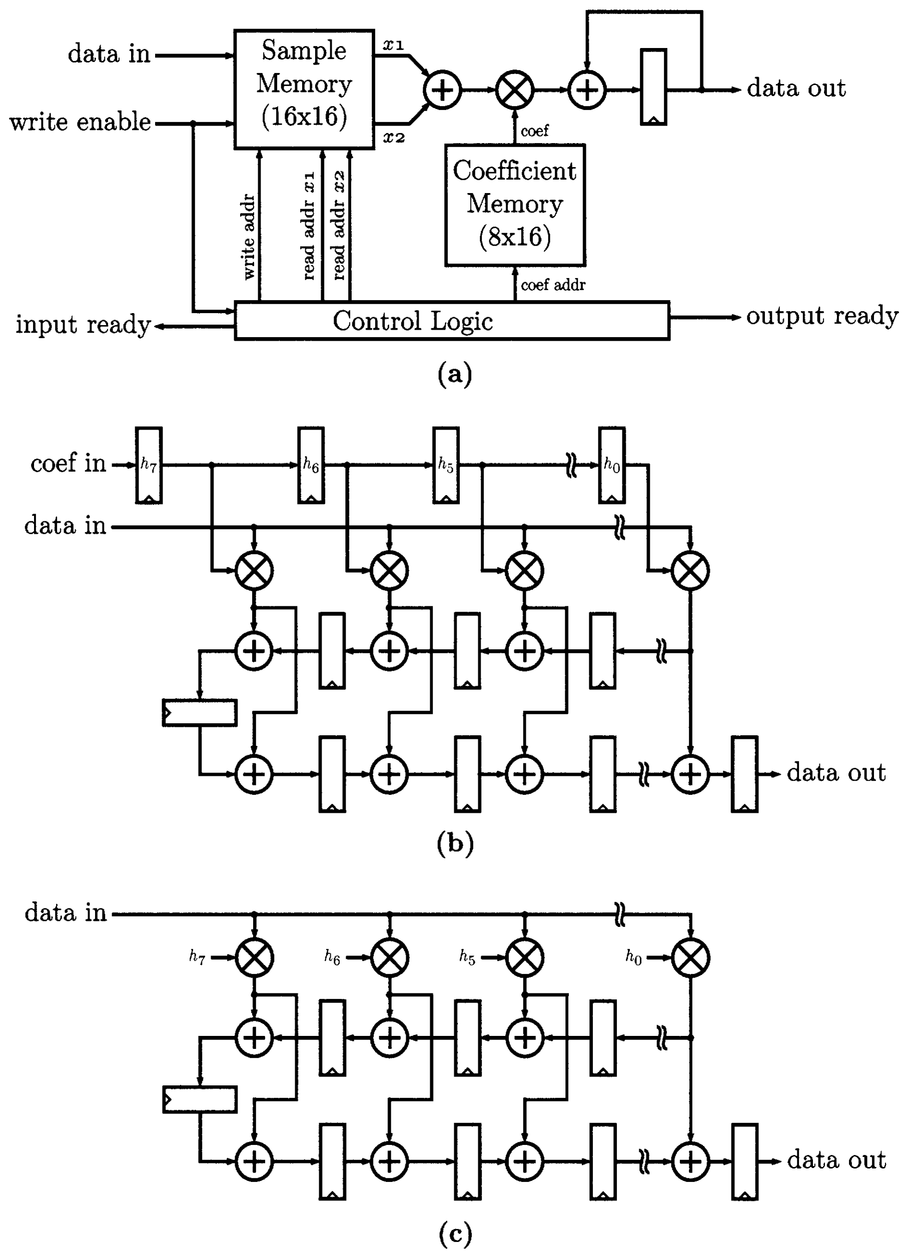


Figure 5-4: FIR filter architectures: (a) sequential, as implemented on the DSP chip, (b) transposed form with programmable coefficients, (c) transposed form with fixed coefficients.

MAC elements will still be significant, especially if the filter must be designed to accommodate much higher order filters than are commonly used in practice.

Architecture (c) is the same transposed form as (b), but with non-programmable tap coefficients. The fixed coefficients not only eliminate the need for the coefficient registers, but also allow much of the multiplier logic to be collapsed. This architecture is obviously not suitable for a general-purpose accelerator, but was considered

Table 5.2: Comparison of FIR accelerator architectures

Data is from simulation with $V_{DD} = 0.5\,\text{V}$. Energy numbers are per sample point.

Architecture	Software Programmable	(a) Sequential Programmable	(b) Transpose Programmable	(c) Transpose Fixed
Cycles/sample	49	8	1	1
Intrinsic energy (pJ)	245	20.1	12.6	5.38
$1/\alpha_{cyc}$	–	6.1	49	49
$1/\alpha_{pwr}$	–	1.94	0.385	0.900
$1/\alpha_{int}$	–	12.2	19.5	45.6
Max frequency (MHz)	–	5.69	7.33	9.11
Leakage (nW)	–	159	528	198
Area (Normalized)	–	1.0	3.3	1.4

only to explore how much additional energy savings is possible when the exact filter configuration is fixed at hardware design time.

The three architectures were synthesized with Synopsys PhysicalCompiler and simulated, using the same filter coefficients and input data as for the software filter implementation that was described earlier. Table 5.2 summarizes the results. The sequential and programmable transposed form architectures achieve comparable reductions in intrinsic energy per sample ($1/\alpha_{int}$) of $14\times$ and $16\times$, respectively. The fixed coefficient architecture of course yields the highest energy reduction, $49\times$. The accelerators were simulated in isolation (not as part of the entire μAMPS DSP), so only intrinsic energy is compared. The memory traffic required is the same for each architecture: one read to load each input sample, and one write to store each output sample. The extrinsic energy reduction would therefore be similar for each architecture.

The accelerators were optimized for energy without regard to clock speed. All three architectures have roughly the same critical path (one MAC operation), and thus all three have roughly the same critical path length. The fixed coefficient architecture is 24 % faster than the transpose architecture with programmable coefficients. The maximum clock rate for the programmable, transpose architecture is 29 % faster than for the sequential architecture, due to the additional control logic required by the sequential architecture. In all three cases, at 0.5 V leakage makes a negligible con-

Table 5.3: FIR accelerator statistics

Energy measurements are per sample point, for a 16-tap symmetric filter.

	Software	Accelerator
Cycles	49	10
Memory reads/writes	24[a]/1	1/1
Intrinsic energy (pJ)	207	35.9
Extrinsic energy (pJ)	339	94.3
Total energy (pJ)	546	130
β	1.6	2.6
Leakage (nW)	–	388
Area (µm^2)	–	23 300

[a]An additional 49 instruction fetches are performed per sample, but these will generally be handled by the instruction cache.

tribution to the total power, so the energy per operation is essentially independent of the clock period.

The programmable, transposed-form architecture has roughly three times the area and leakage of the sequential architecture. The leakage difference will be less consequential if the accelerator is power gated, but the larger accelerator will require a correspondingly larger power switch, and will have a longer break-even time.

The sequential architecture (a) was selected for implementation in the DSP. This architecture has lower area and leakage than a transpose architecture, and the flexibility of the sequential architecture to efficiently implement variable filter lengths, asymmetric filters, and to perform efficient downsampling was judged to outweigh the slight energy-efficiency advantage of a transpose architecture.

Table 5.3 lists statistics for the complete FIR accelerator core, as implemented on the µAMPS DSP. The final accelerator is somewhat more complicated than the bare filter engines illustrated in Figure 5-4, hence these numbers are higher than those for architecture (a) in Table 5.2. The complete DSP implementation requires two additional clock cycles per sample point in order two write the input sample data into the accelerator and read the output sample data. (An N-tap filter thus requires $N/2+2$ cycles per sample point.) The complete accelerator also includes an output scaling unit, which generates the final 16-bit output value from the 35-bit accumulator by shifting right by a programmed amount, with saturation. Finally, the accelerator includes a decimation function, to facilitate digital anti-aliasing (through oversampling of the ADC, low-pass filtering, and downsampling). When k-to-1 decimation is enabled ($k \in \{1 \ldots 32\}$), one output sample is computed for every k input samples written into the filter. The intermediate input samples are simply written to the input sample buffer: no computation is performed.

The filter accelerator achieves a speedup of 4.9 ×, relative to the software implementation. The power consumption of the accelerator is only marginally less than that of the CPU, so the net intrinsic energy reduction is comparable to the speedup factor. The speedup and energy scaling factors for the filter accelerator are as follows.

Speedup	$1/\alpha_{cyc}$	4.9 ×
Processor power reduction	$1/\alpha_{pwr}$	1.2 ×
Intrinsic energy reduction	$1/\alpha_{int}$	5.8 ×
Extrinsic energy reduction	$1/\alpha_{ext}$	3.6 ×
Overall energy reduction factor	ERF	4.2 ×

Measurement of the power-gating parameters for the FIR filter has already been described in sections 2.2 and 5.3; the numbers are repeated below for reference.

Idle power	P_{idle}	1.0 µW
Recharge energy	$E_{recharge}$	125 pJ
Switch energy	E_{switch}	60 pJ
Initialization energy	E_{init}	1.8 nJ
Total startup energy	$E_{startup}$	2.0 nJ
Minimum f_c (always on)	$f_{c,min}$	2.0 kS/s
Minimum buffer length	n_{min}	4.9

Reloading the filter coefficients and repriming the filter after every power-up results in a relatively large startup energy, equivalent to filtering 15 data points. This could be addressed by using data-retention flip-flops [26,56] to preserve the accelerator state during power down, at the cost of increasing power consumption when the accelerator is powered down.

5.4.2 FFT Architecture

The µAMPS FFT accelerator computes transforms on 64-, 128-, 256-, or 512-point complex inputs, or 128-, 256-, 512- or 1024-point real-valued inputs, with 16-bit precision. Real-value n-point FFTs are preformed using an $\frac{n}{2}$-point complex FFT (in $O(n \log n)$ time) followed by minor ($O(n)$ time) post-processing, as described in [57]. (See also Figure 5-6.)

A baseline software implementation of the FFT was written in C and compiled with GCC (version 4.0.3), using a custom ported back-end to generate code for the µAMPS processor.[3] Optimization was enabled with the `-O3` option. Execution time for an n-point transform is approximately $23n \log_2 n + 17n$ cycles. The code comprises 1072 bytes of instructions, plus 2048 bytes for the twiddle factor tables, and achieves an instruction cache hit rate of 47 % (roughly independent of n).

The FFT accelerator design was adopted from the ultra-low voltage FFT processor described in [58]. Bit-precision scaling was not implemented: only 16-bit transforms are supported. Also, the accelerator does not operate at subthreshold voltages, due to the use of standard SRAM macros. Figure 5-5 illustrates the FFT accelerator architecture, including details of the butterfly datapath and local memory implementation. The datapath computes an entire butterfly in one cycle. Local memory holds the complete dataset during computation, so that once the input values are written into the accelerator, no further main memory accesses are necessary for the duration of the transform computation.

To prevent overflow errors, the butterfly datapath generates (and the local memory stores) 17-bit results. An overflow into the 17th bit on any butterfly computation sets a flag in the control logic. The state of the overflow flag at the start of each iteration of butterflies determines whether the 16-bit datapath inputs are taken from the high or low 16 bits of the 17 bits read from the memory. Each iteration with overflows therefore causes the final transform results to be divided by a factor of two. The overflow flag state after each iteration is saved, so that software can determine the overall scaling of the transform results.[4]

The local 512×34 memory (17 bits each for real and imaginary components) is divided into four 128×34 banks, so that two values can be read and two values can be

[3]The software implementation does not perform the rounding and overflow-compensation functions that are built into the accelerator, as these cannot be coded in C with reasonable efficiency.

[4]The overflow handling mechanism was developed by Daniel Finchelstein.

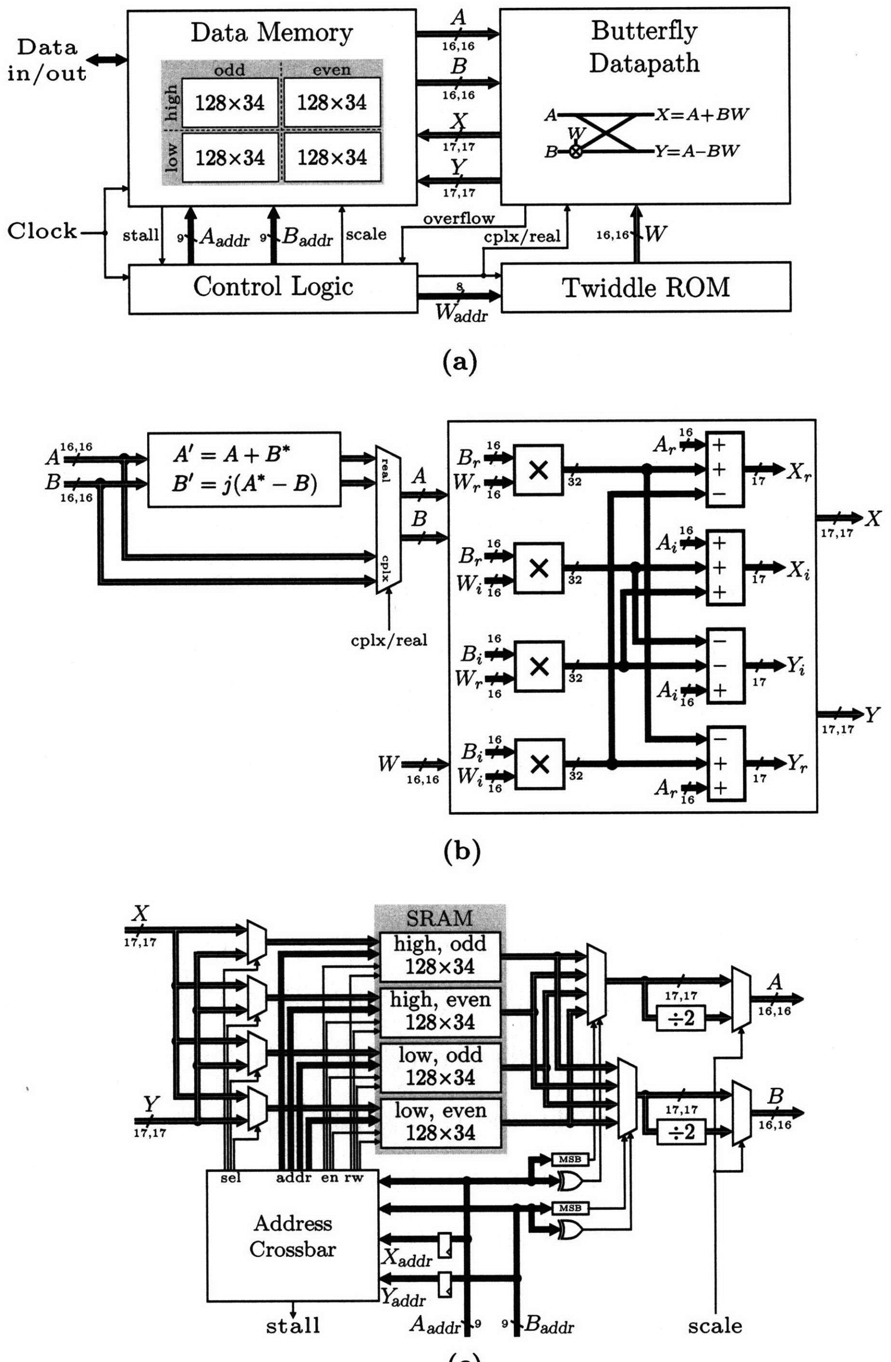


Figure 5-5: FFT accelerator implementation: (a) general architecture, (b) butterfly datapath, including additional logic for the real-value post-processing stage, (c) memory.

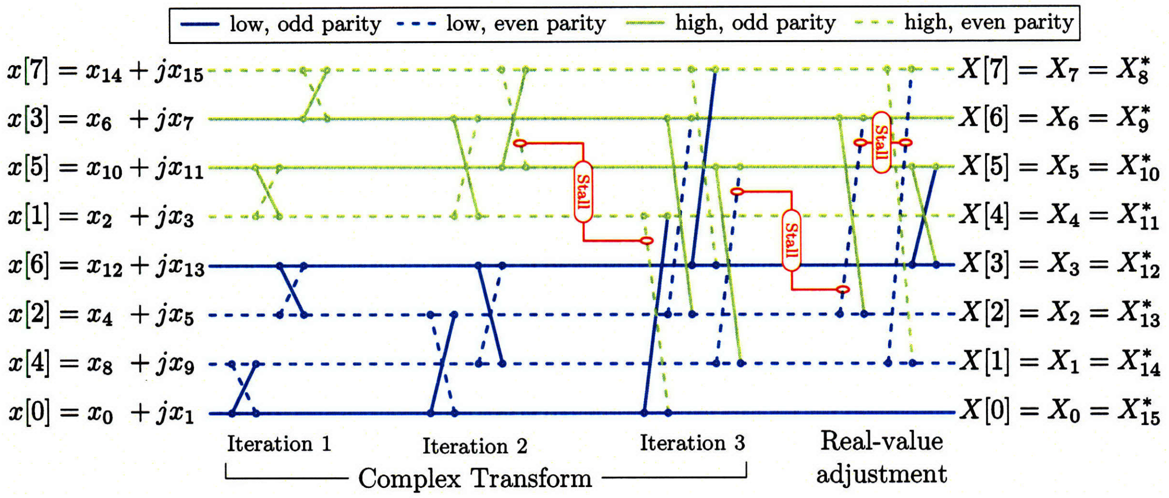


Figure 5-6: Ordering of the butterfly computations so as to minimize hazards. Hazards occur when adjacent butterflies access the same memory bank. A simplified 16-point real-valued transform is shown here. The real input values $x_0 \ldots x_{15}$ are packed into an 8-point complex transform ($x[0] \ldots x[7]$). The output values $X[0] \ldots X[7]$ contain only half of the full transform coefficients ($X_0 \ldots X_{15}$), but the missing coefficients are redundant since the transform of a real-valued signal is symmetric.

written on each clock cycle. Data addresses are split between the four banks based on the MSBs and parity of the addresses. This results in each butterfly computation operating on values from two different banks, so both values can be fetched at the same time. The butterfly operations are specifically ordered, as shown in Figure 5-6 (for a simplified 16-point transform), so that sequential butterflies involve disjoint sets of memory banks. This allows processing one butterfly per clock cycle, with the results from one butterfly being written back to two memory banks while the inputs to the next butterfly are read from the other two banks. (The memory banks are standard, single-port SRAMs, which can perform either a read or write on each cycle.) A small number of hazards are unavoidable, as shown in Figure 5-6, and result in stalling the datapath for one cycle. In particular, a stall is unavoidable before the last iteration of the complex transform, where the high and low halves of the transform are combined. Multiple stalls are also unavoidable during the real-value post-processing stage, as these butterflies do not match the same pattern as the normal FFT butterflies. The accelerator achieves an average throughput of 1.05 cycles per butterfly, a speedup of 95 × relative to the software implementation.

Table 5.4: FFT accelerator statistics

	Software	Accelerator
Cycles	$99.5 \cdot n_b$	$1.05 \cdot n_b$
Intrinsic energy (pJ)	$500 \cdot n_b$	$140 \cdot n_b$
Memory reads/writes	$31 \cdot n_b$ / $13 \cdot n_b$	n / n
Extrinsic energy (pJ)	$1200 \cdot n_b$	$62 \cdot n$
Total energy (pJ)	$1700 \cdot n_b$	$160 \cdot n_b$
β	2.4	0.21
Leakage (µW)	–	6.51
Area (µm^2)	–	175 000

The operating voltage of the FFT accelerator is limited by its use of SRAM for local storage, and the accelerator will not operate at as low a voltage as the CPU, FIR accelerator, and other logic-only portions of the DSP. The FFT accelerator is reliable down to 0.8 V, matching the main memory SRAMs. Although no voltage level converters were used between the FFT accelerator and the remainder of the chip, the FFT works reliably with the accelerator at 0.8 V and the CPU and core logic at 0.5 V.

Energy and runtime comparisons between software and the accelerator are tabulated in Table 5.4, based on measurements on actual silicon at 5 MHz, with main memory and the FFT accelerator powered at 0.8 V and the CPU and other logic at 0.5 V. Energy and runtime for both implementations depend on the length of the transform in question, and are generally proportional to n_b, the number of butterfly operations. For an n-point transform,

$$n_b = \begin{cases} \frac{1}{2} n \log_2 n & \text{complex input transform} \\ \frac{1}{2}(n/2) \log_2 (n/2) + \frac{(n/2)}{2} = \frac{1}{4} n \log_2 n & \text{real input transform} \end{cases}$$

Extrinsic energy for the accelerator is due entirely to copying data into and out of the local memory at the beginning and end of the transform, and is thus proportional to the length of the transform.

Computing an entire butterfly each clock cycle requires four 16×16 multipliers and numerous adders, making the FFT accelerator much larger and power-consuming than the CPU (even including the CPU instruction cache). Also, the accelerator operates at a higher voltage than the CPU. This cancels much of the accelerator's $95\times$ improvement in processing time. The full energy scaling factors are as follows.

Speedup	$1/\alpha_{cyc}$	$95\times$
Processor power *increase*	α_{pwr}	$26\times$
Intrinsic energy reduction	$1/\alpha_{int}$	$3.7\times$
Extrinsic energy reduction	$1/\alpha_{ext}$	$(4.8\log_2 n)\times$
Energy reduction factor	*ERF*	$10.6\times$

Buffer and startup energies for the FFT accelerator were estimated, rather than measured directly. Unlike with the filter accelerator, the FFT accelerator has very little state to maintain between operations, and thus initializing the accelerator after power-up is very inexpensive: all that is required is a write to a single register to set the transform type (real or complex input) and the number of points. The initialization energy is estimated at 30 pJ (three instructions at 10 pJ per instruction). The energy required to buffer an FFT operation will be dominated by the cost of writing and later reading the n input values to memory. E_{buf} is thus estimated at n times the cost of one load and one store instruction (15 pJ each).

Idle power	P_{idle}	7.2 µW
Maximum average idle time	t_{idle}	$1.5 \cdot n_b$ ms
Recharge energy	$E_{recharge}$	680 pJ
Switch energy	E_{switch}	60 pJ
Initialization energy	E_{init}	≈ 30 pJ
Buffer energy	E_{buf}	$\approx 30 \cdot n$ PJ
Minimum buffer length	n_{min}	1

The small startup energy for the FFT accelerator removes the necessity of buffering multiple transforms before powering-up the accelerator. The rate at which the accelerator can be efficiently power-gated is set only by the break-even time discussed in Section 2.2.

The FFT is an exampled of an accelerator that contains much more circuitry than a general-purpose processor, but makes up for it by being much faster. While providing a significant $10\times$ energy reduction, the FFT accelerator is almost $100\times$ faster than our software implementation. Power consumption of the FFT is increased (by an estimated $2.6\times$) by the need to operate at a higher voltage than the CPU, due to the use of SRAMs for local data storage.

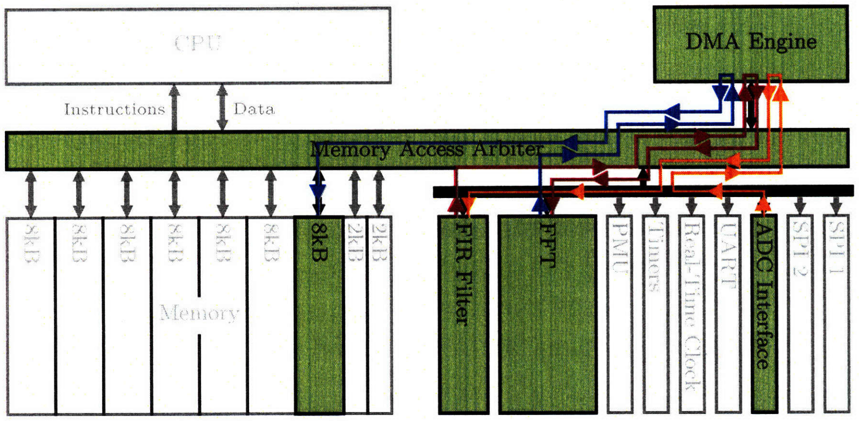


Figure 5-7: The µAMPS DMA engine supports a complete signal processing pipeline—from the ADC interface, through the filter and FFT accelerators, and into main memory—without involving the CPU. A fourth DMA channel is available for serving serial I/O peripherals (e.g., for a radio interface).

5.4.3 Direct Memory Access

A direct memory access (DMA) engine can be considered an accelerator for the trivial—but common—task of copying data from one location to another, such as between peripherals and memory. Efficient data shuffling is especially important for the µAMPS DSP, due to its memory-mapped accelerators.

The µAMPS DSP includes a DMA engine supporting up to four transfers ("channels") in progress at the same time. Only one channel is active on any given clock cycle. Four channels are implemented, so that two channels can be used for copying data from the ADC interface into the FIR filter accelerator and copying the filter results to memory, a third channel can service the FFT accelerator, and the final channel can be used by a radio interface (through an SPI port). As shown in Figure 5-7, the DMA engine enables an ADC ⇒ FIR ⇒ FFT ⇒ MEM processing pipeline to operate without any intervention by the CPU.

A simple memory-to-memory block copy, implemented in software on the CPU, requires four instruction cycles per word copied (assuming no loop unrolling).

```
loop:   ldinc   r3, r0        ; r0 contains the source pointer
        stinc   r3, r1        ; r1 contains the destination pointer
        addi    r2, −1        ; r2 contains the number of words to copy
        bne     loop
```

Table 5.5: DMA engine statistics

Values are per word transferred. Measured at 5 MHz, 0.5 V.

	Software	DMA
Cycles	4	2
Intrinsic energy (pJ)	19	1.8
Memory reads/writes	1/1	1/1
Extrinsic energy (pJ)	35	26
Total energy (pJ)	54	28
β	1.9	14
Idle power[a] (µW)	–	1.0
Area[b] (µm^2)	–	29 362

[a]From simulation, with $V_{DD} = 0.5\,\mathrm{V}$

[b]Estimated using physical prototyping in PhysicalCompiler

The DMA engine performs this copy at two cycles per word, alternating between reading from the source on one cycle and writing to the destination on the next cycle. More importantly, less energy is expended per cycle, because the DMA engine is significantly simpler than the CPU. Table 5.5 compares software- and DMA-based block copy operations (tested using a block length of 512 words). Although the DMA engine is only twice as fast as the CPU, it reduces the intrinsic energy by a factor of 10. Extrinsic energy cannot be significantly reduced, of course, since the memory accesses are the actual operation being accelerated. A small (26 %) reduction in extrinsic energy was measured, which is probably a result of reduced memory address glitching generated by the DMA engine, compared with the CPU.

The DMA engine achieves an energy reduction factor of 1.9 ×, almost identical to its speedup factor of 2 ×. The energy scaling factors are as follows.

Speedup	$1/\alpha_{cyc}$	2.0 ×
Processor power reduction	$1/\alpha_{pwr}$	5.2 ×
Intrinsic energy reduction	$1/\alpha_{int}$	10 ×
Extrinsic energy reduction	$1/\alpha_{ext}$	1.4 ×
Energy reduction factor	ERF	1.9 ×

The DMA controller is not power-gated. Not having its own isolated power domain, the DMA power cannot be measured directly on the actual chip. In simulation at 0.5 V, the idle power of the DMA engine is 1.03 µW (220 nW of which is leakage). The minimum memory-to-memory DMA activity required to justify the DMA idle power is 39.6 kwords/s—a duty cycle of 1.6 % given a DMA bandwidth of 2.5 Mwords/s at 5 MHz. Few applications will utilize this much DMA activity on a continuous basis, so in a future version of the DSP, we would likely opt to power gate the DMA engine so that it can be turned off when not needed. Also, the idle power of the engine could likely be reduced with more aggressive clock gating.

5.5 Conclusions

In the realm of micropower processors, a hardware accelerator may provide a large (multiple orders of magnitude) speedup over software, but the energy savings are likely not as large. Based on the µAMPS DSP accelerators, energy reduction factors of $2 - 10\times$ are reasonable to expect for numeric algorithms. Although this is much lower than the energy reductions that have been achieved by accelerators in higher performance systems, it is expectable because micropower processors are themselves highly optimized for energy efficiency, and therefore software implementations of algorithms on micropower processors waste less energy than the same algorithms running on higher-performance processors. Accelerators for bit manipulations algorithms (e.g., encryption, compression, error correction) might achieve higher energy reduction factors, because general purpose processors are usually limited to operating on whole words at a time, and much less efficient at processing individual bits. (In [58], a StrongARM processor is reported to consume 59 µJ performing a 1024-point FFT. Relative to this baseline, the µAMPS FFT accelerator achieves a $144\times$ energy reduction per transform.)

The energy reduction afforded by an accelerator may be the result of the accelerator doing more useful work per cycle than a software implementation (reducing the number of cycles required), or by expending less energy per cycle, and/or by reducing the activity and energy consumption generated in parts of the chip outside of the processor (another variation on the theme of this thesis of minimizing memory energy). The overall power reduction achieved from an accelerator in a real system depends on the frequency with which the accelerator is used: the idle power of the accelerator between computations counteracts the energy saved during each computation. Unless an accelerator is going to be used constantly, it should be power gated. When the accelerator is used relatively constantly, but at a low rate, it may be advantageous to queue data waiting to be processes, so that the accelerator can perform many computations at once, each time it is powered on.

Chapter 6

Conclusion

6.1 Thesis Conclusions

At 4 MIPS and 40 µW, the µAMPS DSP achieves its design goal of delivering the performance required to implement acoustic sensing applications at power levels that enable self-powered (energy harvesting) operation. This was achieved not just through very low voltage (0.45 V) operation, but also by architectural optimizations specifically tailored for the micropower domain.

Memory access energy is the largest source of power consumption in a complete micropower processor, but has been mostly overlooked in previously published papers that often only report CPU energy consumption. Four different mechanisms are used to reduce memory power in the µAMPS DSP. Dividing the memory into multiple banks reduces the access energy for each bank by approximately 2 ×. Employing an instruction cache reduces the memory access rate due to instruction fetching, resulting in up to a 14 × measured reduction in memory power (depending on the cache hit rate). Utilizing accelerator cores for filter and FFT operations reduces the number of data memory accesses incurred for those computations, resulting in up to a 48 × reduction in extrinsic (memory) energy for a 1024-point FFT. Finally, power gating inactive memory banks reduces memory leakage power proportional to the number of banks that are powered off.

Utilizing the µAMPS accelerator cores, FIR filter and FFT operations are performed $5 - 95\times$ faster and with overall $4 - 10\times$ less energy than software implementations of the same algorithms. Because micropower processors like the µAMPS CPU are designed to be energy-efficient, it is unrealistic to expect the accelerators to provide energy reduction ratios of multiple orders of magnitude (as have been demonstrated with higher-performance systems), at least for numeric algorithms. Algorithms that are particularly inefficient to implement on a general-purpose CPU (such as encryption, compression, or other bit-manipulation intensive tasks) would likely show greater energy reductions from acceleration.

The idle power overhead of adding accelerators to our DSP is mitigated by power gating the accelerators when they will be idle for extended periods of time. Whether an accelerator core should be powered off between operations depends on the rate at which the accelerator is being utilized, and the amount of latency that can be tolerated for queueing operations as they arrive so they can be processed all at once.

Leakage power is often a significant factor in high-performance 90 nm and smaller processes. However, because we fabricated the DSP in a process optimized for low-power rather than high speed, leakage power is well controlled and only significant during idle periods. From the performance versus voltage plot in Figure 1-6, the dynamic energy per cycle is $10\times$ larger than the leakage energy. When the DSP is active, therefore, optimizations which minimize dynamic power are most important.

When components of the DSP are idle, leakage power is mitigated through power gating. Off chip power switches afford high leakage reduction, but lead to long break-even times. Measured break-even times for our chip range from about 500 cycles to many thousands of cycles, long enough for power management by software. On-chip switches (with much reduced switch energy) will result in shorter times possibly requiring hardware-based management.

6.2 Future Work

During the design of the DSP, many good ideas were abandoned due to time constraints and the need to keep the project focused. Were we to design a second-generation of this chip, the following are some of the changes we would make.

- *On-chip power switches.* Implementing the power-gating switches on-chip would streamline the design, reducing the number of off-chip components, allow the switches to be sized according to the load they were powering (reducing the break-even time), and enabling power-gating to be applied at a finer granularity. For example, individual functional units (the multiplier and barrel shifter in the CPU) could be power gated, as well as each of the on-chip peripherals.

- *Improved clock gating.* We relied almost entirely on automatically inserted clock gating, which affected only the lowest levels of the clock tree, leaving most of the tree toggling on every cycle—which consumes $2 - 3\,\mathrm{pJ}$ per cycle. Manually inserting module-level clock gating could eliminate most of this wasted energy. A worthy challenge would be controlling the module-level clock gates automatically, so that they were transparent to software.

- *Low-voltage SRAMs.* We used standard, foundry-supplied SRAM macros, which were not designed for low-voltage operation. As a result, the SRAM portions of the DSP must be powered at a higher voltage than the logic portions. Utilizing custom SRAMs designed to operate at lower voltages [59–63] could reduce the number of power supply voltages required, and would significantly reduce the all-important memory access energy.

This thesis concentrates on hardware design, and assumes that control of power management features such as power gating can be accomplished in software. Effectively controlling all of the various power management knobs that can be implemented in hardware is a growing problem. Exposing these controls to the compiler, operating system, or user software theoretically enables greater energy savings by allowing the control algorithms to be customized to each different user application. In practice, this is an unreasonable demand on the software developers. Encapsulating the control of hardware power management features within the hardware, making power management as transparent as possible to software, is a better way to ensure those features are actually used, and not just ignored.

A key missing feature of all micropower processors published to date—including the µAMPS DSP—is on-chip non-volatile memory (e.g., flash memory). Utilizing non-volatile memory for program storage would open new opportunities for optimizing the memory system power, such as power gating portions of the instruction memory holding infrequently executed code. The quantum-mechanical effects involved in programming flash memories require high voltages ($\approx 10\,\mathrm{V}$), but the read mechanism is similar to SRAM. It should be possible to design flash memories that readback at very low voltages.

Appendix A

Instruction Set

A.1 Notation

ra, rb, rc	Any general purpose register, r0 through r7
rh	Any special purpose register, r8 through r15
$k_{n,s}$	An n-bit signed constant
$k_{n,u}$	An n-bit unsigned constant
k_{periph}	The address of a peripheral control register
$mem_{16}[x]$	The 8-bit memory location at address x
$mem_8[x]$	The 16-bit memory location at address x

A.2 Condition codes

The condition code register (r15) contains seven active flags (Z, N, C, V, I, F, and Q), plus seven saved flags (Z_e, N_e, C_e, V_e, I_e, F_e, and Q_e) that are used to preserve the state of the active flags during exceptions.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	–	Q_e	F_e	I_e	V_e	C_e	N_e	Z_e	–	Q	F	I	V	C	N	Z
Reset	–	0	0	0	0	0	0	0	–	0	0	0	0	0	0	0
Access	–	rw	rw	rw	rw	rw	rw	rw	–	rw	rw	rw	rw	rw	rw	rw

Z—Zero
Indicates that the result of the last operation was zero

N—Negative
Indicates that the result of the last operation was negative

C—Carry
: Indicates that a carry out of the MSB occurred during the last operation

V—Overflow
: Indicates that an overflow occurred during the last operation

I—Interrupt Enable
: When clear, interrupt requests are ignored

F—Cache Freeze
: When set, the cache lines are locked. A cache miss due to a tag mismatch will not cause the affected line to be evicted.

Q—Cache enable
: When set, the cache is enabled

When an exception occurs, the Z, N, C, V, I, F, and Q flags are copied into the the Z_e, N_e, C_e, V_e, I_e, F_e, and Q_e slots, and the I flag is cleared. When a return-from-exception (rfe) instruction is executed, the Z, N, C, V, I, F, and Q flags are restored from their saved values in Z_e, N_e, C_e, V_e, I_e, F_e, and Q_e.

A.3 Branches

B—Unconditional branch

Syntax: b $k_{11,s}$
Encoding: $\boxed{0\;0\;0\;0\;0}\boxed{k\;k\;k\;k\;k\;k\;k\;k\;k\;k\;k}$
Operation: pc $\Leftarrow$ pc $+ 2k$

Skips the next $k-1$ instructions. ($k=1$ is essentially a no-op. $k=0$ results in an infinite loop.)

BCC—Branch if carry clear

Syntax: bcc $k_{11,s}$
Encoding: $\boxed{0\;1\;0\;1\;1}\boxed{k\;k\;k\;k\;k\;k\;k\;k\;k\;k\;k}$
Operation: If $\overline{C}$, then pc $\Leftarrow$ pc $+ 2k$

If the C flag is clear, the next $k-1$ instructions are skipped. *Alias: bgeu (unsigned branch if greater than or equal)*

BCS—Branch if carry set

Syntax: bcs $k_{11,s}$
Encoding: $\boxed{0\;1\;0\;1\;0}\boxed{k\;k\;k\;k\;k\;k\;k\;k\;k\;k\;k}$
Operation: If C, then pc $\Leftarrow$ pc $+ 2k$

If the C flag is set, the next $k-1$ instructions are skipped. *Alias: bltu (unsigned branch if less than)*

BDNE—Decrement and branch if not equal

Syntax: bdne $k_{8,u}$

Encoding:

0 1 1 1 1	a a a	k k k k k k k k

Operation: If ra $= 0$, then
  ra $\Leftarrow$ ra $- 1$
  pc $\Leftarrow$ pc $- 2k$

If the contents of the general-purpose register ra are not zero, ra is decremented by one, and control skips backward by k instructions. (Only backward branches are supported.)

BEQ—Branch if equal

Syntax: beq $k_{11,s}$

Encoding:

0 0 0 1 0	k k k k k k k k k k k

Operation: If Z, then pc $\Leftarrow$ pc $+ 2k$

If the Z flag is set, the next $k - 1$ instructions are skipped.

BGE—Branch if greater than or equal

Syntax: bge $k_{11,s}$

Encoding:

0 0 1 1 0	k k k k k k k k k k k

Operation: If $N = V$, then pc $\Leftarrow$ pc $+ 2k$

If the N and V flags are either both set or both clear, the next $k-1$ instructions are skipped.

BGT—Branch if greater than

Syntax: bgt $k_{11,s}$

Encoding:

0 0 1 1 1	k k k k k k k k k k k

Operation: If $N = V \wedge \overline{Z}$, then pc $\Leftarrow$ pc $+ 2k$

If the N and V flags are either both set or both clear and the Z flag is clear, the next $k - 1$ instructions are skipped.

BGTU—Branch if greater than (unsigned)

Syntax: bgtu $k_{11,s}$

Encoding:

0 1 0 0 1	k k k k k k k k k k k

Operation: If $\overline{Z} \wedge \overline{C}$, then pc $\Leftarrow$ pc $+ 2k$

If the Z and C flags are both clear, the next $k - 1$ instructions are skipped.

BL—Unconditional branch and link

Syntax: bl $k_{11,s}$

Encoding:

0 0 0 0 1	k k k k k k k k k k k

Operation: lp $\Leftarrow$ pc $+ 2$
  pc $\Leftarrow$ pc $+ 2k$

Skips the next $k - 1$ instructions, saving the address of what would have been the next instruction in lp (r11).

BLE—Branch if less than or equal

Syntax: ble $k_{11,s}$
Encoding: $\boxed{0\;0\;1\;0\;1}\boxed{k\;k\;k\;k\;k\;k\;k\;k\;k\;k\;k}$
Operation: If $Z \vee N$, then pc $\Leftarrow$ pc $+ 2k$

If either of the Z or N flags are set, the next $k - 1$ instructions are skipped.

BLEU—Branch if less than or equal (unsigned)

Syntax: bleu $k_{11,s}$
Encoding: $\boxed{0\;1\;0\;0\;0}\boxed{k\;k\;k\;k\;k\;k\;k\;k\;k\;k\;k}$
Operation: If $Z \vee C$, then pc $\Leftarrow$ pc $+ 2k$

If either the Z or C flags are set, the next $k - 1$ instructions are skipped.

BLT—Branch if less than

Syntax: blt $k_{11,s}$
Encoding: $\boxed{0\;0\;1\;0\;0}\boxed{k\;k\;k\;k\;k\;k\;k\;k\;k\;k\;k}$
Operation: If N, then pc $\Leftarrow$ pc $+ 2k$

If the N flag is set, the next $k - 1$ instructions are skipped.

BNE—Branch if not equal

Syntax: bne $k_{11,s}$
Encoding: $\boxed{0\;0\;0\;1\;1}\boxed{k\;k\;k\;k\;k\;k\;k\;k\;k\;k\;k}$
Operation: If $\overline{Z}$, then pc $\Leftarrow$ pc $+ 2k$

If the Z flag is clear, the next $k - 1$ instructions are skipped.

BVC—Branch if overflow clear

Syntax: bvc $k_{11,s}$
Encoding: $\boxed{0\;1\;1\;0\;1}\boxed{k\;k\;k\;k\;k\;k\;k\;k\;k\;k\;k}$
Operation: If $\overline{V}$, then pc $\Leftarrow$ pc $+ 2k$

If the flag is clear, the next $k - 1$ instructions are skipped.

BVS—Branch if overflow set

Syntax: bvs $k_{11,s}$
Encoding: $\boxed{0\;1\;1\;0\;0}\boxed{k\;k\;k\;k\;k\;k\;k\;k\;k\;k\;k}$
Operation: If V, then pc $\Leftarrow$ pc $+ 2k$

If the V flag is set, the next $k - 1$ instructions are skipped.

A.4 Arithmetic

ADD—Add

Syntax: add ra rb rc
Encoding: $\boxed{1\;1\;1\;0\;1}\boxed{a\;a\;a}\boxed{b\;b\;b}\boxed{c\;c\;c}\boxed{0\;0}$
Operation: ra $\Leftarrow$ rb $+$ rc

Adds the contents of registers rb and rc and stores the result in register ra. The

Z, N, C, and V flags are all updated.

ADDC—Add with carry

Syntax: addc ra rb
Encoding: | 1 1 1 1 1 | a a a | b b b | 1 0 1 1 0 |
Operation: ra $\Leftarrow$ ra + rb + C

Adds the contents of registers ra and rb, along with the C flag, and stores the result back in ra. The N, C, and V flags are updated based on the results of the addition. The Z flag is cleared if the result is not zero, but is not changed if the result is zero.

ADDI—Add immediate

Syntax: addi ra $k_{8,s}$
Encoding: | 1 1 0 0 1 | a a a | k k k k k k k k |
Operation: ra $\Leftarrow$ ra + k

Adds the 8-bit signed immediate k to the contents of general-purpose register ra, and stores the results back in ra. The Z, N, C, and V flags are all updated.

ADDS—Add to stack pointer

Syntax: adds ra
Encoding: | 1 1 1 1 1 | a a a | 1 1 0 1 1 1 1 1 |
Operation: sp $\Leftarrow$ sp + ra

Adds the contents of register ra to the stack pointer (r12).

ADDSI—Add immediate to stack pointer

Syntax: addsi $k_{11,s}$
Encoding: | 0 1 1 1 0 | k k k k k k k k k k k |
Operation: sp $\Leftarrow$ sp + k

The 11-bit signed constant k is added to the stack pointer (r12).

CMP—Compare

Syntax: cmp ra rb
Encoding: | 1 1 1 1 1 | a a a | b b b | 1 0 1 0 1 |
Operation: ra − rb

Subtracts the contents of register rb from register ra and updates the Z, N, C, and V flags accordingly. The results of the subtraction are not saved.

CMPI—Compare with immediate

Syntax: cmpi ra $k_{8,s}$
Encoding: | 1 1 0 0 0 | a a a | k k k k k k k k |
Operation: ra − k

Subtracts the signed, 8-bit constant k from the contents of general-purpose register ra, but does not store the result back into ra. The condition codes Z, N, C, and V are updated based on the result of the subtraction.

NEG—Negate

Syntax:	neg ra rb
Encoding:	$\begin{array}{\|c\|c\|c\|c\|}\hline 1\,1\,1\,1\,1 & a\,a\,a & b\,b\,b & 1\,1\,0\,0\,1 \\ \hline\end{array}$
Operation:	$\mathrm{ra} \Leftarrow -\mathrm{rb}$

Stores the twos-complement inverse of rb in ra.

SUB—Subtract

Syntax:	sub ra rb rc
Encoding:	$\begin{array}{\|c\|c\|c\|c\|c\|}\hline 1\,1\,1\,0\,1 & a\,a\,a & b\,b\,b & c\,c\,c & 1\,0 \\ \hline\end{array}$
Operation:	$\mathrm{ra} \Leftarrow \mathrm{rb} + \mathrm{rc}$

Subtracts the contents of register rc from rb and stores the result in register ra. The Z, N, C, and V flags are all updated.

SUBC—Subtract with carry

Syntax:	subc ra rb
Encoding:	$\begin{array}{\|c\|c\|c\|c\|}\hline 1\,1\,1\,1\,1 & a\,a\,a & b\,b\,b & 0\,1\,1\,1\,0 \\ \hline\end{array}$
Operation:	$\mathrm{ra} \Leftarrow \mathrm{ra} - \mathrm{rb} - C$

Subtracts the contents of register rb and the C flag from register ra, and stores the result back in ra. The N, C, and V flags are updated based on the results of the addition. The Z flag is cleared if the result is not zero, but is not changed if the result is zero.

SUBS—Subtract from stack pointer

Syntax:	subs ra
Encoding:	$\begin{array}{\|c\|c\|c\|}\hline 1\,1\,1\,1\,1 & a\,a\,a & 0\,0\,1\,1\,1\,1\,1\,1 \\ \hline\end{array}$
Operation:	$\mathrm{sp} \Leftarrow \mathrm{sp} - \mathrm{ra}$

Subtracts the contents of register ra from the stack pointer (r12).

A.5 Logical

AND—Bitwise AND

Syntax:	and ra rb rc
Encoding:	$\begin{array}{\|c\|c\|c\|c\|c\|}\hline 1\,1\,1\,0\,1 & a\,a\,a & b\,b\,b & c\,c\,c & 0\,1 \\ \hline\end{array}$
Operation:	$\mathrm{ra} \Leftarrow \mathrm{rb} \vee \mathrm{rc}$

Performs a bitwise AND of the contents of rb and rc, and stores the result in register ra. The Z flag is set if the result is zero, and cleared otherwise.

CLRBIT—Clear bit

Syntax:	clrbit ra $k_{4,u}$
Encoding:	$\begin{array}{\|c\|c\|c\|c\|}\hline 1\,1\,1\,1\,0 & a\,a\,a & k\,k\,k\,k & 1\,0\,1\,1 \\ \hline\end{array}$
Operation:	$\mathrm{ra} \Leftarrow \mathrm{ra} \wedge \overline{1 \ll k}$

Clears bit k of register ra.

IAND—Invert and AND

Syntax: iand ra rb

Encoding:

1 1 1 1 0	a a a	b b b	1 1 1 1 1

Operation: $\mathrm{ra} \leftarrow \mathrm{ra} \wedge \overline{\mathrm{rb}}$

Performs a bitwise AND of the contents of ra and th bitwise inverse of rb, and stores the result in register ra. The Z flag is set if the result is zero, and cleared otherwise.

INV—Bitwise inverse

Syntax: inv ra rb

Encoding:

1 1 1 1 1	a a a	b b b	0 1 0 1 1

Operation: $\mathrm{ra} \Leftarrow \overline{\mathrm{rb}}$

Stores the bitwise inverse (NOT) of the contents of register rb in register ra.

OR—Bitwise OR

Syntax: or ra rb rc

Encoding:

1 1 1 0 1	a a a	b b b	c c c	1 1

Operation: $\mathrm{ra} \Leftarrow \mathrm{rb} \wedge \mathrm{rc}$

Performs a bitwise OR of the contents of rb and rc, and stores the result in register ra. The Z flag is set if the result is zero, and cleared otherwise.

TSTBIT—Test bit

Syntax: tstbit ra $k_{4,u}$

Encoding:

1 1 1 1 0	a a a	k k k k	0 0 1 1

Operation: $\mathrm{ra} \wedge 1 \ll k$

Sets the Z flag based on the state of bit k of register ra.

XOR—Bitwise XOR

Syntax: xor ra rb rc

Encoding:

1 1 1 1 0	a a a	b b b	c c c	0 0

Operation: $\mathrm{ra} \Leftarrow \mathrm{rb} \oplus \mathrm{rc}$

Performs a bitwise exclusive OR of the contents of rb and rc, and stores the result in register ra. The Z flag is set if the result is zero, and cleared otherwise.

A.6 Shifts

ASR—Arithmetic shift right

Syntax: asr ra rb

Encoding:

1 1 1 1 1	a a a	b b b	0 0 0 1 0

Operation: $\mathrm{ra} \Leftarrow \mathrm{ra} \ggg \mathrm{rb}$

Shifts the contents of register ra right by the number of bits specified in register rb (modulo 16), filling in the MSBs with whatever was the original state of the

MSB. The result is written back to register ra. The Z and N flags are updated based on the result of the shift.

ASRI—Arithmetic shift right by immediate

Syntax:	asri ra $k_{4,u}$			
Encoding:	1 1 1 1 0	a a a	k k k k	0 0 0 1
Operation:	$\text{ra} \Leftarrow \text{ra} \ggg k$			

Shifts the contents of register ra right by k bits (filling in the MSBs by replicating the original MSB so that the sign is not changed) and stores the result back in register ra. The Z flag is set if the result is zero, and cleared otherwise.

LSL—Shift left

Syntax:	lsl ra rb			
Encoding:	1 1 1 1 1	a a a	b b b	0 1 0 1 0
Operation:	$\text{ra} \Leftarrow \text{ra} \ll \text{rb}_{[3:0]}$			

Shifts the contents of register ra left by the number of bits specified in register rb (modulo 16), filling in the LSBs with zeros. The result is written back to register ra. The Z and N flags are updated based on the result of the shift.

LSR—Logical shift right

Syntax:	lsr ra rb			
Encoding:	1 1 1 1 1	a a a	b b b	1 0 0 1 0
Operation:	$\text{ra} \Leftarrow \text{ra} \gg \text{rb}$			

Shifts the contents of register ra right by the number of bits specified in register rb (modulo 16), filling in the MSBs with zeros. The result is written back to register ra. The Z and N flags are updated based on the result of the shift.

LSRI—Logical shift right by immediate

Syntax:	lsri ra $k_{4,u}$			
Encoding:	1 1 1 1 0	a a a	k k k k	1 0 0 1
Operation:	$\text{ra} \Leftarrow \text{ra} \gg k$			

Shifts the contents of register ra right by k bits (filling in the MSBs with zeros) and stores the result back in register ra. The Z flag is set if the result is zero, and cleared otherwise.

ROLI—Roll left by immediate

Syntax:	roli ra $k_{4,u}$			
Encoding:	1 1 1 1 0	a a a	k k k k	1 1 0 1
Operation:	(see description)			

Rotates the contents of register ra left by k bit. (The k MSBs are moved to the LSBs and all the other bits are shifted left k bits.) The result is written back into ra. (There is no roll right instruction, as right rolls can be accomplished by rolling left by $16 - k$ bits.)

A.7 Multiplication

MAC—Multiply and accumulate

Syntax: mac ra rb

Encoding:

1 1 1 1 1	a a a	b b b	0 0 0 1 1

Operation: $\{\mathrm{mr2}, \mathrm{mr1}, \mathrm{mr0}\} \Leftarrow \mathrm{ra} \times \mathrm{rb} + \{\mathrm{mr2}, \mathrm{mr1}, \mathrm{mr0}\}$

Performs a signed multiplication of the contents of registers ra and rb and adde the results to the three multiplication result registers, mr0, mr1, and mr2.

MACU—Unsigned multiply and accumulate

Syntax: macu ra rb

Encoding:

1 1 1 1 1	a a a	b b b	1 0 0 1 1

Operation: $\{\mathrm{mr2}, \mathrm{mr1}, \mathrm{mr0}\} \Leftarrow \mathrm{ra} \times \mathrm{rb} + \{\mathrm{mr2}, \mathrm{mr1}, \mathrm{mr0}\}$

Performs an unsigned multiplication of the contents of registers ra and rb and adde the results to the three multiplication result registers, mr0, mr1, and mr2.

MUL—Multiply

Syntax: mul ra rb

Encoding:

1 1 1 1 1	a a a	b b b	0 1 1 0 1

Operation: $\{\mathrm{mr2}, \mathrm{mr1}, \mathrm{mr0}\} \Leftarrow \mathrm{ra} \times \mathrm{rb}$

Performs a signed multiplication of the contents of registers ra and rb and stores the results in the multiplication result registers mr0 and mr1. Register mr2 is cleared.

MULU—Unsigned multiply

Syntax: mulu ra rb

Encoding:

1 1 1 1 1	a a a	b b b	1 1 1 0 1

Operation: $\{\mathrm{mr2}, \mathrm{mr1}, \mathrm{mr0}\} \Leftarrow \mathrm{ra} \times \mathrm{rb}$

Performs an unsigned multiplication of the contents of registers ra and rb and stores the results in the multiplication result registers mr0 and mr1. Register mr2 is cleared.

A.8 Loads

LDB—Load byte

Syntax: ldb ra rb

Encoding:

1 1 1 1 1	a a a	b b b	1 1 0 1 0

Operation: $\mathrm{ra} \Leftarrow mem_8[\mathrm{rb}]$

Loads general-purpose register ra with the zero-extended contents of the byte memory location addressed by general-purpose register rb.

LDH—Load high byte with immediate

Syntax: ldh ra $k_{8,u}$

Encoding:

1 0 1 1 0	a a a	k k k k k k k k

Operation: $\text{ra} \Leftarrow \{k, \text{ra}[7:0]\}$

Sets the most significant eight bits of general-purpose register ra to the contstant k. The least significant bits of ra are not affected.

LDI—Load with register offset

Syntax: ldi ra rb rc

Encoding:

1 1 1 0 0	a a a	b b b	c c c	0 0

Operation: $\text{ra} \Leftarrow \mathit{mem}_{16}[\text{ra} + \text{rb}]$

Loads general-purpose register ra with the contents of the memory location addressed by adding the contents of general-purpose registers rb and rc.

LDIB—Load byte with register offset

Syntax: ldib ra rb rc

Encoding:

1 1 1 0 0	a a a	b b b	c c c	0 1

Operation: $\text{ra} \Leftarrow \mathit{mem}_{8}[\text{rb} + \text{rc}]$

Loads general-purpose register ra with the zero-extended contents of the byte memory location addressed by adding the contents of general-purpose registers rb and rc.

LDINC—Load with postincrement

Syntax: ldinc ra rb

Encoding:

1 1 1 1 1	a a a	b b b	1 1 1 1 0

Operation: $\text{ra} \Leftarrow \mathit{mem}_{16}[\text{rb}]$
$\text{rb} \Leftarrow \text{rb} + 2$

Loads general-purpose register ra with the contents of the memory location addressed by general-purpose register rb. The contents of rb are then incremented by two.

LDINCB—Load byte with postincrement

Syntax: ldincb ra rb

Encoding:

1 1 1 1 1	a a a	b b b	1 0 0 0 1

Operation: $\text{ra} \Leftarrow \mathit{mem}_{8}[\text{rb}]$
$\text{rb} \Leftarrow \text{rb} + 1$

Loads general-purpose register ra with the zero-extended contents of the byte memory location addressed by general-purpose register rb. Register rb is then incremented by one.

LDL—Load low byte with immediate

Syntax:	ldl ra $k_{8,s}$
Encoding:	1 0 1 1 1 │ a a a │ k k k k k k k k
Operation:	ra $\Leftarrow$ *sign extend*(k)

Sets the contents of general-purpose register ra to the sign-extended 8-bit constant k.

LDO—Load with immediate offset

Syntax:	ldo ra rb $k_{5,u,ev}$
Encoding:	1 1 0 1 0 │ a a a │ b b b │ k k k k k
Operation:	ra $\Leftarrow mem_{16}[\text{rb} + k]$

Loads general-purpose register ra with the contents of the memory location addressed by adding the 5-bit unsigned constant k to the contents of general-purpose register rb

LDP—Load from peripheral address

Syntax:	ldp ra k_{periph}
Encoding:	1 0 1 0 0 │ a a a │ k k k k k k k k
Operation:	ra $\Leftarrow mem_{16}[k]$

Loads the general-purpose register ra with data read from the peripheral control register at memory address k. The constant k must be even and in the range `0xFE00` through `0xFFFE`. (Only the constant $k -$ `0xFE00` is encoded in the instruction.)

LDSH—Load high register from stack

Syntax:	ldsh rh $k_{9,u,ev}$
Encoding:	1 0 0 0 1 │ h h h │ k k k k k k k k
Operation:	rh $\Leftarrow mem_{16}[\text{sp} + k]$

Loads the special-purpose register rh with the contents of the memory location addressed by adding the 9-bit unsigned, even constant k to the stack pointer (r12).

LDSL—Load low register from stack

Syntax:	ldsl ra $k_{9,u,ev}$
Encoding:	1 0 0 0 0 │ a a a │ k k k k k k k k
Operation:	ra $\Leftarrow mem_{16}[\text{sp} + k]$

Loads general-purpose register ra with the contents of the memory location addressed by adding the 9-bit unsigned, even constant k to the stack pointer (r12).

A.9 Stores

STB—Store byte

Syntax:	stb ra rb
Encoding:	$\boxed{1\ 1\ 1\ 1\ 1}\boxed{a\ a\ a}\boxed{b\ b\ b}\boxed{0\ 0\ 1\ 1\ 0}$
Operation:	$mem_8[\mathrm{rb}] \Leftarrow \mathrm{ra}$

Stores the low eight bits of the contents of ra at the address in register rb.

STI—Store with register offset

Syntax:	sti ra rb rc
Encoding:	$\boxed{1\ 1\ 1\ 0\ 0}\boxed{a\ a\ a}\boxed{b\ b\ b}\boxed{c\ c\ c}\boxed{1\ 0}$
Operation:	$mem_{16}[\mathrm{rb} + \mathrm{rc}] \Leftarrow \mathrm{ra}$

Stores the contents of general-purpose register ra at the memory location addressed by adding the contents of general-purpose registers rb and rc.

STIB—Store byte with register offset

Syntax:	stib ra rb rc
Encoding:	$\boxed{1\ 1\ 1\ 0\ 0}\boxed{a\ a\ a}\boxed{b\ b\ b}\boxed{c\ c\ c}\boxed{1\ 1}$
Operation:	$mem_8[\mathrm{rb} + \mathrm{rc}] \Leftarrow \mathrm{rc}$

Stores the least-significant byte of general-purpose register ra at the byte memory location addressed by adding the contents of general-purpose registers rb and rc.

STINC—Store with postincrement

Syntax:	stinc ra rb
Encoding:	$\boxed{1\ 1\ 1\ 1\ 1}\boxed{a\ a\ a}\boxed{b\ b\ b}\boxed{0\ 0\ 0\ 0\ 1}$
Operation:	$mem_{16}[\mathrm{rb}] \Leftarrow \mathrm{ra}$ $\mathrm{rb} \Leftarrow \mathrm{rb} + 2$

Stores the contents of general-purpose register ra at the memory location addressed by general-purpose register rb. Register rb is then incremented by two.

STINCB—Store byte with postincrement

Syntax:	stincb ra rb
Encoding:	$\boxed{1\ 1\ 1\ 1\ 1}\boxed{a\ a\ a}\boxed{b\ b\ b}\boxed{0\ 1\ 0\ 0\ 1}$
Operation:	$mem_8[\mathrm{rb}] \Leftarrow \mathrm{ra}$ $\mathrm{rb} \Leftarrow \mathrm{rb} + 1$

Stores the least-significant byte of general-purpose register ra at the memory location addressed by general-purpose register rb. Register rb is then incremented by one.

STO—Store with immediate offset

Syntax: sto ra rb $k_{5,u,ev}$
Encoding: $\boxed{1\ 1\ 0\ 1\ 1}\boxed{a\ a\ a}\boxed{b\ b\ b}\boxed{k\ k\ k\ k\ k}$
Operation: $mem_{16}[\text{ra} + k] \Leftarrow \text{rb}$

Stores the contents of general-purpose register ra at the memory location addressed by adding the 5-bit unsigned constant k to the contents of general-purpose register rb.

STP—Store to peripheral address

Syntax: stp ra k_{periph}
Encoding: $\boxed{1\ 0\ 1\ 0\ 1}\boxed{a\ a\ a}\boxed{k\ k\ k\ k\ k\ k\ k\ k}$
Operation: $mem_{16}[k] \Leftarrow \text{ra}$

Stores the contents of general-purpose register ra to the peripheral control register at memory address k. The constant k must be even and in the range `0xFE00` through `0xFFFE`. (Only the constant k − `0xFE00` is encoded in the instruction.)

STSH—Store high register to stack

Syntax: stsh rh $k_{9,u,ev}$
Encoding: $\boxed{1\ 0\ 0\ 1\ 1}\boxed{h\ h\ h}\boxed{k\ k\ k\ k\ k\ k\ k\ k}$
Operation: $mem_{16}[\text{sp} + k] \Leftarrow \text{rh}$

Stores the contents of special-purpose register rh to the memory location addressed by adding the 9-bit unsigned, even constant k to the stack pointer (r12).

STSL—Store low register to stack

Syntax: stsl ra $k_{9,u,ev}$
Encoding: $\boxed{1\ 0\ 0\ 1\ 0}\boxed{a\ a\ a}\boxed{k\ k\ k\ k\ k\ k\ k\ k}$
Operation: $mem_{16}[\text{sp} + k] \Leftarrow \text{ra}$

Stores the contents of general-purpose register ra to the memory location addressed by adding the 9-bit unsigned, even constant k to the stack pointer (r12).

A.10 Other

CEI—Disable interrupts

Syntax: cei
Encoding: $\boxed{1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1}$
Operation: $I \Leftarrow 0$

Clears the I flag, disabling all interrupts.

CEQ—Disable cache

Syntax: ceq
Encoding: $\boxed{1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1}$
Operation: $Q \Leftarrow 0$

Clears the Q flag, disabling the instruction cache.

FLUSH—Flush cache

Syntax:	flush
Encoding:	1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1
Operation:	(see description)

Invalidates the contents of the instruction cache. (Clears all of the valid flags.)

FREEZE—Freeze cache

Syntax:	freeze
Encoding:	1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1
Operation:	$F \Leftarrow 1$

Sets the F, locking the contents of the cache.

J—Jump

Syntax:	j ra
Encoding:	1 1 1 1 1 | a a a | 0 0 0 1 1 1 1 1
Operation:	rpc $\Leftarrow$ ra

Jumps to the location in register ra,

JL—Jump and link

Syntax:	jl ra
Encoding:	1 1 1 1 1 | a a a | 1 0 0 1 1 1 1 1
Operation:	*reglp* $\Leftarrow$ pc + 2
	pc $\Leftarrow$ ra

Stores the location of the next instruction (pc+2) in the linkage pointer register lp, and then jumps to the location in register ra.

MOVHL—Move (low register to high register)

Syntax:	movhl rh rb
Encoding:	1 1 1 1 1 | h h h | b b b | 0 1 1 1 1
Operation:	rh $\Leftarrow$ rb

Copies the contents of register rb into register rh.

MOVLH—Move (high register to low register)

Syntax:	movlh ra rh
Encoding:	1 1 1 1 1 | a a a | h h h | 1 0 1 1 1
Operation:	ra $\Leftarrow$ rh

Copies the contents of register rh into register ra.

MOVLL—Move (low register to low register)

Syntax:	movll ra rb
Encoding:	1 1 1 1 1 | a a a | b b b | 0 0 1 1 1
Operation:	ra $\Leftarrow$ rb

Copies the contents of register rb into register ra.

NOP—Null operation

Syntax: nop

Encoding: $\boxed{1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1}$

Operation: (none)

Does nothing.

RET—Return from subroutine

Syntax: ret

Encoding: $\boxed{1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1}$

Operation: pc $\Leftarrow$ lp

Resumes execution at the address in the linkage pointer register, lp.

RFE—Return from exception

Syntax: rfe

Encoding: $\boxed{1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1}$

Operation: $\mathrm{cc}_{[7:0]} \Leftarrow \mathrm{cc}_{[15:8]}$
pc $\Leftarrow$ erp

Copies the high eight bits of the condition code register (15) into the low eight bits of that register, and then resumes execution at the address stored in

SEI—Enable interrupts

Syntax: sei

Encoding: $\boxed{1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1}$

Operation: $I \Leftarrow 1$

Sets the I flag, enabling interrupts.

SEQ—Enable cache

Syntax: seq

Encoding: $\boxed{1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1}$

Operation: $Q \Leftarrow 1$

Sets the Q flag, enablging the instruction cache.

SWI—Software interrupt

Syntax: swi

Encoding: $\boxed{1\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1}$

Operation: (see description)

Causes an exception to be taken and execution to jump to the address `0x001C`.

SXT—Sign extend

Syntax: sxt ra rb

Encoding:

1 1 1 1 1	a a a	b b b	0 0 1 0 1

Operation: ra $\Leftarrow$ *sign extend*$(\mathrm{rb}_{[7:0]})$

Sign extends the contents of register rb and stores the result in register ra.

THAW—Thaw cache

Syntax: thaw

Encoding: $\boxed{1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1}$

Operation: $F \Leftarrow 0$

Clears the F flag, unlocking the contents of the cache.

WAIT—Wait for interrupt

Syntax: wait

Encoding: $\boxed{1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1}$

Operation: (see description)

Stops the CPU clock until an unmasked interrupt request occurs.

Appendix B

On-Chip Peripherals

B.1 ADC Interface

The ADC interface is designed to control a custom external ADC part, described in [21]. Conversions can be initiated manually (using the RY bit in the interface configuration register) or automatically at a rate programmed by the sample rate register.

Configuration and Status (0xFE00)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	RY	*undefined*						AS	PR	CRD						
Reset	0	–	–	–	–	–	–	0	x	x	x	x	x	x	x	x
Access	rw	–	–	–	–	–	–	rw	rw	rw	rw	rw	rw	rw	rw	rw

CRD—Conversion clock divider (bits 6-0)
Controls the frequency of the clock supplied to the external ADC. The ADC clock is obtained by dividing the system clock by the value in this field plus one. ($f_{adc} = f_{sys}/(CRD + 1)$)

PR—Precision (bit 7)
Selects 8-bit precision when 0 or 12-bit precision when 1

AS—Autosample (bit 8)
When set, conversions are initiated automatically at the rate specified by the sample rate register

RY—Data ready/manual sample trigger (bit 15)
This bit will read one if there is a valid conversion result that has not been read yet waiting in the sample data register. Writing a one to this bit will initiate

a conversion manually. This bit is automatically cleared when the sample data register (0xFE06) is read.

Sample rate register (0xFE02)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

When autosampling is enabled, conversions will be initiated at a rate determined by dividing the system clock by the value in this register.

Offset register (0xFE04)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The value in this register is subtracted from the raw conversion result (e.g. to remove a dc offset).

Sample data register (0xFE06)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Access	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Contains the measurement result from the most recent conversion.

B.2 DMA Engine

The DMA engine supports up to four transfers at once. Each channel is controlled by its own set of of four registers: registers 0xFE20–0xFE26 control channel 0, registers 0xFE28–0xFE2E control channel 1, registers 0xFE30–0xFE36 control channel 2, and registers 0xFE38–0xFE3E control channel 3. Only one channel can be active on any given clock cycle. The channels have a hard-wired priority order, with channel 0 having the highest priority and channel 3 the lowest.

Configuration and Status (0xFE20, 0xFE28, 0xFE30, 0xFE38)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	EN	*undefined*			DBY	SBY	ID	IS	DTR				STR			
Reset	0	–	–	–	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	–	–	–	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

STR—Source trigger (bits 3-0)

Selects the signal used to control when transfers occur, according to the table below.

0	ADC sample ready	8	SPI2 received data ready
1	FIR input ready	9	UART received data ready
2	FIR output ready	10	UART transmitter ready
3	FFT ready for input	11	*undefined*
4	FFT output ready	12	*undefined*
5	SPI1 transmitter ready	13	*undefined*
6	SPI1 received data ready	14	*undefined*
7	SPI2 transmitter ready	15	None (trigger immediately)

DTR—Destination trigger (bits 7-4)

Selects the signal used to control when transfers occur, according to the same table as for the source trigger

IS—Increment source (bit 8)

When this field is one, the source pointer is incremented (by one if the source is byte-wide, by two if the source is word wide) after every transfer.

ID—Increment destination (bit 9)

When this field is one, the destination pointer is incremented (by one if the destination is byte-wide, by two if the destination is word wide) after every transfer.

SBY—Source is byte-wide (bit 10)

When one, the source is byte-wide; when zero, the source is word-wide

DBY—Destination is byte-wide (bit 11)

When one, the destination is byte-wide; when zero, the destination is word-wide

EN—Channel enable (bit 15)

When set, the channel is enabled. A transfer can be paused by clearing this bit while the transfer is in progress. The transfer will resume when the bit is set again.

Count (`0xFE22`, `0xFE2A`, `0xFE32`, `0xFE3A`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Number of values to transfer. (When read, gives the number of values left to transfer.)

Source Pointer (`0xFE24`, `0xFE2C`, `0xFE34`, `0xFE3C`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Source address for the transfer

Destination Pointer (`0xFE26`, `0xFE2E`, `0xFE36`, `0xFE3E`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Destination address for the transfer

B.3 FFT Accelerator

The FFT accelerator performs 128-, 256-, 512-, and 1024-point real-value Fourier transforms, and 64-, 128-, 256-, and 512-point complex Fourier transforms. The configuration register must be written to initiate a transform computation. Then the input data values are written in order to the data register. The transform computation begins automatically once the last input value is loaded. When the transform is complete (stage 3, as reported by the status register), the output data is read from the data register.

Configuration (`0xFE40`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	undefined													CPX	LEN	
Reset	–	–	–	–	–	–	–	–	–	–	–	–	–	0	0	0
Access	–	–	–	–	–	–	–	–	–	–	–	–	–	w	w	w

LEN—Transform Length (bits 1-0)

Sets the number of in the transform points, according to the following table

Real	0	128 points	1	256 points	2	512 points	3	1024 points
Complex	0	64 points	1	128 points	2	256 points	3	512 points

CPX—Transform type (bit 2)

Sets the type of transform

Status (0xFE40)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	undefined				OVF										STG	
Reset	–	–	–	–	0	0	0	0	0	0	0	0	0	0	0	0
Access	–	–	–	–	r	r	r	r	r	r	r	r	r	r	r	r

STG—Stage (bits 1-0)

Indicates progress of the transform

0	Idle
1	Reading input data
2	Computing transform
3	Output data ready

OVF—Overflow flags (bits 11-2)

Indicates whether overflow occurred on each of the butterfly stages. The final output values are divided by two for every bit that is set in this field.

Data (0xFE42)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Input data for the transform is written to this register. When the transform is complete, the results are read from this register.

B.4 FIR Accelerator

Configuration and Status (0xFE60)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	OY	IY	*undefined*								TYP		–	NTPS		
Reset	0	0	–	–	–	–	–	–	–	–	0	0	–	0	0	0
Access	r	r	–	–	–	–	–	–	–	–	rw	rw	–	rw	rw	rw

NTPS—Number of taps (bits 2-0)

TYP—Filter type (bits 5-4)

Sets the type of filter to implement

0 Asymmetric 1 Symmetric 2 *undefined* 3 Anti-symmetric

IY—Input ready (bit 14)

Indicates that the accelerator is ready for a new input sample to be written to the sample data register.

OY—Output ready (bit 15)

Indicates that an output sample is ready to be read from the sample data register. Reading from the sample data register automatically clears this flag.

Scaling and Decimation (`0xFE62`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	undefined			DECI					*undefined*			SCALE				
Reset	–	–	–	0	0	0	0	0	–	–	–	0	0	0	0	0
Access	–	–	–	rw	rw	rw	rw	rw	–	–	–	rw	rw	rw	rw	rw

SCALE—Scale factor (bits 4-0)

The data values read from the sample data register are formed by shifting the contents of the internal 25-bit accumulator right by the number of bits specified by this field.

DECI—Decimation factor (bits 12-8)

Sets the number of input samples that must be written between each output sample computation. When zero, an output sample is computed for every input sample; when one, an output sample is computed for every other input sample, etc.

Coefficient (`0xFE64`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Coefficient															
Reset	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Access	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Sample Data (`0xFE66`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

B.5 General-Purpose I/O

Sixteen pins are available for general-purpose I/O. Each pin can be individually configured as either an input or an output. Each pin can be configured as a level-sensitive interrupt request line. (All GPIO interrupts share a single interrupt vector, however.) GPIO pins can be configured to generate timer input capture events, or to be controlled by timer output compare events.

Input/Output (`0xFE80`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	GPIO state															
Reset	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Direction (0xFE82)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Direction control															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

IRQ (0xFE84)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Setting a bit in this register configures the corresponding GPIO pin as a level sensitive interrupt request signal.

Sense (0xFE86)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

For any GPIO pin configured as an interrupt request signal, determines whether the request is active low (the corresponding bit in this register is clear) or active high (the corresponding bit is set).

OC1 Mask (0xFE88)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

If the timer OC1 signal is set, GPIO outputs are driven with the bitwise OR of this register and the GPIO output register.

OC2 Mask (0xFE90)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

If the timer OC2 signal is set, GPIO outputs are driven with the bitwise OR of this register and the GPIO output register.

IC1 Mask (0xFE92)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The IC1 signal to the timer module is formed by bitwise ANDing this register with the state of the GPIO pins, and then ORing all of the bits together.

IC2 Mask (0xFE94)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The IC2 signal to the timer module is formed by bitwise ANDing this register with the state of the GPIO pins, and then ORing all of the bits together.

B.6 Power Management Unit

Interrupt Mask (0xFEA0)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	undefined				UTX	URX	SP2	SP1	FIO	FII	ADC	FFT	RTC	GIO	TIM	DMA
Reset	–	–	–	–	0	0	0	0	0	0	0	0	0	0	0	0
Access	–	–	–	–	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

DMA—Enable DMA interrupts (bit 0)

TIM—Enable timer interrupts (bit 1)

GIO—Enable GPIO interrupts (bit 2)

RTC—Enable RTC interrupts (bit 3)

FFT—Enable FFT interrupts (bit 4)

ADC—Enable ADC interrupts (bit 5)

FII—Enable FIR input interrupts (bit 6)

FIO—Enable FIR output interrupts (bit 7)

SP1—Enable SPI1 interrupts (bit 8)

SP2—Enable SPI2 interrupts (bit 9)

URX—Enable UART receive interrupts (bit 10)

UTX—Enable UART transmit interrupts (bit 11)

Wakeup Mask (0xFEA2)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	BOOT		WS	–	WAKMSK											
Reset	*	*	0	–	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	r	–	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

WAKMSK—Wakeup mask (bits 11-0)
Determines which interrupt sources can wake the CPU from power-down.

WS—Warmstart flag (bit 13)
This flag is cleared by a chip-level reset, and is set whenever the CPU is powered down.

BOOT—Boot mode (bits 15-14)
This field contains the state of the boot mode pins on reset.

Clock Enable (0xFEA4)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	undefined			FIRRS	FFTRS	FIREN	FFTEN	MEM								
Reset	–	–	–	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	–	–	–	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

MEM—Memory bank enables (bits 8-0)

FFTEN—FFT enable (bit 9)

FIREN—FIR enable (bit 10)

FFTRS—FFT reset (bit 11)

FIRRS—FIR reset (bit 12)

Power Enable (0xFEA6)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CPUP	*undefined*				FIRP	FFTP	MEM								
Reset	0	–	–	–	–	0	0	0	0	0	0	0	0	0	0	0
Access	rw	–	–	–	–	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

MEM—Memory power enable (bits 8-0)

FFTP—FFT power enable (bit 9)

FIRP—FIR power enable (bit 10)

CPUP—CPU power enable (bit 15)

B.7 Real-Time Clock

The real-time clock (RTC) consists of a 32-bit counter clocked at 32.768 kHz, and a 32-bit compare register.

Configuration (`0xFEC0`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	I	*undefined*														E
Reset	0	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0
Access	r	–	–	–	–	–	–	–	–	–	–	–	–	–	–	rw

E—Interrupt request enable (bit 0)

I—Interrupt request flag (bit 15)

Temp (`0xFEC2`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Used store/supply the upper 16 bits during reads/writes of the 32-bit counter and compare registers.

Counter (`0xFEC4`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Counter Register															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Compare (`0xFEC6`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Compare Register															
Reset	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

B.8 SPI Ports

Configuration (0xFEE0, 0xFF00)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	SCRATCH					PD	PWR	PHS	POL	RATE						
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

RATE—Clock Rate (bits 6-0)

POL—Clock Polarity (bit 7)

PHS—Clock Phase (bit 8)

PWR—Power Enable (bit 9)

PD—Pullup Enable (bit 10)
For MISO

SCRATCH—Scratch pad (bits 15-11)

Status (0xFEE2, 0xFF02)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	undefined													RF	TF	I
Reset	–	–	–	–	–	–	–	–	–	–	–	–	–	0	0	0
Access	–	–	–	–	–	–	–	–	–	–	–	–	–	r	r	r

I—Idle (bit 0)

TF—Transmitter FIFO full (bit 1)

RF—Receiver FIFO full (bit 2)

Data (0xFEE4, 0xFF04)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	undefined								Data							
Reset	–	–	–	–	–	–	–	–	x	x	x	x	x	x	x	x
Access	–	–	–	–	–	–	–	–	rw	rw	rw	rw	rw	rw	rw	rw

Data—Data (bits 7-0)

Chip Select (`0xFEE6`, `0xFF06`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	undefined														PW	CS
Reset	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	0
Access	–	–	–	–	–	–	–	–	–	–	–	–	–	–	w	rw

CS—Chip Select (bit 0)

PW—Power Enable (bit 1)

B.9 Timer

The timer consists of a free-running counter, two output compare registers, and two input capture registers. The counter, compare, and capture registers are all 32 bits wide, although only the low 16 bits of each register are directly accessible. Whenever one of these registers is read, the high 16 bits of the register are copied into a temporary register (0xFF22) where they can be read out later. Similarly, during a write to any of these 32-bit registers, the contents of the temporary register are copied into the high 16 bits of the register being written. In C programs, these registers can be accessed directly as though they were 32 bits wide: the compiler automatically generates the necessary reads or writes of the temporary register.

When the RUN bit in the configuration register is set, the counter increments on every system clock cycle. The counter can be read or written to at any time. When the counter overflows from 0xFFFFFFFF to 0x00000000, the OVF flag is set in the status register, which may be configured to generate an interrupt request.

When the counter register is equal to one of the output compare registers, an output compare event occurs, and the corresponding OCx flag in the status register is set (which may cause an interrupt request). Output compare events can be configured to automatically toggle GPIO pins: see the GPIO module documentation for details.

When an input capture event occurs (as defined in the GPIO module), the current contents of the counter register are copied into the corresponding input capture register, and the the corresponding ICx flag is set in the status register (which may cause an interrupt request).

Configuration (0xFF20)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CO1	SO2	CO1	SO1	*undefined*		RUN	OVM	IC2M		IC1M		OC2M		OC1M	
Reset	0	0	0	0	–	–	0	0	0	0	0	0	0	0	0	0
Access	w	w	w	w	–	–	w	w	w	w	w	w	w	w	w	w

OC1M—OC1 Mode (bits 1-0)

Determines what happens when an output compare 1 event occurs

0	Nothing happens
1	An interrupt is requested
2	An interrupt is requested and the OC1 controlled GPIO pins are set
3	An interrupt is requested and the OC1 controlled GPIO pins are cleared

OC2M—OC2 Mode (bits 3-2)

Determines what happens when an output compare 2 event occures

0	Nothing happens
1	An interrupt is requested
2	An interrupt is requested and the OC2 controlled GPIO pins are set
3	An interrupt is requested and the OC2 controlled GPIO pins are cleared

IC1M—IC1 Mode (bits 5-4)

Determines what kind of transitions on the IC1 GPIO pins will cause an IC1 event

0	IC1 events are disabled
1	A falling edge will cause an IC1 event
2	A rising edge will cause an IC1 event
3	Any edge will cause an IC1 event

IC2M—IC2 Mode (bits 7-6)

Determines what kind of transitions on the IC2 GPIO pins will cause an IC2 event

0	IC2 events are disabled
1	A falling edge will cause an IC2 event
2	A rising edge will cause an IC2 event
3	Any edge will cause an IC2 event

OVM—Overflow Mode (bit 8)

When set, the OVF flag will cause an interrupt request. When clear, the OVF flag can still be polled, but will not generate interrupt requests.

RUN—Main counter enable (bit 9)

When set, the counter register increments on every system clock cycle. When clear, the counter register is disabled to save power.

SO1—Set OC1 pins (bit 12)

Writing a 1 to this bit will cause the OC1 controlled GPIO pins to be set

CO1—Clear OC1 pins (bit 13)

Writing a 1 to this bit will cause the OC1 controlled GPIO pins to be cleared

SO2—Set OC2 pins (bit 14)
: Writing a 1 to this bit will cause the OC2 controlled GPIO pins to be set

CO1—Clear OC1 pins (bit 15)
: Writing a 1 to this bit will cause the OC2 controlled GPIO pins to be cleared

Status (0xFF20)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	IC2	IC1	OC2	OC1				*undefined*				OVF	IC2F	IC1F	OC2F	OC1F
Reset	x	x	0	0	–	–	–	–	–	–	–	0	0	0	0	0
Access	r	r	r	r	–	–	–	–	–	–	–	r	r	r	r	r

OC1F—OC1 Interrupt Flag (bit 0)
: When set, indicates an OC1 event has occurred. Automatically cleared when the status register is read.

OC2F—OC2 Interrupt Flag (bit 1)
: When set, indicates an OC2 event has occurred. Automatically cleared when the status register is read.

IC1F—IC1 Interrupt Flag (bit 2)
: When set, indicates an IC1 event has occurred. Automatically cleared when the status register is read.

IC2F—IC2 Interrupt Flag (bit 3)
: When set, indicates an IC2 event has occurred. Automatically cleared when the status register is read.

OVF—Overflow Flag (bit 4)
: When set, indicates the counter register has overflowed from 0xFFFFFFFF to 0x00000000. Automatically cleared when the status register is read.

OC1—OC1 State (bit 12)
: Indicates the state of the OC1 controlled GPIO pins

OC2—OC2 State (bit 13)
: Indicates the state of the OC2 controlled GPIO pins

IC1—IC1 State (bit 14)
: Indicates the state of the IC1 GPIO pins

IC2—IC2 State (bit 15)
: Indicates the state of the IC1 GPIO pins

Temp (0xFF22)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Temporary Register

Counter (`0xFFF4`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Main Counter

Output Compare 1 (`0xFFF6`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Output Compare 1 Match Register

Output Compare 2 (`0xFFF8`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Output Compare 2 Match Register

Input Compare 1 (`0xFFFA`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Input Compare 1 Match Register

Input Compare 2 (`0xFFFC`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Input Compare 2 Match Register

B.10 UART

Configuration and Status (`0xFF40`)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	undefined											RR	TR	TB	TE	RE
Reset	–	–	–	–	–	–	–	–	–	–	–	x	x	x	0	0
Access	–	–	–	–	–	–	–	–	–	–	–	r	r	r	rw	rw

RE—Receiver enable (bit 0)

TE—Transmitter enable (bit 1)

TB—Transmitter busy (bit 2)

TR—Receiver ready (bit 3)
When high, data is ready to be read from the receive data register.

RR—Transmitter ready (bit 4)
When this flag is high, data can be written into the transmit data register.

Baud Rate (0xFF42)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Data (0xFF44)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	undefined															
Reset	–	–	–	–	–	–	–	–	x	x	x	x	x	x	x	x
Access	–	–	–	–	–	–	–	–	rw	rw	rw	rw	rw	rw	rw	rw

Data Register (bits 7-0)

Bibliography

[1] D. Estrin, L. Girod, G. Pottie, and M. Srivastava, "Instrumenting the world with wireless sensor networks," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 4, 2001, pp. 2033–2036.

[2] K. Bult, A. Burstein, D. Chang, M. Dong, M. Fielding, E. Kruglick, J. Ho, F. Lin, T. Lin, W. Kaiser, H. Marcy, R. Mukai, P. Nelson, F. Newburg, K. Pister, G. Pottie, H. Sanchez, O. Stafsudd, K. Tan, S. Xue, and J. Yao, "Low power systems for wireless microsensors," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 1996, pp. 17–21.

[3] "Panasonic CR2477 datasheet." [Online]. Available: http://www.panasonic.com/industrial/battery/oem/

[4] V. Raghunathan, A. Kansal, J. Hsu, J. Friedman, and M. Srivastava, "Design considerations for solar energy harvesting wireless embedded systems," in *Information Processing in Sensor Networks (IPSN)*, 2005.

[5] "Perpetuum PMG17 datasheet." [Online]. Available: http://www.perpetuum.co.uk

[6] R. Guigon, J.-J. Chaillout, T. Jager, and G. Despesse, "Harvesting raindrop energy: Experimental study," *Smart Materials and Structures*, vol. 17, no. 1, p. 015039 (6pp), 2008.

[7] V. L. Tom, Torfs, P. Fiorini, and C. V. Hoof, "Thermoelectric converters of human warmth for self-powered wireless sensor nodes," *IEEE Sensors Journal*, vol. 7, no. 5, pp. 650–657, May 2007.

[8] R. Min, M. Bhardwaj, S.-H. Cho, N. Ickes, E. Shih, A. Sinha, A. Wang, and A. Chandrakasan, "Energy-centric enabling technologies for wireless sensor networks," *IEEE Wireless Communications*, vol. 9, no. 4, pp. 28–39, August 2002.

[9] R. Min, M. Bhardwaj, N. Ickes, A. Wang, and A. Chandrakasan, "The hardware and the network: Total-system strategies for power aware wireless microsensors," in *IEEE CAS Workshop on Wireless Communications and Networking*, 2002.

[10] J. Hill and D. Culler, "A wireless embedded sensor architecture for system-level optimization," *UC Berkeley Technical Report*, 2002.

[11] B. A. Warneke and K. S. Pister, "An ultra-low energy microcontroller for smart dust wireless sensor networks," in *International Solid-State Circuits Conference (ISSCC)*, February 2004.

[12] V. Ekanayake, I. Clinton Kelly, and R. Manohar, "An ultra low-power processor for sensor networks," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004, pp. 27–36.

[13] L. Nazhandali, B. Zhai, J. Olson, A. Reeves, M. Minuth, R. Helfand, S. Pant, T. Austin, and D. Blaauw, "Energy optimization of subthreshold-voltage sensor network processors," in *International Symposium on Computer Architecture (ISCA)*, 2005, pp. 197–207.

[14] M. Hempstead, N. Tripathi, P. Mauro, G.-Y. Wei, and D. Brooks, "An ultra low power system architecture for sensor network applications," in *International Symposium on Computer Architecture (ISCA)*, 2005, pp. 208–219.

[15] M. Bajura, B. Schott, J. Flidr, J. Czarnaski, C. Worth, T. Tho, and L. Wang, "An integrated, modular, power-aware microsensor architecture and application to unattended acoustic vehicle tracking," *International Society for Optical Engineering (SPIE)*, 2005.

[16] L. Nachman, R. Kling, R. Adler, J. Huang, and V. Hummel, "The Intel mote platform: a bluetooth-based sensor network for industrial monitoring," in *Information Processing in Sensor Networks (IPSN)*, 2005, p. 61.

[17] R. Adler, M. Flanigan, J. Huang, R. Kling, N. Kushalnagar, L. Nachman, C.-Y. Wan, and M. Yarvis, "Intel Mote 2: An advanced platform for demanding sensor network applications," in *Embedded Networked Sensor Systems (SenSys)*, 2005, pp. 298–298.

[18] "Stargate datasheet." [Online]. Available: http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Stargate_Datasheet.pdf

[19] R. Riley, B. Schott, J. Czarnaski, and S. Thakkar, "Power-aware acoustic processing," in *Information Processing in Sensor Networks (IPSN)*, 2003, pp. 566–581.

[20] M. Stanacevic and G. Cauwenberghs, "Micropower mixed-signal acoustic localizer," in *European Solid-State Circuits Conference (ESSCIRC)*, 2003, pp. 69–72.

[21] N. Verma and A. P. Chandrakasan, "A 25μw 100ks/s 12b ADC for wireless microsensor applications," in *International Solid-State Circuits Conference (ISSCC)*, 2006.

[22] B. Zhai, L. Nazhandali, J. Olson, A. Reeves, M. Minuth, R. Helfand, S. Pant, D. Blaauw, and T. Austin, "A 2.60pJ/inst subthreshold sensor processor for optimal energy efficiency," in *VLSI Circuits*, 2006, pp. 154–155.

[23] J. Kwong, Y. Ramadass, N. Verma, M. Koesler, K. Huber, H. Moormann, and A. Chandrakasan, "A 65nm sub-Vt microcontroller with integrated sram and switched-capacitor dc-dc converter," in *International Solid-State Circuits Conference (ISSCC)*, 2008, pp. 318–319.

[24] S. Mutoh, T. Douskei, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada, "1-V power supply high-speed digital circuit technology with multithreshold voltage cmos," *IEEE Journal of Solid-state Circuits*, pp. 847–854, August 1995.

[25] J. Kao, A. Chandrakasan, and D. Antoniadis, "Transistor sizing issues and tool for multi-threshold cmos technology," in *Design Automation Conference (DAC)*, 1997, pp. 409–414.

[26] S. Shigematsu, S. Mutoh, Y. Matsuya, Y. Tanabe, and J. Yamada, "A 1-V high-speed MTCMOS circuit scheme for power-down application circuits," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 6, pp. 861–869, June 1997.

[27] H. Mizuno and T. Kawahara, "ChipOS: Open power-management platform to overcome the power crisis in future lsis," in *International Solid-State Circuits Conference (ISSCC)*, 2001, pp. 344–345, 463.

[28] M. Anis, S. Areibi, and M. Elmasry, "Design and optimization of multithreshold CMOS (MTCMOS) circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 10, October 2003.

[29] F. Li, Y. Lin, and L. He, "Fpga power reduction using configurable dual-Vdd," in *Design Automation Conference (DAC)*, 2004, pp. 735–740.

[30] Y. Cheng and C. Hu, *MOSFET Modeling & BSIM3 User's Guide.* Kluwer Academic Publishers, 1999.

[31] C. K. IV, V. Ekanayake, and R. Manohar, "SNAP: A sensor-network asynchronous processor," in *Asynchronous Circuits and Systems*, 2003, pp. 24–33.

[32] L. Nazhandali, M. Minuth, B. Zhai, J. Olson, T. Austin, and D. Blaauw, "A second-generation sensor network processor with application-driven memory optimizations and out-of-order execution," in *Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2005, pp. 249–256.

[33] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: scalable sensor network simulation with precise timing," in *Information Processing in Sensor Networks (IPSN)*, 2005, pp. 477–482.

[34] A. Sinha and A. P. Chandrakasan, "JouleTrack: a web based tool for software energy profiling," in *Design Automation Conference (DAC)*, 2001, pp. 220–225.

[35] R. A. Ravindran, R. M. Senger, E. D. Marsman, G. S. Dasika, M. R. Guthaus, S. A. Mahlke, and R. B. Brown, "Increasing the number of effective registers in a low-power processor using a windowed register file," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2003, pp. 125–136.

[36] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *International Symposium on Microarchitecture*, 1997, pp. 184–193.

[37] A. Gordon-Ross, S. Cotterell, and F. Vahid, "Exploiting fixed programs in embedded systems: A loop cache example," *IEEE Computer Architecture Letters*, vol. 1, no. 1, 2002.

[38] S. Cotterell and F. Vahid, "Tuning of loop cache architectures to programs in embedded system design," in *International Symposium on System Synthesis*, 2002, pp. 8–13.

[39] L. H. Lee, B. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 1999, pp. 267–269.

[40] ——, "Low-cost embedded program loop caching—revisited," University of Michigan, Tech. Rep. CSE-TR-411-99, 1999.

[41] R. A. Ravindran, P. D. Nagarkar, G. S. Dasika, E. D. Marsman, R. M. Senger, S. A.Mahlke, , and R. B. Brown, "Compiler managed dynamic instruction placement in a low-power code cache," in *International Symposium on Code Generation and Optimization*, March 2005, pp. 179–190.

[42] P. Jain, S. Devadas, D. Engels, and L. Rudolph, "Software-assisted cache replacement mechanisms for embedded systems," in *International Conference on Computer Aided Design (ICCAD)*, November 2001, pp. 119–126.

[43] D. J. Wheeler and R. M. Needham, "TEA, a tiny encryption algorithm," in *Internationa Workshop on Fast Software Encryption*, 1994, pp. 363–366.

[44] J. B. Rothman and A. J. Smith, "Sector cache design and performance," Computer Science Division (EECS), University of California, Berkeley, Tech. Rep. UCB/CSD-99-1034, 1999. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1999/CSD-99-1034.pdf

[45] L. A. Belady, "A study of replacement algorithms for virtual storage computers," vol. 5, no. 2, pp. 78–101, 1966.

[46] S. Hosseini-Khayat, "On optimal replacement of nonuniform cache objects," *IEEE Transactions on Computers*, vol. 49, no. 8, pp. 769–778, 2000.

[47] M. Brehob, S. Wagner, E. Torng, and R. Enbody, "Optimal replacement is NP-hard for nonstandard caches," *IEEE Transactions on Computers*, vol. 53, no. 1, pp. 73–76, 2004.

[48] L. Benini, L. Macchiarulo, A. Macii, and M. Poncino, "Layout-driven memory synthesis for embedded systems-on-chip," *IEEE Transactions on Very Large Scale Integration Systems (VLSI)*, vol. 10, no. 2, pp. 96–105, April 2001.

[49] R. Haukilahti, "Energy characterization of a RTOS hardware accelerator for SoCs," in *Swedish System-on-Chip Conference*, Falkenberg, Sweden, March 2002.

[50] J. Goodman and A. P. Chandrakasan, "Low power scalable encryption for wireless systems," *Wireless Networks*, vol. 4, no. 1, pp. 55–70, 1998.

[51] H. Kim, S. Choi, and H.-J. Yoo, "A low power 16-bit RISC with lossless compression accelerator for body sensor network system," in *Asian Solid-State Circuits Conference (ASSCC)*, 2006, pp. 207–210.

[52] B. Mathew, A. Davis, and Z. Fang, "A low-power accelerator for the sphinx 3 speech recognition system," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2003, pp. 210–219.

[53] A. Hodjat and I. Verbauwhede, "Interfacing a high speed crypto accelerator to an embedded CPU," in *Asilomar Conference on Signals, Systems and Computers (ACSSC)*, vol. 1, 2004, pp. 488–492.

[54] *MSP430x1xx Family userGuide.* Texas Instruments Inc., February 2006.

[55] K. Venkat, "Efficient multiplication and division using MSP430," Application Note SLAA329, 2006. [Online]. Available: http://focus.ti.com/lit/an/slaa329/slaa329.pdf

[56] S. Mutoh, S. Shigematsu, Y. Matsuya, H. Fukuda, , H. Fukuda, and J. Yamada, "A 1-V multithreshold-voltage CMOS digital signal processor for mobile phone application," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 11, pp. 1795–1802, November 1996.

[57] H. Sorensen, D. Jones, M. Heideman, and C. S. Burrus, "Real-valued fast fourier transform algorithms," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 35, no. 6, pp. 849–863, 1987.

[58] A. Wang, "An ultra-low voltage fft processor using energy-aware techniques," Ph.D. dissertation, Massachusetts Institute of Technology, 2003.

[59] B. H. Calhoun and A. Chandrakasan, "A 256kb sub-threshold SRAM in 65nm CMOS," in *International Solid-State Circuits Conference (ISSCC)*, 2006, pp. 2592–2601.

[60] N. Verma and A. P. Chandrakasan, "A 65nm 8T sub-Vt SRAM employing sense-amplifier redundancy," in *International Solid-State Circuits Conference (ISSCC)*, 2007, pp. 328–606.

[61] B. Zhai, D. Blaauw, D. Sylvester, and S. Hanson, "A sub-200mV 6T SRAM in 0.13μm CMOS," in *International Solid-State Circuits Conference (ISSCC)*, 2007, pp. 332–606.

[62] T.-H. Kim, J. Liu, J. Keane, and C. H. Kim, "A high-density subthreshold SRAM with data-independent bitline leakage and virtual ground replica scheme," in *International Solid-State Circuits Conference (ISSCC)*, 2007, pp. 330–606.

[63] I. J. Chang, J.-J. Kim, S. P. Park, and K. Roy, "A 32kB 10T subthreshold SRAM array with bit-interleaving and differential read scheme in 90nm CMOS," in *International Solid-State Circuits Conference (ISSCC)*, 2008, pp. 388–389,622.