# Exploring Instruction Level Energy Efficiency

Terje Runde and Stian Hvatum
*TDT4501 - Computer Science, Specialization Project*
*Department of Computer Science, NTNU*

{terjr,hvatum}@stud.ntnu.no

***Abstract*—Modern microprocessors are limited by power density and new designs must emphasize energy efficiency to become successful. Building energy efficient hardware requires better understanding of how the ISA and its implementation relates to energy efficiency. This paper investigates the instruction level energy efficiency of an ARM Cortex-A9 and presents current drain and pipeline utilization for different instructions. These numbers reveal information about how instructions are executed and how energy efficient their implementations are in this proprietary architecture.**

**The testbench used in the experiments relies on accurate energy measurements. This is achieved by measuring voltage drop over a shunt resistor placed between the CPU core voltage supply pins and an external power supply. Memory usage is avoided by exploiting instruction buffers and omitting loads/stores. This renders the memory hierarchy unused, isolating the core as much as possible.**

**The results shows that the ARM ISA is well balanced. Commonly used instructions such as `add`, `sub` and `mul` seems to be energy efficient. Unfortunately, the ISA suffers from an inefficient handling of conditional execution and status flag updates. Such instructions seems to force synchronization, which then leads to inefficient utilization of the otherwise computationally strong core.**

***Index Terms*—Energy efficiency, microprocessor chips, performance analysis**

## I. Introduction

MAKING processors burn less energy while at the same time increasing performance is currently one of the greatest challenges hardware designers are facing. Performance alone can be improved by cramming more components onto integrated circuits [1] utilizing new process technologies. However, due to the end of Dennard scaling [2], power density on chip will increase linearly with transistor count. Computer designers are forced to employ novel techniques mitigating this issue, such as shutting down parts of the chip [3] and designing specialized hardware.

As processors grow more sophisticated, it becomes harder to reason about their energy efficiency. Even RISC processors, which traditionally were designed to be simple[4], have seen a steep increase in complexity during the last decade[5]. Features that previously only existed in CISC processors are now entering the RISC domain; more complex operations are done per clock cycle. Current RISC designs may have deep pipelines, increased component complexity, advanced branch predictor units and a high degree of instruction level parallelism. They include features that aims to reduce energy consumption and increase throughput in return for added complexity. Moreover, processors are increasingly designed to integrate seamlessly with external components such as accelerators and the memory system.

The need for energy efficient processors is increasing. To better understand how execution of different instructions contribute to energy consumption we propose a method to measure energy efficiency at the instruction level of a processor. Being able to model and monitor energy consumption has recently got the industry's attention: Semiconductor companies are making software and hardware targeting the embedded market providing a monitor for energy consumption on-chip, giving application engineers the opportunity to optimize for energy.

With the emergence of performance counters, it is now possible to detect how different pipeline stages affects the overall power consumption. Previous research correlates power drain seen from the wall outlet with performance counters on the CPU[6]–[8]. Others look at energy usage under different workloads[9].

In this paper, we analyze the instruction level energy efficiency of a modern RISC architecture. We isolate as many architecture components as possible by correlating performance counters with observed current drain for specific instructions. We also show that it is feasible to get a per instruction energy overview of an existing architecture by understanding the hardware and writing benchmark programs. We look at simple single-cycle instructions as well as the most complex instructions using multiple cycles on our target processor. When each relevant instruction has been measured, we compare their normalized energy consumption and discuss properties of different instructions.

This work is motivated by and related to the SHMAC project[10]–[12] at NTNU. The goal of the SHMAC project is to build an energy efficient heterogeneous many-core computer architecture: multiple processing cores with different capabilities share a single ISA and can be put on the same die to create a processor tailored for a specific application. Exploring instruction level energy efficiency gives us a better view on how different ISA implementations perform at different tasks[13]. This information can be useful in the design phase of novel computer architectures such as SHMAC.

Another use for this kind of information is that compilers can optimize code for energy efficiency and not only performance. Further, it can enable simulators to estimate energy consumption of a given program and even compare energy efficiency on different cores, as shown to be effective in [13].

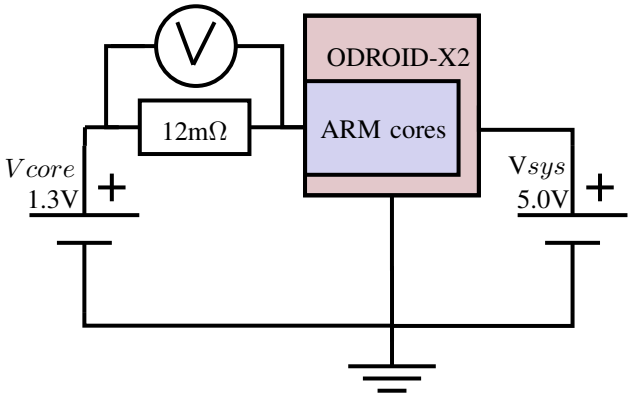


Fig. 1: Experiment setup

| | |
|---|---|
| Manufacturer | Hardkernel |
| Platform | ODROID-X2 |
| SoC | Samsung Exynos 4412 "Prime" |
| CPU Core | ARM Cortex-A9 (r3p0) |
| Number of Cores | 4 |
| Clock Freq. | 1.7 GHz |
| Core Voltage | $1.3V$ |
| OS | Debian GNU/Linux Testing ("jessie") |
| Kernel | Linux 3.8 (custom) |
| Voltmeter | Agilent 34410A |
| Power Supply | Agilent E3631A |
| Shunt Resistor | Thermovolt AB 5697 0002 12mΩ |

TABLE I: System specifications

## II. Methodology

### A. Test Environment

In our experiments, we are using the ODROID-X2 [14] developer platform, which has an Exynos 4412 "Prime" System-on-Chip with four ARM Cortex-A9 processor cores. We disable three of the cores through *sysfs*, leaving only one core available to the scheduler. The processor runs at a fixed frequency of 1.7 GHz. The test environment is sketched in Figure 1 and the details are summarized in Table I.

The Cortex-A9 is a 32-bit out-of-order dual-issue speculative RISC processor, and even though its primary use is in mobile and embedded applications, it shares many features with current desktop processors [15], [16]. It can issue two instructions per cycle and branches its pipeline into four lanes, as depicted in Figure 2. Most instructions can execute in either of the two general ALU's, but multiply instructions must execute in the ALU with a hardware multiplier. The processor core also has separate units for floating point operations (the NEON co-processor) and address manipulation, but these will not be further considered in this paper.

In general, energy consumption of a processor varies with respect to the workload; the harder it has to work, the more energy it uses. In this paper, we seek to achieve the highest possible ALU throughput the processor can offer. To accomplish this, we are required to gain knowledge of the pipeline and other components within the CPU.

Official documentation of the pipeline structure is limited to the "Cortex-A9 Technical Reference Manual" [18] and "The ARM Cortex-A9 Processors" whitepaper [17]. However, by running some architectural experiments and consulting the performance counters we are able to infer some details.

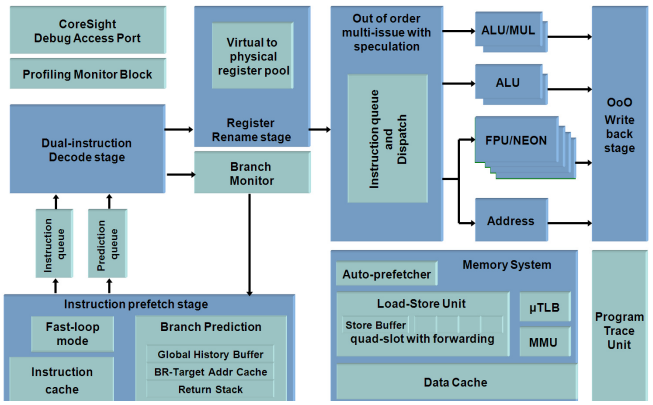


Fig. 2: ARM Cortex-A9 pipeline and peripherals[17]

### B. Architectural Experiments

The A9 processor has 58 distinct events[1] that each can be mapped to one of six generic event counters in the Performance Monitor Unit (i.e. only six generic events can be tracked simultaneously). It also has a separate cycle counter. By comparing execution unit counters for the two ALUs and the cycle counter, we obtain detailed statistics about the pipeline activity. For example, we run `add` instructions with and without hazards, and verify that the core is able to share the work between execution units in the latter case. Note that these performance counters are approximate due to the speculative core, and only serves as a guideline and sanity check for our assumptions.

Using performance counters as above, we are able to confirm a feature on the A9 processor that is very vaguely documented; fast-loop mode. As the name suggests, this feature enables rapid execution of small loops. It does so by fetching instructions from the instruction cache only at the first loop iteration, effectively voiding time and energy spent on instruction lookups between iterations. However, which loops that falls into this category is not documented, but by using performance counters we are able to determine this with confidence. We disable the L1 cache, penalizing runs that do not fit in fast-loop, making it easy to distinguish between runs within and outside fast-loop. We find that for a loop to be executed in fast-loop it must hold one subtle property: the loop in its entirety must fit within the first 60 bytes of a 64 byte cache line. Consequently, the loop body must be 13 instructions or less (15 including `sub` and `bne`). Loops without this property will cause code to be executed outside fast-loop and get a significant decrease in performance.

Furthermore, executing code within fast-loop limits the number of cache mispredicts to two, independent of the iteration count. We confirm this by looking at the cache mispredict performance counter.

[1]A complete overview can be seen in table A.18 in [18]

```
label:
instruction
... ; repeats 13X
instruction
subs
bne label
```

Fig. 3: Instruction loop

### C. Benchmarks

As a first approximation, the benchmark programs consists of an infinite series of identical instructions. Since the A9 core runs at a fixed frequency and we are providing a fixed core voltage, so energy usage per instruction can be deduced from the continuous power drain during instruction runs. In general, the power drain related to a particular instruction can be calculated as following:

$$P_{instruction} = I_{instruction} \cdot V_{core} \tag{1}$$

The fixed core clock cycle gives a fixed amount of time per cycle. Thus, we use clock cycles as our base time unit. Energy is then given as

$$E_{instruction} = P_{instruction} \cdot cycles \tag{2}$$

We are unable to measure core power directly with our equipment. However, voltage and frequency are assumed constant, which gives a linear relation between current and consumed energy. Instead of providing numbers in Joule per instruction, we measure the current drain and multiply with the number of cycles the instruction lies within any of the processors functional units. This gives us the unit of Ampere-cycles, which in our environment maps directly to energy. We neglect the voltage drop over the shunt resistor, which is in the order of a few mV.

This simple setup does not take the memory system into account; we are undoubtedly not able to feed the processor instructions at no cost in terms of access speed and – more importantly – memory system energy usage. Thus, we enhance our setup by running all benchmark code within fast-loop. To explicitly feed the processor instructions without the overhead of interrupts, we write Linux kernel modules that once inserted, execute a loop similar to the one shown in Figure 3. Note that the `subs` and `bne` are used to generate a loop body small enough to fit in fast-loop, and at the same time allows us to terminate the program after some number of iterations.

It is stated in [18] that branching to immediate locations does not use clock cycles. Our micro-benchmarks branches to immediate locations, but it does so conditionally. We assume that the calculation of the branch condition, the `subs`, takes its normal execution time, while the following branch instruction is invisible.

### D. Power Measurements

To measure energy consumption, we use an Agilent 34410A multimeter[19] to sense the voltage drop over a shunt resistor, set up as shown in Figure 1. The multimeter is configured to sample at its highest sampling rate of 10 kHz. This yields one sample every 170,000 instructions with an error of at most $1mV$. It is obvious that we are unable to observe inter-cycle fluctuations with this equipment, but as we run the same instruction practically indefinitely we extract the average. The loop for each instruction runs for about 20 seconds and we gather 50,000 samples (over a period of 5 seconds) in the middle of this loop. Observational errors are accounted for by running the power measurement loop many times for each instruction. We also sleep 30 seconds in between runs to dilute the effect of temperature variations. Running over the entire testbench takes about 100 minutes and we average the medians for each instruction run to get a single value.

We separate power consumption on the ARM cores and the development board by modifying the ODROID-X2 and providing a separate power supply for the A9 cores. They get powered by an external power supply giving $1.3V$ DC, while the rest of the board is powered from a another power supply at $5.0V$, as depicted in Figure 1. We cannot verify that CPU cores sit alone on the $1.3V$ power rail, but we observe a strong degree of correlation between core activity and $V_{core}$ power drain.

Certain instructions use more than one cycle to complete their work, so the energy usage has to be normalized. An instruction that occupies the pipeline for two cycles is believed to use approximately twice as much energy. By normalizing, we can convert point-in-time current drain in terms of Amperes to energy per instruction in Ampere-cycles.

### E. Pitfalls

We measure the current drain of different instructions separately, so we need to fix as many parameters as possible. We must acknowledge that some factors affects power consumption and produces noise in our data.

One obvious such factor is the chip temperature: it is known that power consumption increases at higher core temperatures. We explore the boundaries by physically applying cooling spray and notice that our measurements on average gets 4% higher with a temperature increase from 9°C to 63°C . The `mul` instruction had the greatest leap and used 7% more energy at 63°C. In our experiments, only one of the four available cores are used. Stressing a single core over time did not increase temperature by more than 7°C (from idle at 47°C to 54°C at load) and reached an equilibrium where temperature remained constant. Assuming that it is generally true that a single core cannot heat the entire SoC significantly, and that the increase in power consumption is at most 10% over 50°C, we get

$$P_{inc} = P_{orig} \cdot T_{inc} \cdot \frac{0.10}{50} = P_{orig} \cdot T_{inc} \cdot 0.002 \tag{3}$$

Assuming the trend is close to linear, output will increase by 0.2% per °C increased. Also, we start our measurements several seconds after the benchmarks, giving the core plenty of time to reach work temperature. For our purpose, the time used to reach work temperature was pretty much instant. Note that

this temperature logging was done with a different kernel as it required support for Dynamic Voltage and Frequency Scaling (DVFS), which we disabled in the test setup in order to fix the clock frequency.

Energy consumption is almost certainly affected by the amount of bit flipping within the core. In all the tests, the instruction arguments are static. This means that the results could be different if we changed the arguments. To mitigate this, we used as equal arguments as possible. Still, different instructions contain and use arguments differently, so we cannot guarantee complete fairness between instructions.

We are running Linux as the base environment for our tests, which makes it simpler to run our micro-benchmarks. However, running an entire operating system beneath our benchmark programs implies that there is much going on where we have no direct control. To mitigate the artifacts originating from the operating system, we disable all the maskable interrupts and run our benchmark programs entirely uninterrupted as a kernel module.

As explained in section subsection II-B, we utilize the fast-loop mode of the processor to avoid memory access latency. We disable the L1 cache to easier detect when we are outside the fast-loop mode, and thus we are certain that there is no memory access going on.

## III. Results

### A. Introduction

In this section we present data gathered from our experiments on the ARM Cortex-A9. A brief description of each instruction can be found in the "ARM and Thumb-2 Instruction Set Quick Reference Card"[20]. First, we discuss performance counters from experimental testbench runs. Together with the sparse official documentation, it enables us to make some assumptions about how different instructions are executed in the processor. We then discuss the results from the per instruction energy analysis.

### B. Decomposing the Core

Instructions executed in the processor will utilize a subset of all the available core components. By combining the components depicted in Figure 2 with the performance counter data listed in Table II and Table III, we can deduce which instructions that trigger what parts. We can also see how frequently each part of the pipeline is used, as a fraction of cycle count and the given component event counters.

All results in Table II and Table III are gathered by running each instruction included in our experiments using the template shown in Figure 3. The cycle count (*Cycles*) tells us how long time, in terms of clock cycles, it took for the processor to execute the $252 \cdot (13 + 2) = 3780$ instructions. The loop has room for 13 test instructions, while the last 2 is the loop head consisting of `subs` and `bne`. *Main Ex.* is the number of cycles where the main execution pipeline is active, labeled ALU/MUL in Figure 2. *Second Ex.* is for the second execution pipeline, labeled ALU. All instructions in our test bench have a correct branch prediction count of 251 (*Pred.*). This is most likely because the first and the last iteration of

| Instr. | Cycles | Main Ex. | Second Ex. | Pred. | Mis pred. | No disp. | Issue Empty |
|---|---|---|---|---|---|---|---|
| adc | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| adcs | 3599 | 1762 | 1756 | 251 | 2 | 1693 | 1690 |
| add | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| addeq | 6594 | 1635 | 1883 | 251 | 2 | 4709 | 4709 |
| addne | 3349 | 1762 | 1758 | 251 | 2 | 1463 | 1463 |
| adds | 3598 | 1761 | 1757 | 251 | 2 | 1712 | 1712 |
| and | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| ands | 3599 | 1762 | 1756 | 251 | 2 | 1693 | 1690 |
| asr | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| asrs | 6361 | 2135 | 1385 | 251 | 2 | 2968 | 204 |
| bic | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| bics | 3599 | 1762 | 1756 | 251 | 2 | 1693 | 1690 |
| clz | 2104 | 1824 | 1699 | 251 | 2 | 151 | 88 |
| cmn | 3599 | 1761 | 1757 | 251 | 2 | 1713 | 1713 |
| cmp | 3598 | 1761 | 1757 | 251 | 2 | 1712 | 1712 |
| cpsid | 14627 | 3516 | 1 | 251 | 2 | 11110 | 11110 |
| eor | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| eors | 3600 | 1762 | 1756 | 251 | 2 | 1694 | 1691 |
| lsl | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| lsls | 6361 | 2135 | 1385 | 251 | 2 | 2968 | 204 |
| lsr | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| lsrs | 6362 | 2135 | 1385 | 251 | 2 | 2969 | 205 |
| mov | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| movs | 3599 | 1762 | 1756 | 251 | 2 | 1693 | 1690 |
| mvn | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| mvns | 3600 | 1762 | 1756 | 251 | 2 | 1694 | 1691 |
| nop | 3604 | 3268 | 251 | 251 | 2 | 84 | 84 |
| orr | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| orrs | 3599 | 1762 | 1756 | 251 | 2 | 1693 | 1690 |
| pkhbt | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| pkhtb | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| qadd | 3598 | 1761 | 1757 | 251 | 2 | 1712 | 1712 |
| qdadd | 3600 | 1885 | 1633 | 251 | 2 | 1712 | 1588 |
| qdsub | 3600 | 1885 | 1633 | 251 | 2 | 1712 | 1588 |
| qsub | 3598 | 1761 | 1757 | 251 | 2 | 1712 | 1712 |
| rev16 | 2104 | 1824 | 1699 | 251 | 2 | 151 | 88 |
| rev | 2104 | 1824 | 1699 | 251 | 2 | 151 | 88 |
| revsh | 2105 | 1824 | 1699 | 251 | 2 | 152 | 89 |
| ror | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| rors | 6361 | 2135 | 1385 | 251 | 2 | 2968 | 204 |
| rrx | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| rrxs | 3599 | 1762 | 1756 | 251 | 2 | 1693 | 1690 |
| rsb | 1977 | 1762 | 1758 | 251 | 2 | 90 | 90 |
| rsbs | 3598 | 1761 | 1757 | 251 | 2 | 1712 | 1712 |
| rsc | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| rscs | 3599 | 1762 | 1756 | 251 | 2 | 1693 | 1690 |
| sbc | 1976 | 1762 | 1758 | 251 | 2 | 89 | 89 |
| sbcs | 3599 | 1762 | 1756 | 251 | 2 | 1693 | 1690 |
| sel | 2100 | 1761 | 1758 | 251 | 2 | 337 | 89 |
| setend | 14627 | 3516 | 1 | 251 | 2 | 11110 | 11110 |
| ssat16 | 3598 | 1761 | 1757 | 251 | 2 | 1712 | 1712 |
| ssat | 3598 | 1761 | 1757 | 251 | 2 | 1712 | 1712 |
| sub | 1977 | 1762 | 1758 | 251 | 2 | 90 | 90 |
| subs | 3598 | 1761 | 1757 | 251 | 2 | 1712 | 1712 |
| sxtab16 | 6443 | 3021 | 3761 | 251 | 2 | 1832 | 77 |
| sxtab | 4481 | 2932 | 2344 | 251 | 2 | 666 | 81 |
| sxtah | 6443 | 3021 | 3761 | 251 | 2 | 1832 | 77 |
| sxtb16 | 2104 | 1824 | 1699 | 251 | 2 | 151 | 88 |
| sxtb | 2104 | 1824 | 1699 | 251 | 2 | 151 | 88 |
| sxth | 2104 | 1824 | 1699 | 251 | 2 | 151 | 88 |
| teq | 3599 | 1762 | 1756 | 251 | 2 | 1693 | 1690 |
| tst | 3600 | 1762 | 1756 | 251 | 2 | 1694 | 1691 |
| usat16 | 3598 | 1761 | 1757 | 251 | 2 | 1712 | 1712 |
| usat | 3598 | 1761 | 1757 | 251 | 2 | 1712 | 1712 |
| uxtab16 | 6443 | 3021 | 3761 | 251 | 2 | 1832 | 77 |
| uxtab | 6443 | 3021 | 3761 | 251 | 2 | 1832 | 77 |
| uxtah | 6443 | 3021 | 3761 | 251 | 2 | 1832 | 77 |
| uxtb16 | 2105 | 1824 | 1699 | 251 | 2 | 152 | 89 |
| uxtb | 2104 | 1824 | 1699 | 251 | 2 | 151 | 88 |
| uxth | 2104 | 1824 | 1699 | 251 | 2 | 151 | 88 |

TABLE II: Performance counter data from 252 iterations of all tested instructions, excluding multiply

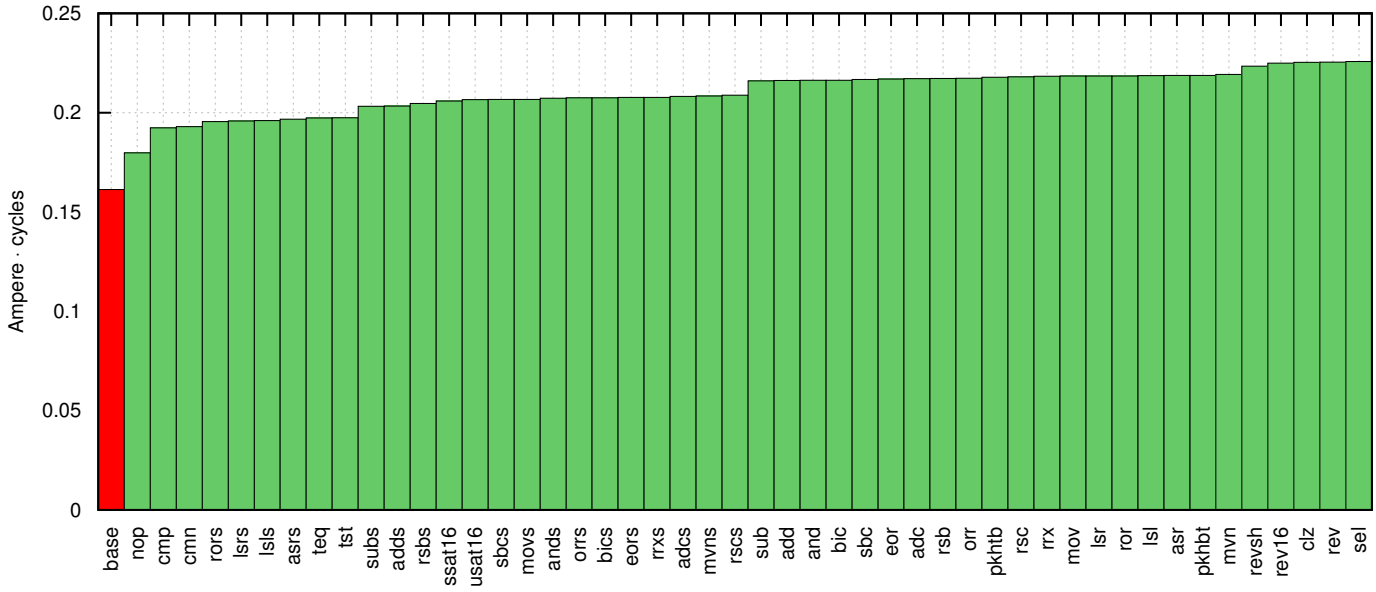


Fig. 4: Energy profile of single-cycle instructions, excluding multiply

the loop is mispredicted (*Mis pred.*). *No disp.* is the number of cycles where there the processor was unable to dispatch any instructions to any execution lane. *Issue Empty* is the number of cycles where there was no instructions in the instruction queue. Note that we can see how many cycles the processor is stalling by looking at the *No disp.* and the *Issue Empty* counters. When the *No disp.* number is higher than *Issue Empty*, it means that the processor had to stall due to hazards or intended flushing (e.g. `setend` flushes the pipeline as all further issued instructions must follow the new endianess). In special cases, such as our baseline instruction `setend`, we see that the amount of *No disp.* is very high, which again means that the CPU is mostly stalling. It seems to be a strong relation between low power usage and high stall numbers.

### C. Instruction Level Energy Efficiency

We distinguish between single-cycle instructions and multi-cycle instructions because they behave differently in and around the execution pipelines. Instructions using only one cycle are fairly easy to reason about as there is no need to normalize energy consumption with respect to the cycle count (i.e. time). However, it is important to also recognize CPU capabilities such as dual issuing which are present on the processor: most single-cycle ALU instructions execute pairwise in parallel – one in each ALU – giving a peak performance of two instructions per clock cycle. Multi-cycle instructions needs to be carefully considered. Typically, multi-cycle instructions divide work which can be done in a subset of the available ALUs (e.g. one) over several cycles, and can therefore introduce bottlenecks in the execution path. This again makes the processor do less, lowering the average current drain. For all these reasons, we partition the measured data in two data sets; one for single-cycle instructions and one for multi-cycle instructions.

Figure 4, 6, 7 and 5 displays our results from measuring current drain for each instruction. The instructions is sorted in increasing order by Ampere cycles. Green bars represents single-cycle instructions, light blue are two-cycle instructions, while dark blue represents three-cycle instructions. The red bar to the left on each graph shows the baseline for current measurement. The baseline is an alias for the least power-consuming instruction we could find, which is the `setend`-instruction. This instruction sets the endianness for all memory operations to either big or little endian [21], and has a current drain of only $161.3mA$ when executed repeatedly. This is expected because it would force pipelines to be empty most of the time.

During measurements, $V_{core}$ was kept stable at $1.3V \pm 50mV$, well within the specifications of the processor. The pipelines were kept as full as possible, avoiding hazards and instruction loading. This implies that instructions utilizing large parts of the processor will most likely be more energy consuming than those using only few components. This statement is supported by the fact that the `setend`-instruction has little pipeline activity at the same time as it has a low continuous current drain.

### D. Single-Cycle Instructions

On our target CPU, 70 of the 115 tested instructions[2] use a single cycle, while the remaining 45 uses 2 or 3. Nearly half of the instructions are multiply instructions, so these will be discussed separately. Figure 4 displays a comparison of the 50 non-multiply single-cycle instructions.

The results in Figure 4 shows that the ordinary single-cycle instructions do not differ very much. An interesting result is how instructions bearing the `s`-flag seems to have a lower consumption than their non-`s` companion. These instructions updates status flags and will likely force in-order execution. According to the performance counters in Table II there is reason to believe that the processor has to stall one cycle

[2] 119 including conditionals

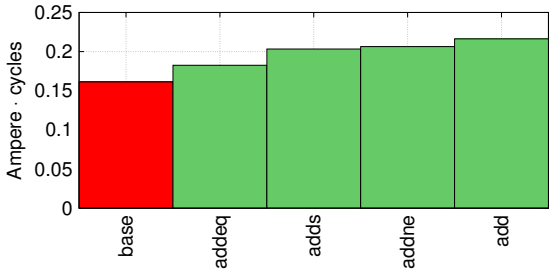


Fig. 5: Energy profile showing conditional execution.

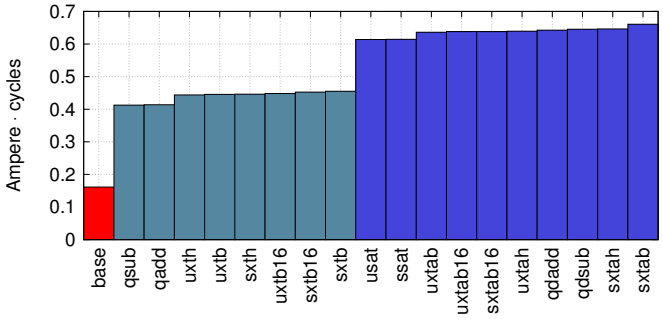


Fig. 6: Energy profile of multi-cycle instructions, excluding multiply.

between each issue. From our results, it seems that this saves energy. However, the instructions needs longer time to complete, which is indeed less energy efficient.

The results from the conditional-executed instructions are also subject to forced in-order execution. We can see from Figure 5 how variations of `add` compares. In the figured test, `addne` is committed every time, while `addeq` never has its results committed. It is interesting to see that even though `addeq` is never committed, it uses almost as much power as the other `adds`. By looking at Table II we see that `addeq` and `addne` introduces a lot of both *No disp.* and *Issue Empty*. We do not know exactly why, but it is reasonable to believe that one must assure that the previous instruction did not alter the status flags before the results are committed or discarded. In an in-order single-issue processor, conditional execution provides a framework to avoid unnecessary jumps, while in an out-of-order core, conditional execution is most likely much harder to implement. Also note that this test is very synthetic and the ISA is likely to be unoptimized for such activity. In a real world workload, it is possible that the required synchronization is hidden.

Further, Figure 4 shows that the `nop`-instruction has a rather low power consumption. This is a bit misleading, as the `nop`-instruction assembles to `mov r0, r0`, having both read-after-write and write-after-write hazard on itself. This makes the `nop`-instruction serialize itself, and it is hard to fill the pipeline with this instruction. Knowing this, it makes sense that `nop` works in this way, as it is often used to fill out clock cycles with non-destructive work. It would not make sense to optimize the `nop` instruction, as it then would fail to complete it's goal as a space-and-time filler.

Generally, when accounting for the number of cycles used by the different instructions, we see that the least current demanding single cycle instructions are `add` and `sub` at $216.2mA$, while `rev` and `sel` consumes slightly more with a drain of $225.4mA$. The measurements have a standard deviation of $4mA$ and $3.7mA$, respectively. Overall, the standard deviation ranges from $2mA$ to $7mA$, which we consider to be more than good enough.

### E. Multi-Cycle Instructions

45 of the instructions that was compared used 2 or 3 cycles to complete their results. 18 of these instructions are non-multiply. Non-multiply instruction power measurements are displayed in Figure 6. A selection of the performance counter results are shown in Table II. We see that the unsigned extend instructions(`ux*`) are slightly cheaper than signed extend (`sx*`). This might indicate that some hardware is left idle when not needing sign extension. The instructions are normalized according to their stated cycle count in the table B-5 in [18].

### F. Multiply

The ARM Cortex-A9 contains a single multiply pipeline, but has two general ALUsFigure 2. The multiply instructions are queued up waiting to execute through the same pipeline. This implies that multiply instructions would have a lower continuous power drain because it does less useful work and will seemingly use less energy compared to instructions utilizing both pipelines at its full potential. We have not compensated for this matter other than multiplying the power drain with the number of cycles used to finish one multiply instruction. It is unknown how the different multiply instructions utilize the pipeline(s). As we can see from Table III, there is reason to believe that at least some of the multiply-accumulate instructions utilize both pipelines[22]. This means that some instructions are able to utilize more hardware while still queuing up through the multiply-enabled main pipeline.

By looking at Figure 7 we see that the single-cycle multiply instructions are quite similar, but those using two or three cycles are more interesting. We do not know why the results are as stated, as most of the internal architecture are not available for the public. According to Table B-5 in [18], some multiply instructions uses more time than others before the result is available.

From the performance counters in Table III, we see that instructions are treated differently by the architecture. We have not considered all the tested instructions in detail, but it is evident to us that there is a strong negative correlation between performance counters (*No Disp.* and *Issue Empty*) and processor power drain. The results in Figure 7 shows power drain in Amperes multiplied by the cycle counts. The values are not normalized according to the performance counter values.

### G. Evaluation

Each instruction was measured 41 times. We found small variations in power consumption between testbench runs, but all results shows the same trend. As stated in subsection II-E,

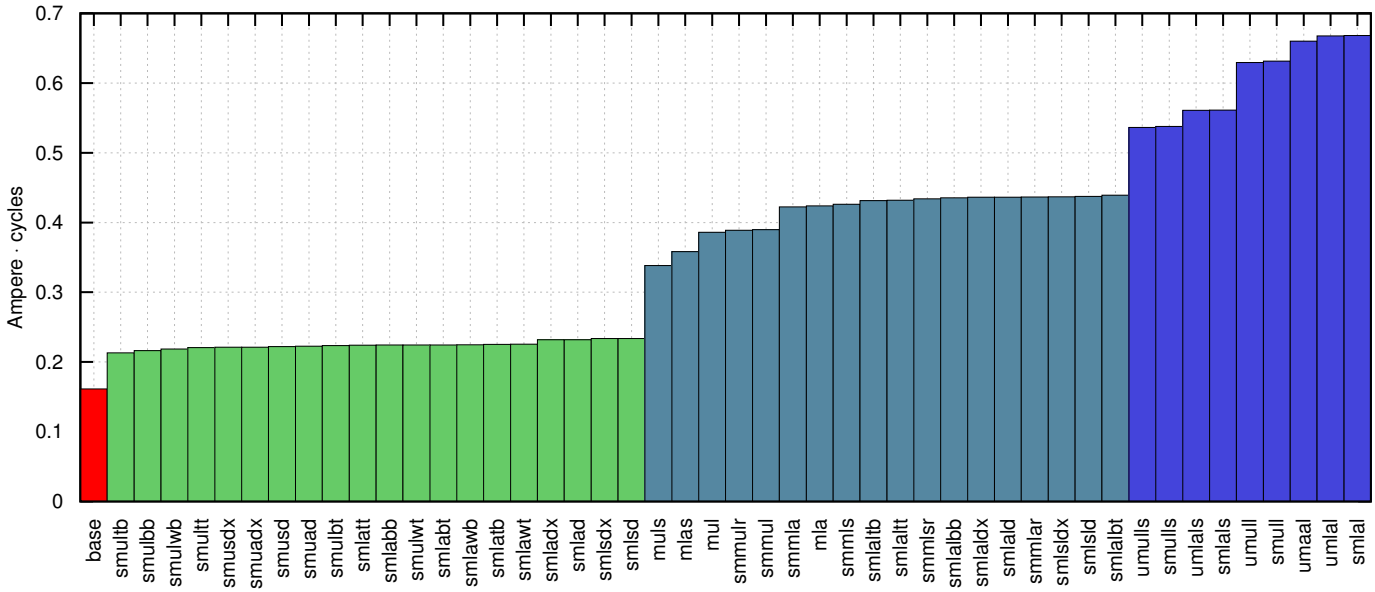


Fig. 7: Energy profile of multiply instructions.

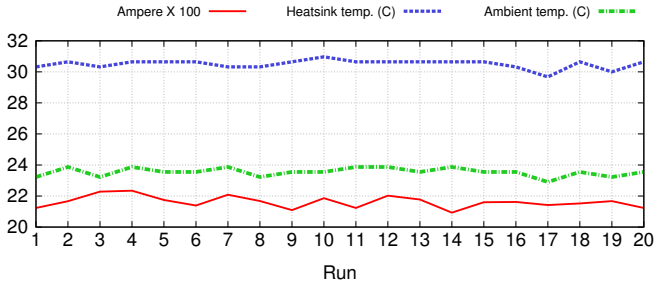


Fig. 8: Changes in heat and energy consumption for `add` at different runs together with heatsink and ambient temperature

the power consumption is not easily pushed by temperature. Figure 8 shows how the change in power consumption of the instruction `add` over different runs combined with the ambient temperature and the heatsink temperature. According to these results, we assume that the change in power consumption was not due to heat. We did not log temperature for all test runs, but assume that the results from Figure 8 holds, and that this small change in temperature is at least not solely responsible for variations in the current drain measurements.

## IV. Conclusion

We have explored the inner workings of the ARM Cortex-A9 processor core and measured energy consumptions for various instructions. We found that the energy consumption for simple single-cycle instructions are rather equal. This RISC processor also includes a range of more advanced instructions that needs more than a single cycle to complete. We have looked into all multi-cycle instructions related to multiply and multiply-accumulate, along with a few register level data movement instructions.

Our main observation is that those instructions that are unable to fully fill the pipelines comes out as more energy efficient on the current readings. This is most likely because near empty pipeline consume less energy than a full pipeline. We must emphasize that these instructions are not more energy efficient than their counterparts, only slower in producing their intended results. Apart from this, the numbers tell that the most efficient instructions are `sub` and `add`, followed by common logical functions. This is expected as all these instruction are both easily implemented and commonly used.

It is also seen that the instructions executing conditionally and those settings status flags are subject to a less efficient instruction dispatching. We assume that synchronization is needed for this kind of instructions. Conditional executing is most likely better idea in a simple in-order CPU than in advanced out-of-order CPU cores. We also notice that instructions that should not be committed is issued, executed and then discarded.

For the multi-cycle instructions, we observed that even though the processor datasheet[18] states a number of cycles for each instruction to complete its result, different pipelining schemes apply to the different instructions. Multiply can only be done in the main execution unit, while accumulate is seemingly executed in the second pipeline. This means that even though `mul` introduces queueing for access to the main pipeline, multiply-accumulate (`mla`), is equally fast, see Table III.

### A. Further Work

Our results comes from completely synthetic benchmarks, and we do not yet know how this would differ from real world workloads. The synthetic tests fill the pipeline with equal instructions, while common workloads would at least contain a few different instructions simultaneously.

The results was normalized according to numbers found in the CPU datasheet. We believe that more informative results would emerge if the performance counter data was used to adjust the measured current drain, rather than number of cycles used. This is after all a multiple-issue pipelined processor core.

| Instr. | Cycles | Main Ex. | Second Ex. | Pred. | Mis pred. | No disp. | Issue Empty |
|---|---|---|---|---|---|---|---|
| mla | 6608 | 6530 | 4895 | 251 | 2 | 76 | 76 |
| mlas | 15639 | 6529 | 12548 | 251 | 2 | 8857 | 73 |
| mul | 6602 | 6525 | 252 | 251 | 2 | 3336 | 76 |
| muls | 15617 | 6526 | 252 | 251 | 2 | 12100 | 61 |
| smlabb | 3604 | 3518 | 3265 | 251 | 2 | 83 | 83 |
| smlabt | 3604 | 3518 | 3265 | 251 | 2 | 83 | 83 |
| smlad | 3604 | 3518 | 3265 | 251 | 2 | 83 | 83 |
| smladx | 3604 | 3518 | 3265 | 251 | 2 | 83 | 83 |
| smlal | 7106 | 7028 | 5020 | 251 | 2 | 575 | 76 |
| smlalbb | 7102 | 7021 | 5019 | 251 | 2 | 3582 | 76 |
| smlalbt | 7102 | 7021 | 5019 | 251 | 2 | 3582 | 76 |
| smlald | 7102 | 7021 | 5019 | 251 | 2 | 3582 | 76 |
| smlaldx | 7102 | 7021 | 5019 | 251 | 2 | 3582 | 76 |
| smlals | 15888 | 6529 | 12548 | 251 | 2 | 9106 | 73 |
| smlaltb | 7102 | 7021 | 5019 | 251 | 2 | 3582 | 76 |
| smlaltt | 7102 | 7021 | 5019 | 251 | 2 | 3582 | 76 |
| smlatb | 3604 | 3518 | 3265 | 251 | 2 | 83 | 83 |
| smlatt | 3604 | 3518 | 3265 | 251 | 2 | 83 | 83 |
| smlawb | 3604 | 3518 | 3265 | 251 | 2 | 83 | 83 |
| smlawt | 3604 | 3518 | 3265 | 251 | 2 | 83 | 83 |
| smlsd | 3604 | 3518 | 3265 | 251 | 2 | 83 | 83 |
| smlsdx | 3604 | 3518 | 3265 | 251 | 2 | 83 | 83 |
| smlsld | 7102 | 7021 | 5019 | 251 | 2 | 3582 | 76 |
| smlsldx | 7102 | 7021 | 5019 | 251 | 2 | 3582 | 76 |
| smmla | 6608 | 6530 | 4895 | 251 | 2 | 76 | 76 |
| smmlar | 6608 | 6530 | 4895 | 251 | 2 | 76 | 76 |
| smmls | 6608 | 6530 | 4895 | 251 | 2 | 76 | 76 |
| smmlsr | 6609 | 6530 | 4895 | 251 | 2 | 77 | 77 |
| smmul | 6602 | 6525 | 252 | 251 | 2 | 3336 | 76 |
| smmulr | 6603 | 6525 | 252 | 251 | 2 | 3337 | 77 |
| smuad | 3602 | 3266 | 252 | 251 | 2 | 334 | 334 |
| smuadx | 3602 | 3266 | 252 | 251 | 2 | 334 | 334 |
| smulbb | 3353 | 3267 | 252 | 251 | 2 | 84 | 84 |
| smulbt | 3353 | 3267 | 252 | 251 | 2 | 84 | 84 |
| smull | 6857 | 6779 | 6774 | 251 | 2 | 326 | 76 |
| smulls | 15637 | 6528 | 15558 | 251 | 2 | 8856 | 72 |
| smultb | 3353 | 3267 | 252 | 251 | 2 | 84 | 84 |
| smultt | 3353 | 3267 | 252 | 251 | 2 | 84 | 84 |
| smulwb | 3353 | 3267 | 252 | 251 | 2 | 84 | 84 |
| smulwt | 3353 | 3267 | 252 | 251 | 2 | 84 | 84 |
| smusd | 3602 | 3266 | 252 | 251 | 2 | 334 | 334 |
| smusdx | 3602 | 3266 | 252 | 251 | 2 | 334 | 334 |
| umaal | 7106 | 7028 | 5020 | 251 | 2 | 575 | 76 |
| umlal | 7106 | 7028 | 5020 | 251 | 2 | 575 | 76 |
| umlals | 15888 | 6529 | 12548 | 251 | 2 | 9106 | 73 |
| umull | 6857 | 6779 | 6774 | 251 | 2 | 326 | 76 |
| umulls | 15637 | 6528 | 15558 | 251 | 2 | 8856 | 72 |

TABLE III: Performance counter data from 252 iterations of all tested multiply instructions.

Also, we have not yet dived into how instruction arguments affects the energy usage on modern processors. We believe that instruction patterns that causes a high degree of bit toggling would yield higher energy usage, due to the amount of energy used to charge and release the transistors. A problem rising is the fact that we do not know how the processor schedules or distributes the different instructions, thus one has to be very careful when writing the benchmarks.

When selecting instructions for our benchmarks, we have omitted the set of floating-point instructions. This is because in the ARM Cortex-A9, the floating point unit (NEON) is considered a co-processor[18], and thus out of our scope. Investigating the energy efficiency of co-processors versus processors that embed such functionality would add value to our results.

There are also room for improvements regarding the experimental setup. Ultimately, one would like to be able to measure each instruction individually, but according to the Nyquist-Shannon theorem[23], this would require a sampling rate of at least 3.4 GHz. We could not simply go slower on the clock, as a clock frequency reduction will affect the energy efficiency, possibly in the negative direction[24].

Compilers, simulators and synthesis tools would benefit from this kind of information, and one could possibly generate output that is more energy optimized than currently available.

## REFERENCES


[1] G. E. Moore *et al.*, "Cramming more components onto integrated circuits," 1965.

[2] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, and H.-S. P. Wong, "Device scaling limits of Si MOSFETs and their application dependencies," *Proceedings of the IEEE*, vol. 89, no. 3, pp. 259–288, 2001.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 365–376.

[4] S. P. Dandamudi, *Guide to RISC Processors for Programmers and Engineers*. Springer, 2005, ch. 3.

[5] A.-E. Bogen, "Risc versus cisc," http://alfbogen.com/2013/06/16/risc-versus-cisc/.

[6] S. A. M. Karan Singh, Major Bhadauria, "Real Time Power Estimation and Thread Scheduling via Performance Counters," May 2009.

[7] R. B. et.al., "Decomposable and responsive power models for multi-core processors using performance counters," June 2010.

[8] Bircher and John, "Complete system power estimation using processor performance events," April 2012.

[9] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, 2010, pp. 21–21.

[10] NTNU, "The Single-ISA Heterogeneous MAny-core Computer (SHMAC)," http://www.ntnu.edu/ime/eecs/shmac, Desember 2013.

[11] Y. Umuroglu, "SHMACsim : A Cycle-accurate Simulation Infrastructure for the Heterogeneous SHMAC Multi-Core Prototype," p. 111, 2013.

[12] L. T. Rusten and G. I. Sortland, "Implementing a heterogeneous multi-core prototype in an fpga," Ph.D. dissertation, Norwegian University of Science and Technology, 2012.

[13] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 2003, pp. 81–92.

[14] Hardkernel, "hardkernel.com," http://www.hardkernel.com/renewal_2011/products/prdt_info.php?g_code=G135235611947.

[15] J. L. H. David A. Patterson, *Computer Organization and Design*, 4th ed. Morgan Kaufmann, 2012.

[16] D. A. P. John L. Hennessy, *Computer Architecture, A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012.

[17] "The ARM Cortex-A9 Processors," http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf.

[18] "Cortex-A9 Technical Reference Manual revision r3p0," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388g/DDI0388G_cortex_a9_r3p0_trm.pdf.

[19] I. Agilent Technologies, "Agilent 34410A and 34411A multimeters data sheet," http://www.home.agilent.com/en/pd-692834-pn-34410A/digital-multimeter-6-digit-high-performance, April 2013.

[20] "ARM and Thumb-2 Instruction Set Quick Reference Card," http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf.

[21] "ARM Compiler toolchain Version 4.1 Assembler Reference," http://infocenter.arm.com/help/topic/com.arm.doc.dui0489c/DUI0489C_arm_assembler_reference.pdf.

[22] R. Radhakrishnan, "Integer pipeline description for cortex a9." http://gcc.gnu.org/ml/gcc-patches/2009-10/msg01858.html, October 2009.

[23] C. E. Shannon, "Communication in the presence of noise," *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, 1949.

[24] T. D. Burd and R. W. Brodersen, "Energy efficient cmos microprocessor design," in *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, vol. 1. IEEE, 1995, pp. 288–297.