UPPSALA
UNIVERSITET

UPTEC IT 13 016
Examensarbete 30 hp
November 2013

# Breeding power-viruses for ARM devices

Ludvig Norinder

UPPSALA
UNIVERSITET

**Teknisk- naturvetenskaplig fakultet**
**UTH-enheten**

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

# Abstract

## Breeding power-viruses for ARM devices

*Ludvig Norinder*

Designing power-viruses, programs created for consuming as much power as possible, is a non-trivial task. This task is often performed by hand and is both time-consuming and complicated. As power-viruses may be used for testing the stability of hardware it is important that the viruses are well designed. This thesis presents an approach to automate the process of creating power-viruses with the help of Artificial Intelligence. Furthermore, the process of generating these programs will be performed on real hardware rather than using simulators. The hardware considered in this thesis is the Pandaboard ES and Raspberry PI, two boards built around ARM-based System-on-a-chip's. During the thesis, power-viruses have been successfully generated on both the Pandaboard ES and Raspberry PI. On the Pandaboard ES up to a 7.1% power-consumption increase has been achieved when compared with hand-written power-viruses for the same hardware. The process used in this thesis is easy to use and reduces the effort required for designing a power-virus.

Handledare: Tony Collander & Tobias Skoglund
Ämnesgranskare: Philipp Rümmer
Examinator: Lars-Åke Nordén
ISSN: 1401-5749, UPTEC IT 13 016
Tryckt av: Reprocentralen ITC

# Acknowledgments

This section aims to display gratitude towards the persons who have helped or supported me throughout this master thesis. The following persons are presented in no specific order.

Thanks to Lars for the excellent soldering-skills and breaking out that tiny INA, it was used for the entire duration of the thesis. Thanks to Patrik for technical consultation. Thanks to Ville for cheering me up throughout the thesis. Thanks to Martin for always helping me find the right equipment among all the things in that room. Thanks to Linnea for general support and cheering me up. Thanks to **all** the people in room 1413 for letting me win once in a while in Dominion.

Finally, a sincere thank you to Tobias for making this thesis happen.

# Contents

# 1 Introduction

A power virus is a program written specifically for stressing a processor such that it consumes as much power as possible. Writing code which accomplishes this is hard and requires both detailed knowledge and understanding of the targeted hardware. Using an experimental approach, this process can become time consuming. By using Artificial Intelligence (AI) for creating power viruses, this process becomes faster and requires less detailed knowledge of the hardware. Furthermore, possibly better results can be obtained.

It is possible to approximate the maximum power consumption as the sum of the theoretical maximum consumption of all concerned components. This is typically not a realistic approximation as a simultaneous maximum consumption of all different components in the hardware is impossible to achieve. A more realistic approximation can be found by using a power virus, designed specifically for the hardware, to maximize the power consumption. This is but one area where automated generation of power viruses can be useful. Consider the modification of a system under special circumstances such as overclocking, new casing, extreme environments or similar. Optimized power viruses can be used to test the stability of the system in the new environment or configuration.

Assuming it is possible to recognize and define certain patterns of assembler instructions in generated power viruses which consume substantially more power than others, compilers could be improved to avoid such patterns of instructions in systems where energy consumption is crucial. Many modern smartphones and various other systems contain embedded circuits for which a prolonged battery life would be greatly appreciated.

There are few programs designed to stress test ARM systems. Thus new freely available stress tests may be of interest for the more popular versions of ARM hardware. As shall be shown, some available stress tests fail to reach peak power consumption. Furthermore, the creation of these stress tests generally requires very skilled software designers, and takes considerable time.

To avoid the tiresome task of writing power viruses by hand, this thesis evaluates automated design of test code sequences with the help of AI. More specifically, a type of optimization technique known as genetic algorithm will be used. Automatically generated test code will be executed on real hardware. The power usage is measured and used as feedback for the AI to ascertain power consumption for each iteration of the optimization. Previous work have used simulators. However, simulators cannot always simulate all details exactly according to the real hardware. Therefore this thesis evaluates the use of real hardware for automation of power virus code generation.

The use of AI for generating power viruses makes the process less complicated and time-consuming. Using AI for generating such programs effectively is not a straightforward process. One key result in this thesis are generated programs that out-perform hand-written programs freely available online. This approach is evaluated on two development boards built around ARM microprocessors, namely the Raspberry Pi and the Pandaboard ES. The boards were chosen mainly due to two reasons. First, the boards are running different versions of the ARM architecture and offer different sets of functionality. Thus, a sufficiently generic method to work for both boards is needed. Second, both microprocessors can be found in consumer products such as smartphones or tablets, but are also commonly used by hobbyists and tinkerers and can be easily obtained. The

systems running on the hardware will be common variants of Linux to make the results easily reproducible.

The approach proposed in this thesis, as previously mentioned, uses real hardware in combination with AI and an instruction selection method to aid in configuring the process. The approach has the benefit of being simple and requiring very little detailed knowledge of the system under evaluation while still delivering good results. As will be shown, the programs generated with this method consume more power than hand-written freely available programs created for the same purposes. This thesis does also present an instruction-centric approach to generation of power-viruses and performs selection among instructions in an ISA, followed by generation of programs on instruction precision basis.

## 1.1 Related work

The idea of using AI for generating power viruses is not new. MAMPO, an automatic power virus generation framework for multi-core systems, produced promising results for multi core systems [8]. SYMPO, another power-virus framework, produced good results on single core systems [9]. Both projects use genetic algorithms for code generation and use simulators for measuring performance and power during the code generation. When generating code, both projects use a frequency based algorithm, where instructions are generated based on frequency values. In contrast to both MAMPO and SYMPO, this thesis will be evaluated on real hardware and also proposes a different method for code generation. The main difference when compared to previous work is the more instruction centric approach used in this thesis which offers a high instruction precision for the AI. This thesis does not focus on generation of multi-threaded power-viruses, but will evaluate the results of the generated single core power-viruses on multiple cores.

The results of this thesis will be measured and compared against two different versions of CPUBurn for the Pandaboard ES, namely ssvb-cpuburn-a9 [17] and burncortexA9 [11], from here on referred to as cpuburn respectively burnCortexA9. Both cpuburn and burnCortexA9 were created for the purpose of consuming power and/or inducing heat in the circuit. For some comparisons, cpuburn has been modified to not spawn more than one process. This modification does not change the behavior in the main calculation loop of the program. However, cpuburn will be compared in its original state against the results in this report as well. On the Raspberry PI, no available power-viruses were found. As comparison with other programs is necessary for evaluating the result of the thesis, the PARSEC suite will be run to represent the power consumption of multiple "real" programs.

# 2 Background

This section briefly presents some common hardware features, the hardware considered in this thesis and also an introduction to genetic algorithms.

## 2.1 An architecture primer

This section briefly describes computer architecture concepts which will appear throughout this report. This serves to refresh the concepts for the reader and is considered optional reading.

### 2.1.1 Pipelining

> Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction. Today, pipelining is the key implementation technique used to make fast CPUs. [10]

A pipeline can be seen as an assembly line in a factory. Consider an assembly line with multiple stages where each stage adds a part to the final assembled product. Then, in an assembly line with $n$ stages, a total of $n$ different products can be assembled simultaneously, one for each stage. Assume for this example that the time required for assembly in each stage is the same for all stages, e.g. 1 time-unit. Once the pipeline is full, one completely assembled product will appear at the end of the assembly line per time-unit. Compare this to the assembly process with only one stage in which all work is performed and no parallelism is exploited. Assuming the same amount of assembly is to be done, a total of $n$ time-units is required for each product, at any time. This simplified example introduces the concept of pipelining and the increase of throughput it can cause.

Computers utilizes the concept of pipelining when executing instructions. The goal of executing instructions in a pipelined fashion is speed and an increased throughput resulting in fewer cycles-per-instruction (CPI).

### 2.1.2 Memory model

Multiple levels of hardware are involved in the process of accessing memory. Main memory is typically big and slow. Smaller and faster caching memories were added to reduce the number of accesses to main memory. An example of a common hierarchy would be: registers, L1-cache, L2-cache, L3-cache and main memory. Every step up in the hierarchy increases latency and size of the memory. The registers are the smallest and also fastest level. When the processor requests a piece of memory, it tries the L1-cache. If the L1-cache does not contain the wanted data, the next level in the hierarchy is tried. Once found, the requested memory is inserted into all levels of the caches before continuing. Caches work with chunks of data, known as cache lines, rather than words.

### 2.1.3 Out-of-Order execution, NEON

A statically scheduled pipeline fetches instructions and issues them in sequence. If an instruction to be executed is depending on a currently executing instruction

in the pipeline, the pipeline may be stalled until the dependency is resolved and then continue execution. Stalling the pipeline hurts performance. A common way of optimizing the hardware utilization is to use out-of-order execution. This means that instructions does not have to be executed in the sequential order in which they are appearing in a program. Instead of stalling execution units due to a dependency issue, other instructions without dependencies can execute while the dependency is resolving. Less stalling leads to a higher efficiency. When executing instructions out of order a reorder buffer makes sure that side-effects of the executed instructions occurs in the expected order.

ARM NEON is a general purpose Single Instruction Multiple Data (SIMD) engine. NEON instructions consider register data to be vectors of elements of the same data type and applies operations on these vectors. SIMD usage is commonly on media data such as video, images or audio [2].

## 2.2 Power consumption

As previously mentioned, power viruses are designed to maximize power consumption. The difficulties in designing this kind of software by hand is explained in [8]. It is stated that the process of writing a power virus is tedious. More specifically this is due to the many components interacting in the hardware when executing a piece of code. Power saving features such as clock gating or dynamic voltage scaling makes this even more complicated.

The primary energy consumption for CMOS hardware comes from switching transistors. The power required for a transistor can be calculated with the following formula: $\frac{1}{2} * Capacitive\_load * Voltage^2 * Frequency\_switched$ [10]. Writing code which utilizes the right parts of the hardware, thus switching the right transistors at the right time and in the right sequence, is non-trivial with the complexity of todays hardware.

## 2.3 Considered hardware

This subsection provides a quick introduction to the hardware used in this thesis. It is meant to show the reader the range of the functionality and capabilities provided by the boards.

### 2.3.1 ARM

ARM is currently the world's leading semiconductor intellectual property company. Their business model involves designing technology and licensing it rather than manufacturing the actual hardware. Licensed partners may then use ARM's intellectual properties for manufacturing actual semiconductor chips. Since the company started in 1990, over 40 billion ARM based chips have been shipped. As of today, ARM technology can be found in 95% of smartphones, 80% of digital cameras, and 35% of all electronic devices [3]. Among the products offered by ARM are 32-bit RISC microprocessors, graphics processors and memory.

### 2.3.2 Raspberry PI

Raspberry PI is an embedded computer of the size of a credit card. It was built around the BCM2835 System-on-a-Chip (SoC) manufactured by Broadcom. It

exists in two similar versions, A and B. The latter will be used in this thesis and cost about 42 Euros. The following is some of the functionality offered: an ARM1176JZF-S processor running at 700 MHz with a floating point unit, a Videocore 4 GPU, HDMI support, an Ethernet port and 512 MB RAM. It runs of 5V over a micro-USB connector and requires a power supply which can source 700 milliamperes [7].

### 2.3.3 Pandaboard ES

The Pandaboard ES is a single board computer built around the OMAP4460 System-on-a-Chip (SoC) from Texas Instruments. It is intended to be used as a platform for software development. At the time, the price was about 150 Euros. Among the offered connectors found on the board are HDMI, DB-9, USB OTG/USB host, SD/MMC and ethernet (RJ-45). Wireless LAN and Bluetooth functionality is available as well. See the Pandaboard reference manual for more information [16]. It runs of 5V over a center-positive 5mm DC barrel connector.

### 2.3.4 OMAP4460

The features mentioned in this section are a subset of the functionality found in the OMAP4460 chosen to be relevant for this thesis. This thesis aims to generate stress tests for the CPU and thus not all of the available functionality in the SoC is mentioned.

The OMAP4460 is a system-on-a-chip manufactured by Texas Instruments. It is a SoC with support for multiple operating systems such as Linux, Palm OS, Symbian OS and Windows CE. The SoC itself is a Cortex-A9 microprocessor unit with two ARM Cortex-A9 cores. It is capable of streaming video up to full HD resolution at 30 fps and draw 2D/3D graphics powered by a graphics accelerator subsystem based on POWERVR SGX540 from Imagination Technologies.

The two Cortex-A9 cores supports ARM version 7 ISA and Thumb-2. Each core has its own NEON SIMD and VFPv3 co-processor. Further, each core has its own 32kB instruction and 32kB data level 1 caches. The L1 cache-line size is 32B and the caches are 4-way set associative. Both cores shares a 1MB L2 cache. The L2 cache-line size is also 32B and 16-way set associative. A snooping protocol is used to maintain data cache coherence between CPUs. The Cortex-A9 is a SMP architecture (Symmetric Multi-Processor) superscalar with a 8-stage pipeline. It has out-of-order (OoO) instruction dispatch and completion.

Power management is an important device design aspect on embedded systems. Included in the OMAP4460 are power management techniques which can disable parts of the hardware by disabling its clock or reduce consumption by scaling frequencies, voltage or both [13].

## 2.4 Genetic algorithms

The genetic algorithms were invented by John Holland at the University of Michigan in the 1960s and was further developed in the 1960s and 1970s. They belong to a group of optimization techniques labeled with Evolutionary Computation, a subfield of Artificial Intelligence. As the name and classification

may imply, the genetic algorithms were inspired by nature, namely evolution, in which a solution improves over time. Due to this it is encircled with terminology borrowed from biology and evolution [14]. This section serves as an introduction to genetic algorithms and introduces the necessary terminology.

### 2.4.1 Algorithm description

The genetic algorithms consider populations of candidate solutions to a problem. Each individual, or candidate solution, in the population is commonly referred to as a chromosome and consist of multiple genes. Each gene can be said to represent a certain feature of the chromosome and is often represented by a bit, an integer or a real value. Starting with a population which can be chosen randomly or pseudo-randomly, the algorithm breeds new generations, hopefully of higher quality. The goal is to evolve towards an optimal solution. The three operators below are the basic operators of a genetic algorithm [15] and are introduced as the terminology appears throughout this report:

**Selection:** Selects individuals for reproduction from a population based on their fitness, i.e. the quality of the solution. Generally, the fitter, the more likely a chromosome is to be picked and the more likely it is to reproduce. Roulette selection and tournament selection are two examples of selection algorithms [14].

**Crossover:** Crosses the genes of two chromosomes and creates an offspring with features from both parents. Multiple algorithms for crossover exists and is to be chosen depending on the chosen representation of an individual [14].

**Mutation:** Changes some genes in a chromosome at random. The algorithms used for mutating chromosomes can vary depending on the representation of an individual.

In [15] the flow of a simple genetic algorithm is presented according to the steps enumerated below. Multiple variations exist with slightly different behavior [14], for example with Elitism or the Steady-State genetic algorithm.

1. Start with a population of $n$ randomly generated individuals
2. Evaluate the fitness of each individual in the population.
3. Until a new population with $n$ individuals has been created
   (a) Select parents using the selection operator
   (b) With a certain probability (knowns as the crossover rate), perform crossover. If no crossover is performed, the two children will be clones of each respective parent.
   (c) With a certain probability (known as the mutation rate), perform mutation on the children. Add the resulting individuals to the new population.
4. Replace the current population with the new population and go to step 2.

Every population is referred to as a generation and typically 50 - 500 or even more generations are iterated throughout a complete run. A population typically consists of 50 - 1000 individuals [15]. Population size, crossover rate and mutation rate are often tweaked to suit different problems and models. The crossover rate and the mutation rate can be varied between zero and one, since they represent a probability.

# 3 Experimental setup

In order to measure the power consumption of the boards, measurement equipment was needed. Since the main point of interest was the CPU, high resolution measurement equipment was needed as the CPU may consume merely a fraction of the composed consumption of the board. A configuration for measuring power consumption, logging the data and allowing for arbitrary programs to be executed on the board under evaluation was needed. This section describes these parts of the thesis.

## 3.1 Measuring power

To measure the power consumption of the CPU and memory on a board, it would be necessary to locate the power supply connectors for each part and somehow attach measurement equipment to these connectors. This process would have to be repeated for each board. By instead measuring the consumption of the entire board, less board specific knowledge is required and connecting new devices become easy. One of the goals of this thesis was to make the process of generating power-viruses less complicated. Therefore it was decided to measure the power consumption of the entire board rather than specific parts. The total power consumption of a board can be seen as consisting of two parts: a base consumption (required for the board to be on, but idling) and the extra power consumption caused by executing a program. If the base consumption can be considered constant, it is possible to tell if program A consumes more power than program B by comparing the total consumption of the board for each program.

Less expensive multimeter models are generally not capable of continuously measuring currents larger than a few hundred milliamperes and devices capable of logging data comes at a much higher price. For these reasons, custom measuring equipment was designed and built for this thesis. More specifically, the circuit measures the voltage drop over a shunt resistor, which is an inexpensive component. The shunt resistor was connected in series with the board under evaluation. An overview of the circuit created for measuring current is shown in Figure 1.

The shunt voltage drop was measured using an INA219 integrated circuit from Texas Instruments, another inexpensive component created specifically for this purpose. The INA219 contains a 12-bit ADC for measuring differences in voltage and its precision is configurable to the application. The conversion time is also configurable and ranges from 84 $\mu$s to 68.10 $ms$. The IC allows for data acquisition by communication using I2C or SMBUS protocols [12]. The shunt resistor has a minimal effect on the circuit as a whole because of its small resistance. The impact on the circuit can be calculated using Ohm's law ($U = R * I$) where $R = 0.02$ Ohm and $I = 3$ A. This calculation shows that: $U = 0.02 * 3 = 0.06$V. Thus 3A results in a reasonably small voltage drop considering that the boards are fed with 5V, therefore this leaves the boards well within their operating voltage range.

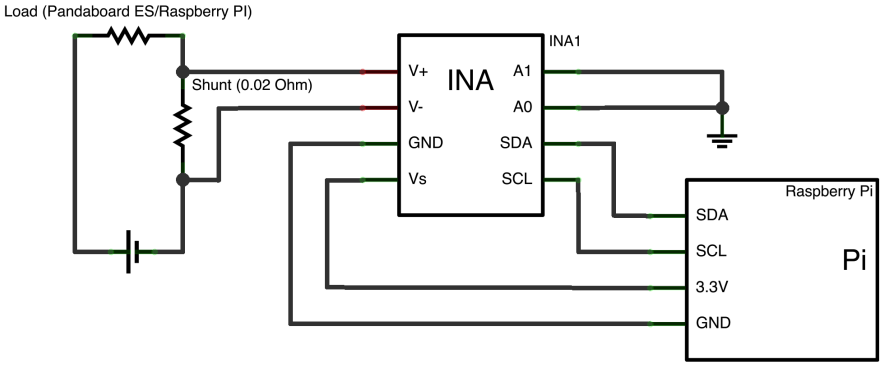



Figure 1: Schematic for measurement setup

| Platform | Operating system | Kernel |
|---|---|---|
| Pandaboard ES | Ubuntu 12.04 | 3.4.0-1490-omap4 |
| Raspberry PI, model Bv1 | Raspbian | 3.6.11+ |

Table 1: Software installed on boards

## 3.2 Configuration and program sampling

The INA219 was configured to calculate the average value of multiple samples before reporting back to the logging server receiving measurements. In this thesis a second Raspberry PI was used as logging server as its GPIO pins allows for a straight-forward I2C communication setup. A networking daemon was programmed for the logging server to allow for further distribution of measurement data over Ethernet. The setup was then calibrated manually using a multimeter and small currents in order to achieve relatively accurate measurements. Using this setup, enough accuracy to capture the power behavior of a board during runtime was achieved.

The boards chosen for investigation were running different distributions of Linux with differing kernel and software versions. An overview can be found in Table 1. The installed system had unnecessary functionality disabled by means of unloading kernel modules and removing unnecessary software services in order to reduce potential noise in the upcoming measurements.

To minimize the time needed for each program, the CPUFreq governor was set to performance mode which essentially sets the CPU statically to its highest frequency [5]. The time allocated for evaluation of each program is limited and statically locking the frequency may reduce the time taken to reach full CPU usage (although the governor switches frequency quickly).

Further functionality may be disabled by building custom kernels, this was however not required as the measured results showed a practically useful precision. Also, the results of this thesis are easier to reproduce if little customization is done. The final stability measurements includes reoccurring peaks, most visible when idling, which does not pose a problem for the measurements due to calculating the results using a median value. Figure 2 shows typical current usage levels for an idling Pandaboard ES before and after configuration.

Many components on each board could interfere with the measurements.

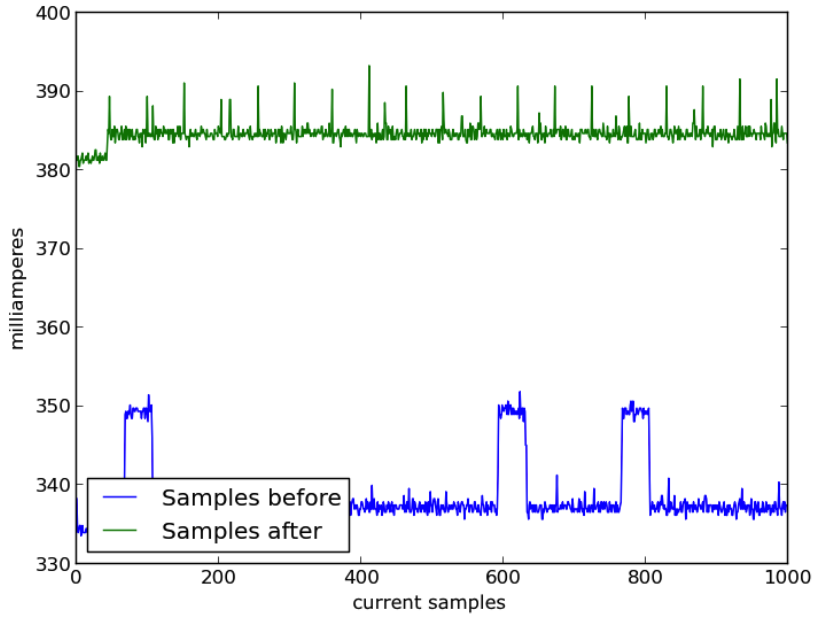



Figure 2: Current drawn by the Pandaboard ES when idling prior to and after configuration. The square-shaped noise turned out to be caused by the on board LEDs, showing the kernel heartbeat. The heightened average power usage after configuration is due to settings the CPUFreq governor in performance mode.

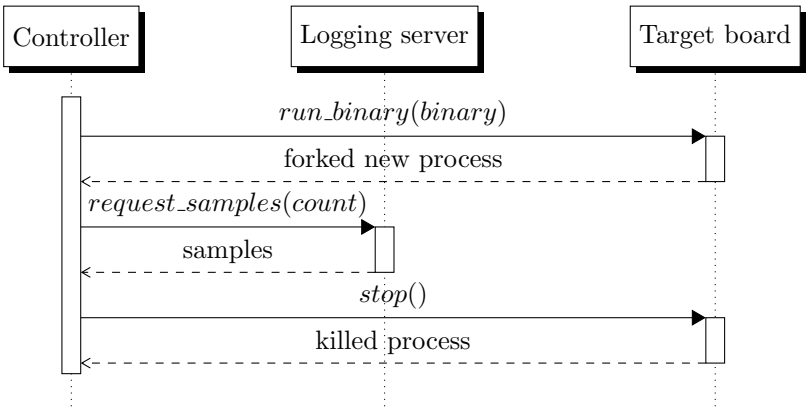



Figure 3: The sequence of communication between units

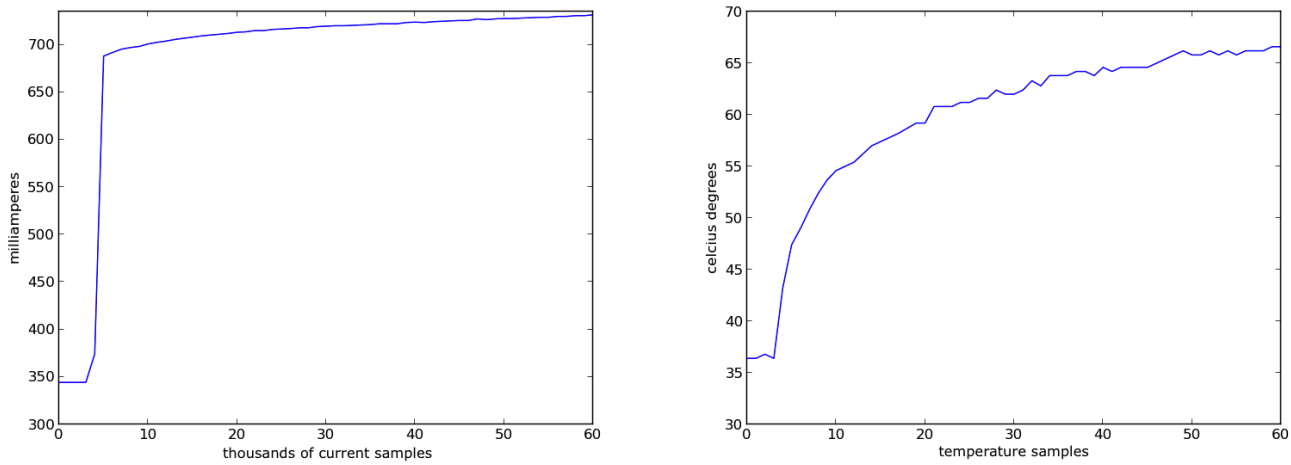
The processor accounts for only one, although significant, part of the entire power budget. Therefore the highest accuracy is achieved when no peripheral units are attached to the board and communication modules have been disabled. Due to the inter-communication setup used in this thesis, one network interface needed to remain connected to an Ethernet network. Minimization of network traffic to the board under evaluation was considered and the implementation makes sure that no connections are established to the board when recording measurements. Figure 3 shows the communication sequence when running a program and sampling its power consumption.

Essentially, a program is cross-compiled by a controlling computer (the controller), resulting in a binary compatible with the board under evaluation. The binary is then passed to the board, who in turn forks and executes it. Samples are taken while the new process is running. The controlling computer then tells the board to kill the process once enough samples has been collected. This procedure can then be repeated an arbitrary number of times. The possibility to execute binaries which forks more than one process was considered when designing the software running on the board under evaluation. Thus, when the controller wants to stop a running binary, no zombies are created.

## 3.3 Managing heat

During experimentation with the setup described in the section 3, peculiar behavior was discovered early with the Pandaboard ES. This section describes this behavior and addresses the methods used to circumvent and/or minimize their impact.

During longer runs on the Pandaboard ES an increase of heat was noticed, even during single core program generation. This is to be expected, especially at the end of longer runs, as the average power consumption of generated programs is expected to be high. On the Pandaboard the power consumption of the board increased as the temperature increased. This behavior becomes an issue as the genetic algorithm favors programs which consume more power than other programs. By introducing false data caused by the increased temperature and thus a higher power consumption, the wrong programs may be favored. As a result, suboptimal results may be generated.




(a) Current drawn

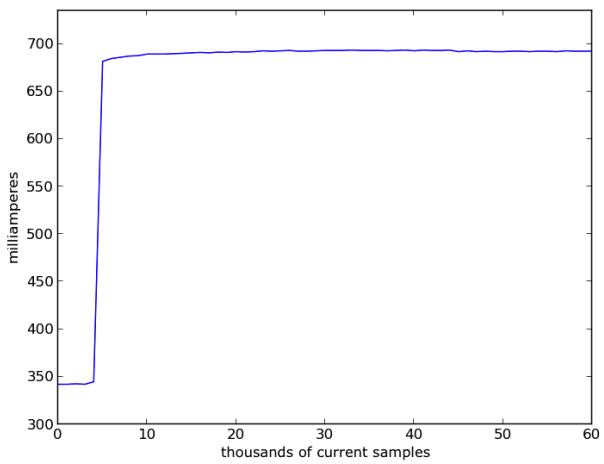
(b) Heat measured

Figure 4: The current usage and temperature of the Pandaboard ES without added cooling when running a single core program. The x-axis represents every $n * 1000$ current samples. 1000 samples takes approximately 1.8s to sample and collect

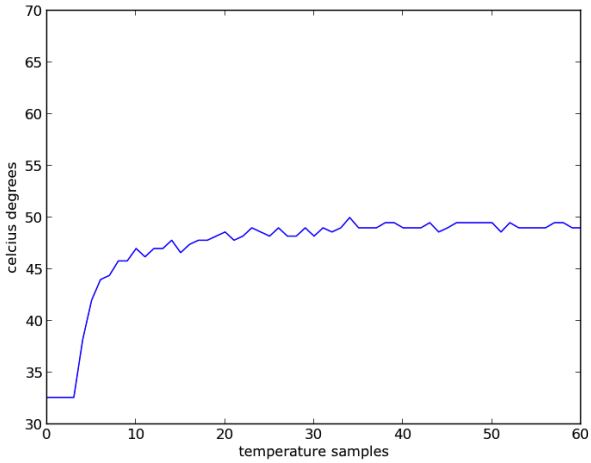
To determine the component with the highest temperature on the physical board a Dibotech IR-temperature meter was used. The specific meter is capable of measuring from $-50°$ to $500°$ Celsius with $\pm 2\%$ precision according to the packaging. Manual inspection shows that the OMAP4460 maintains the highest temperature among the components found on the board. To make the disturbance caused by shifts in temperature smaller, a heat-sink was mounted on top of the SoC and a fan aimed towards the board to further increase the heat dissipation of the entire board.

Figure 4 shows the power consumption and temperature of the OMAP4460 for heavy load program on a single core. Similarly, Figure 5 shows the power consumption and temperature of the OMAP during a run of the same program as in Figure 4, but with the added cooling.

The initially low values are the measured values of the idling system. Each of the current values used in the graphs was calculated as the median of 1000 current samples to enhance the readability of the graph. The total duration of the measurements was roughly 110 seconds. As is seen in the current-readings in Figure 4, the current drawn may increase even further if let running over a longer period of time. By comparison of Figures 4 and 5 the significant effect of the heat-sink can be seen. The result is a more stable power consumption with the extra cooling mounted.




(a) Current drawn




(b) Heat measured

Figure 5: The current usage and temperature of the Pandaboard ES with a heat-sink mounted on top of the SoC when running a single core program. The x-axis represents every $n * 1000$ current samples. 1000 samples takes approximately 1.8s to sample and collect

# 4 Generating synthetic programs

Constructing power viruses by hand is typically a very time-consuming task and requires the programmer to have a good understanding of the targeted hardware. Constructing programs and delivering results matching those created by a skilled software designer is non-trivial. The preferred outcome of this thesis was not programs which accomplished a certain operation, such as calculating prime numbers as done in the well-known MPrime tests. Instead the code was allowed to produce random results and execute virtually any instruction found in the instruction set for the hardware. The only requirements was that programs should avoid self termination and, due to the nature of this thesis, consume a significant amount of power. This makes the generation stage much easier, but some fundamental issues exists which must be handled. Primarily, the program execution must be controlled to avoid execution of illegal instructions as a result of uncontrolled branching and memory accesses must be constrained to a defined memory area to avoid segmentation faults. A solution for controlling these parameters was required. Any program which fulfilled these fundamental requirements was considered a valid synthetic program in this thesis.

The construction of programs should be automatic, and preferably based on a few parameters which the optimization algorithm can operate upon. A code generator accompanied by a suiting generalized model for assembler code can accomplish this. Under the assumption that virtually any instructions can be used, what characteristics or parameters can be found in assembler code? How can instructions be tweaked? What parameters should a synthetic generator use to synthesize a program?

In order to use a genetic algorithm for generating synthetic programs, two main issues had to be solved. First, the invention of a model which could express various aspects of a program using only binary, floating point or integer parameters. Second, a code generator operating on this model. The code generator had to make sure that all memory accesses were valid and that all branches would execute code at valid addresses. The following sections describes the parameters chosen for the code generator model in this thesis and also the algorithms with which the generator generated the assembly code.

## 4.1 Generator model

The model used for generating synthetic programs needed a number of parameters which affect the behavior and instructions of the generated program. The chosen parameters had to allow the possibility of generating code which consumes extra power. Assembler code was the language of choice for the generator as it allows for instruction level granularity during generation. By inspection of instructions supported by the hardware targeted, possible parameters can be found.

Looking at arithmetic and logic instructions, not much can be tweaked assuming that the values the instructions operate upon are to be equally treated. However, the dependency between registers can be changed and may have effect on a block of instructions. This is not limited to arithmetic and logic instructions, but applies to all instructions. Consider the following code snippet:

```
Instruction format: <operation> <result>, <source>, <source>
```

```
add r1, r2, r3
sub r4, r5, r1
```

In this case the *add* instruction needs to deliver a result before the *sub* instruction is executed, because the value in r1 is dependent on the previous instruction. This is a dependency and does generally have an impact on the flow of instruction execution.

When accessing memory using a load or store instruction, a required parameter is the memory address to access. For a single memory access, this is not interesting, but when performing multiple memory accesses in a row, the behavior of the memory accesses makes a difference. For example, when loading from one and only one address in memory repeatedly, the value is cached and thus returned very quickly. On the contrary, when stepping through memory with a considerable stride length, more cache misses will occur initially, cold misses. Depending on the total size of the memory iterated over, if the memory is big enough, caches will be filled up, cache lines evicted and look-ups in main memory performed. The point here is that depending on a combination of memory stride and size of the memory iterated over, different parts of the memory hierarchy can be triggered. Possibly, one type of behavior consumes more power than the other or strikes a sweetspot between cycles per instruction/memory access and power consumption. By creating parameters for expressing memory stride and memory size, the AI may find that sweetspot. The good thing about this is that the person generating the power-virus does not need to sort out the details.

The total length of a program affects the usage of instruction cache. The reasoning is analogue to the reasoning for the data memory. Thus a longer program could engage more of the instruction cache and potentially have effect on the power consumption. As in the case with main memory stride and size, there may exist a sweetspot in size of program and stride used within the program. Strides within the compiled binary code segment of the program can be expressed using branches jumping over chunks of no-operation instructions (NOPs) or other instructions. This can also be seen as a possibility for the AI and code generator to issue memory loads with a different stride than the stride previously mentioned for the explicit load and store instructions.

In the ARM instruction sets found on the hardware used in this thesis, there are conditional branches. The system may contain hardware for predicting the outcome of conditional branches in order to start executing the right code after the branch. For every conditional branch, mis-prediction of the execution will have instructions squashed, removed and the correct instructions executed. Thus there is a penalty for making a faulty prediction. By adding the possibility of using conditional branches in the generated code, the branch predictor may be enabled which hypothetically could consume some extra power. As a typical branch predictor keeps a history regarding whether a branch was taken and not taken, the parameter may be expressed with a number 0 to $n$ where $n$ indicates that the branch is executed $n$ times, followed by $n$ times where it is not executed. Instructions can be equipped with a conditional execution flag which decides whether the instruction is to be executed or not and can use the same parameter when generating code.

A summary and description of each of the parameters chosen to be included in the code generator model can be found below.

**Minimum register reuse distance**

Decides the minimum number of instructions to be executed between two accesses to a register (in some situations the distance cannot be reached and thus this parameter works on a best-effort basis). The register distance was parameterized in order to determine if highly interdependent code was consuming more power than non-interdependent code.

**Branch offset**

When performing a branch in the code, how far should the branch jump in memory. In this case the size of the jump is measured in a number of instructions. A branch offset of 0 means that it branches to the instruction directly after the branch-instruction.

**Code block count**

The code generator works with a sequence of instructions of a limited length, referred to as a code block. The code block count parameter defines how many times this code sequence is generated in series and has great impact on the total length of the program. The program length affects the fetching and caching of instructions.

**Memory stride**

The memory stride defines the distance between each subsequent memory access. This parameter was added to find a memory access stride which exercises different caches at suitable moments. Caches are a huge and important part of the memory hierarchy and may as such consume plenty of power when exercised.

**Memory size**

This defines the number of iterations of the entire program before the memory address is reset. The parameter affects the total amount of main memory the program can use and in turn also affect its cache usage.

**Conditional iterations**

Defines the number of iterations before changing the conditional used throughout the program. The conditional value is used by conditional instructions to set whether to perform or not perform the instruction.

**Instructions**

A list of values defining the instructions to be used for code generation. The actual representation depends on the code generation algorithm in use. See 4.2 for more information about the different code generation algorithms.

Not all parameters are independent of each other. The memory size parameter, which would appear as easy to implement and use, is not expressed in bytes but rather by a number of memory iterations. This is because the total amount of memory iterated over by a program is depending on the number of blocks, the number of memory accesses using a stride, and the stride itself. Limiting the memory size by performing a memory boundary check for every memory access instruction would be simple to implement and exact. However, this would

incur a penalty as it would require the code generator to insert extra assembler instructions for the memory boundary check at every memory accessing instruction. This would result in diluted code and potentially result in a lowered power consumption as the inserted extra instructions may not be optimal in terms of power consumption. Performing the checks at the end or beginning of every code-block would lessen the amount of checks, and have less impact on the code. The following example assumes that the number of instructions required to check and manage memory boundaries is three to five and that a code block consists of 20 instructions before inserting any instructions for checking memory boundaries. The inserted code for memory boundary checks if one instruction in the code block is a memory access would be $5/(20+5) = 0.2$, i.e. 20%. A 13% pollution can be expected if a check consists of three instructions. Thus, this is not a satisfactory solution either. By instead inserting the check in the beginning or the end of the program main loop, a lower level of pollution is achieved. The downside with this approach is the lower precision of the total memory size which can only be measured in number of program iterations.

## 4.2 Code generation methods

So far the parameters involved in the construction of the view has been outlined. Those parameters express properties and behavior of the generated code. This subsection considers how instructions are placed and how the program is created. Besides having different parameters as input to the generator, code can be generated in different ways using these parameters. The main input parameter used when deciding which instruction to put where is the instruction list. The possibility to regenerate a program from a set of code-generator parameters and always have the exact same outcome was desired as it encourages manually tweaking or experimenting with the parameters of programs. The difference between multiple complete power-virus generation runs using the same setup is introduced by the random factors of the genetic algorithm rather than using randomness in the code generator. In order to create a deterministic code generator an algorithm for "spreading" the instructions needed to be invented. Three different approaches to spreading instructions were tried in this thesis. They will be referred to as the chunked-, the interleaved- and the self-coded method. The genetic algorithm handles the same type of integer genes for all three code generation algorithms. However, once the parameters reaches the code generator the outcome differs between algorithms. Below follows a description of each of the three approaches.

**Chunked generation**

*Instructions* is a list of pairs consisting of instructions and their respective value. For every pair of instruction $i$ and value $n$ in *Instructions*, issue $n$ instructions of type $i$ in a series. This method is called chunked as each instruction appears $n$ times, in series, as a chunk. An example can be found in Figure 6.

**Interleaved generation**

*Instructions* is a list of pairs consisting of instructions and their respective value. For every pair of instruction $i$ and value $n$ in *Instructions*, if $n$ is larger than 0, issue one instruction $i$ and set $n = n - 1$. This method is

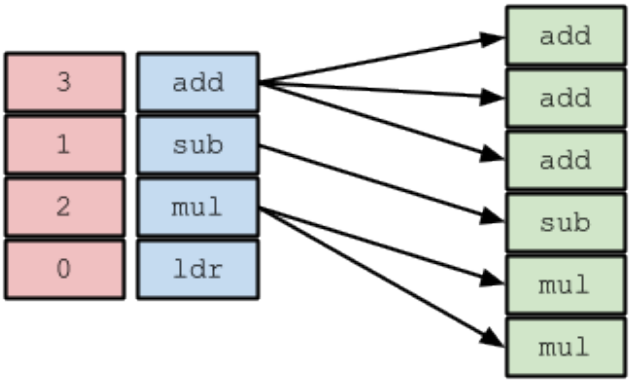



Figure 6: Chunked code generation example

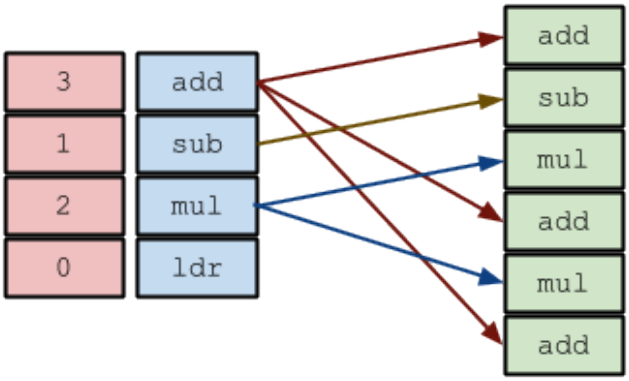



Figure 7: Interleaved code generation example

called interleaved as it interleaves the instructions, one after another, until $n$ is zero for all instructions in *Instructions*. An example can be found in Figure 7.

**Self-coded generation**

The self-coded generation algorithm uses two lists, a list of assembler instructions $l$ and a list of values $v$. Iterate over $v$, for each value $n$, emit the instruction in $l$ with index $n$. Allowing the genetic algorithm to operate with the list of values as a part of its chromosome essentially makes the genetic algorithm responsible for deciding the sequence of instructions, and as it is freely choosing what instructions to use where, it is coding on its own. Thus this method is considered to be self-coded (by the genetic algorithm). An example can be found in Figure 8.

Each of the above methods shows different behaviors. The chunked method at first appears to only serve to create chunky code sequences such as the program below which could have been generated from the instruction-value pairs: $(add, 4), (sub, 2), (mul, 1)$.

```
add  r0,  r1,  r2
```

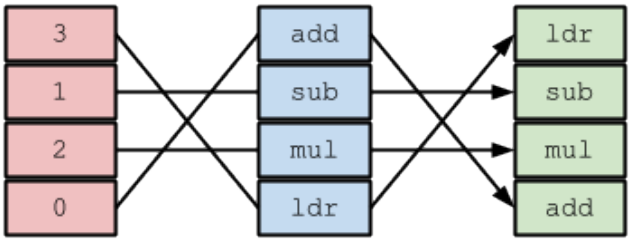



Figure 8: "Self-coded" code generation example

```
add  r3,  r4,  r5
add  r6,  r7,  r8
add  r9,  r10, r11
sub  r0,  r1,  r2
sub  r3,  r4,  r5
mul  r6,  r7,  r8
```

It is possible to generate chunks with length zero or one to make the resulting code show an interleaved pattern of instructions. This is done by limiting the instruction values to zero or one in the instruction list parameter (the leftmost values in Figure 6). By setting an instruction value to zero the instruction is ignored. The chunked code-generation offers the possibility to create an interleaved pattern where an instruction can appear more than once in its position when used this way. Thus typically, when running the genetic algorithm, the instruction values are bounded to a small range such as $[0 - 4]$ when using the chunked code generation method.

The interleaved method tries to generate code as interleaved as possible. However, this is not always the case. Consider the situation where the instruction-value pairs used for code generation are the following: $(add, 4), (sub, 1), (mul, 1)$. This would result in the following structure:

```
add  r0,  r1,  r2
sub  r3,  r4,  r5
mul  r0,  r1,  r2
add  r6,  r7,  r8
add  r9,  r10, r11
add  r12, r10, r11
```

Notice the trailing chunk, or "tail", of add-instructions. Since instructions appear different number of times, it is possible for instructions to interleave with themselves at the end of the code sequence, such as the example above. What this method offers, apart from an interleaved behavior, is the ability for the program to change throughout as instructions deplete their value during code generation. Consider the setup $(add, 6), (ldr, 6), (sub, 2), (mul, 2)$ as an example.

Both the chunked and interleaved approach make it impossible for the genetic algorithm to change the order in which instructions occur in the resulting program. The order of the instructions is decided by the order in which instruction genes is configured in the genetic algorithm. Due to this, the self-coded method was created. The self-coded code generation method gives the genetic

algorithm great possibilities to design the resulting code. Given a set of instructions, the genetic algorithm can decide exactly which instructions to have and where. This allows for a wide range of possibilities of organizing instructions such as using no instructions at all, only one multiple times in a row or, in the best case, a well matched mixture of instructions which consumes a huge amount of current.

# 5 Genetic algorithm setup

This section describes the chromosome layout and crossover operations as used in the genetic algorithm in this thesis.

## 5.1 Chromosome layout

The genetic algorithm operates on genes and chromosomes. As used in this thesis, every individual has a chromosome which represents an entire program. The genes of an individual should thus represent different properties of the program. Table 2 shows the genes used in the genetic algorithm. The values of these genes, and the properties they represent, is used as input to the code generator. Due to this it is clear that the genetic algorithm is controlling the code generator and as such has complete control over all decisions regarding the code. As can be seen in Table 2, the genetic algorithm genes are a $1:1$ mapping to the generator parameters.

| Parameter | Value range | Gene type |
|---|---|---|
| Minimum register distance | 2 - 12 instructions | Integer |
| Branch size | 0 - 4096 instructions | Integer |
| Code blocks | 1 - 10 blocks | Integer |
| Memory stride | 0 - 1024 bytes | Integer |
| Memory iterations | 1 - 40 iterations | Integer |
| Conditional iterations | 0 - 10 iterations | Integer |
| Instruction a | 0 - variable | Integer |
| Instruction b | 0 - variable | Integer |
| ... | ... | Integer |

Table 2: Genes used in the genetic algorithm

The instruction genes, referred to as instruction a and instruction b etc., and the integer value found at each corresponding position in the chromosome in the above table, are used in the code-generation methods described in section 4.2. *Instruction a* represents a specific assembler instruction, such as "add" or "sub". *Instruction a* and its corresponding integer value together form an instruction-value pair as used in both the chunky and interleaved code-generation methods. The entire list of pairs consists of: [(Instruction a, value 1), (Instruction b, value 2), ...]. The self-coded code-generation algorithm uses two separate lists: a list of instructions [Instruction a, Instruction b, ...] and a list of indices [value 1, value 2]. Code-generation using the self-coded method requires the instruction integer value range to be limited from zero to the number of instructions. If this requirement is not fulfilled it is possible for integer values to reference non-existing instructions.

The ranges for each parameter are non-trivial to choose. The register distance depends on the number of actually available general purpose registers. This may vary depending on how the code generation system decides to handle program state such as current memory address, memory stride or similar values. The programs created by the code generator use three registers to maintain internal state of such as current memory address, memory stride and memory

iterations. Thus fewer general purpose registers are left for the artificial intelligence to control. A total of 15 registers is available for use in ARM mode on all ARM architectures. These are 13 general purpose registers, the stack pointer register (sp) and the link register (lp). As the code generator is emitting assembler code the sp and lp registers can be used freely, this leaves 11 registers for free usage. Consider an instruction which uses two register operands. Using one such instruction, at most six instructions can be issued before having to reuse a register. In reality, few instructions use only two operands, and thus a value higher than six could be considered unnecessary. On the other hand, using the same parameter for controlling the register reuse distance for both ARM registers and floating point registers, makes six seem a bit low since the number of floating point registers is 32 or more, depending on hardware. Thus the distance was increased to twelve, to offer the possibility to use the larger amount of available floating point registers.

The bounds for the memory stride is volatile. The lower boundary can be adapted due to requirements of certain chosen instructions, such as aligned load/store NEON-instructions. Otherwise, considering that the cache-line size is $32B$ on the systems in this thesis, the possibility to use a stride both smaller than a cache-line and a stride much larger than a cache line should be available, perhaps ranging up to a page.

The number of genes in the Genetic algorithm can be changed to reduce or increase the size of the search space. Allowing the genetic algorithm to operate over many instructions makes it possible to generate longer and more complex code segments. On the opposite, having very few instruction genes would result in a simple and likely sub-optimal program. Inspection of the burnCortexA9 source, written specifically for ARM Cortex-A9, shows that, in the main loop, a total of ten different instructions are used. In the main loop for ssvb-cpuburn-a9, also written specifically for Cortex-A9, it appears that only four different instructions are used. In SYMPO, ten different instructions were used [9]. It would appear that a power virus can be created using a rather sparse set of instructions. The number of instruction genes used in this thesis was commonly set to between $14 - 18$ and the instructions were picked based on the results of the benchmarks as described in section 5.3. Note that an instruction gene always can assume a value which makes the code generator ignore that instruction.

Hypothetically, a small register distance and many conditional mis-predictions may have negative impact on the instruction throughput of the program. Thus, they may increase the risk of finding a suboptimal solution. When using conditional instructions, extra instructions with a conditional suffix are included in the instruction set. Thus both an increased number of genes involved and a larger instruction set is used when including the use of conditional instructions. In worst case, the result would be a lower power consumption. As a result, both runs including conditional instructions and runs excluding conditional instructions may need to be examined.

## 5.2 Genetic algorithm operators

The one-point crossover is a basic crossover operation, where typically two chromosomes are crossed. A random position within the chromosome is chosen and both chromosomes split at that index. Then one of the split parts are interchanged between the two chromosomes, creating two new chromosomes consist-

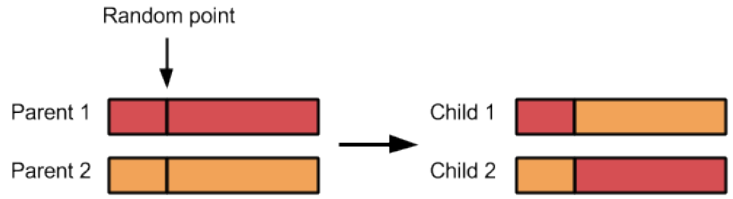



Figure 9: The one-point crossover performed on two parent chromosomes

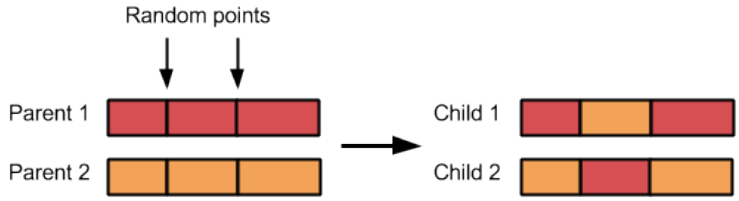



Figure 10: The two-point crossover performed on two parent chromosomes

ing of genes from both parents. Figure 9 shows the principle of the one-point crossover. The chromosomes in this thesis are vectors of integers and applying one-point crossover to two chromosomes would mean splitting and recombining the vectors as in the previously mentioned figure. In this thesis, exchanging genes between two chromosomes is equivalent to exchanging program properties and code between two programs.

Two-point crossover is similar to the one-point crossover but performs crossover using two points instead of one point. Two random points are chosen in the chromosomes and the genes between the two points is swapped creating two new chromosomes consisting of genes from both parents. Figure 10 shows the principle of the two-point crossover.

Elitism is used to keep a number of good individuals unchanged between populations. In the configuration used in this thesis, one elite individual was transferred between populations.

Multiple selection operators exists. A well known selection operator is roulette wheel selection. Using roulette wheel selection, the probability of selecting a individual is proportional to its fitness score and individuals with higher fitness score are more likely to be picked [14]. It works like a roulette wheel where every individual has its own field and the size of the field is proportional to the fitness of that individual. An example with three individuals with scores 100, 50 and 50 would render the respective probability of being selecting to 50%, 25% and 25%. Another one used in this thesis was tournament selection due to the possibility of easily tweaking the level of its elitist behavior, the selection pressure. Tournament selection picks $n$ random individuals from a population and arranges a tournament amongst the chosen ones. The individual with best fitness value among the individuals in the tournament is considered the winner. The selection pressure is controlled by the size of the tournament, $n$. Using too high selection pressure leads to a quick and premature convergence and thus a suboptimal result. Too low selection pressure leads to an unnecessarily slow convergence or no convergence. This is problem-dependent and there there is no value which fits all problems.

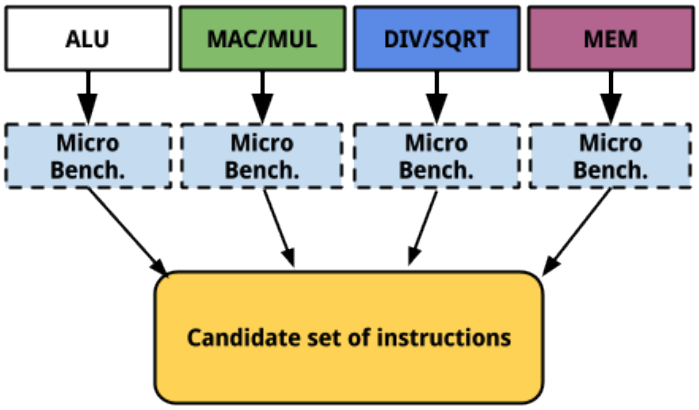



Figure 11: Proposed approach to an effective candidate instruction subset

## 5.3 Minimizing the search space

The genetic algorithm can be seen as exploring a search space of candidate solutions to a problem. A smaller search space is more likely to result in a satisfactory solution to the given problem. Thus, it is in the interest of this thesis to try to keep the search space reasonably large. Preferably, a selection of instructions should be used rather than the entire instruction set supported by the hardware. Even though the architectures investigated in this thesis are different versions of ARM, which follows the Reduced Instruction Set Computing (RISC) design strategy, using all instructions in the instruction sets when running the genetic algorithm would result in a very large search space. Instead, a strategy for picking a subset of instructions to be used in the genetic algorithm is needed. The approach presented in this section is to perform micro-benchmarks to sort out the most interesting CPU instructions in order to limit the diversity of instructions in the genetic algorithm and thus limit the size of the search space.

The instructions were grouped into four different categories. The classification was based on the separation of instructions in hardware. It is common to distinguish between ALU, MAC/MUL, DIV/SQRT and Memory in computer hardware. Even when examining the pipelines for the VFP11 co-processor three pipelines were found. These were: the multiply and accumulate pipeline, the divide and square root pipeline and finally, the load and store pipeline [1]. Due to this, the four instruction categories were chosen to be the aforementioned: ALU, MAC/MUL, DIV/SQRT and MEM. The instructions of each class were benchmarked and the candidate instruction set was based on the score of each instruction. Figure 11 shows an overview of the proposed concept.

One program was generated for each assembler instruction. The program consisted of an endless loop in which the instruction was run repeatedly 500 times. The Pandaboard has 32kB instruction cache size in ARM mode. Since $512 * 4 = 2048B$, the generated code fits in the instruction cache with its mere size. Adding the setup and tear-down used in the generated programs does not make noticeable difference in binary program size. The power consumption of each program was then measured and lists of considered instructions and

their respective current usage can be found in Appendices A, B, C, D, E, F, G, H, I and J. Due to the huge amount of instructions in the instruction sets and the rather time consuming nature of the manual work required to setup the benchmarks it is possible that some instructions and/or variations of instructions may have been overlooked during the benchmarks.

The goal of this approach was, as previously mentioned, to separate the more interesting instructions from the less interesting. In this case, as this thesis aims to generate power consuming code, the most interesting instructions were the ones consuming most power. If only the most power consuming instructions were to be used in the genetic algorithm, the search space would be smaller and the probability of finding a good solution, within a shorter amount of time, higher.

Which instructions should be included in the candidate set? Choosing the best performing instruction from each class of instructions should give a good baseline. However, there are more things to consider. In the technical reference manual for ARM1176JZF-S, three stages can be seen in the ALU pipeline. These are: shifter, ALU, saturation [1]. In order to keep a high power usage, as much hardware as possible should be used continuously. Besides including the best performing instructions from each class, including instructions to cover more stages in each pipeline would result in a instruction set with the potential of covering large if not all possible parts of the hardware. Load/store instructions do not necessarily need to be picked from every load/store class as there is only one main memory and one hierarchy of caches. If memory bandwidth is to be saturated it might as well be done by the instructions consuming most power.

# 6 Results

This section presents the results of this thesis. When generating power-viruses for both the Pandaboard ES and the Raspberry PI, the crossover rate was set to 0.80, mutation rate 0.03 and population size 80. The selection algorithm used was tournament selection and the crossover method used was two-point crossover. The results presented here were generated using the self-coded code generation method, as both the chunked and interleaved methods were found inferior during experimentation sessions. The results were generated using the PyEvolve framework version 0.6rc1 [6].

The instruction set used when generating the power-virus for the Pandaboard ES and a the first block of the resulting assembler code can be found in Appendices R respectively S. The instruction set used for generating the power-virus for the Raspberry PI and the first block of the resulting assembler code can be found in Appendices V respectively W. The sets were chosen based on the micro-benchmarks for each board and very little hardware specific knowledge was applied.

The results presented in this section will be compared to the PARSEC suite [4]. When benchmarking the Pandaboard ES, the PARSEC suite was compiled and run twice. The first run used the serial configuration of PARSEC and was pinned to one core. The second run used the default configuration and was run with at least two threads. The first run is compared against one instance of the power-virus generated in this thesis, and the second run is compared against two instances of the same power-virus. The power consumption of the PARSEC programs were measured during what PARSEC calls the "region of interest" and the mean power consumption calculated. The power-viruses were sampled for five seconds each. Table 3 shows the single core measurements from PARSEC, the CPUBurn power-viruses and a power-virus generated in this thesis.

The power-virus generated for the Pandaboard ES, referred to as gen1, displays the best result among the programs included in the comparison by a 7.1% increase when run in two instances, essentially creating a simple dual-core power-virus. When comparing gen1 to the most power consuming multi-threaded program in the PARSEC suite a 49.7% increase can be noticed and comparing the single-core version against the PARSEC suite shows a 25.1% increase. The single-core version of gen1 as compared to other single-core applications shows a 4% higher power consumption than the second best program in the comparison.

Figures showing statistics during the evolution of gen1 and the program generated for the Raspberry PI can be seen in Figure 12 respectively 13.

Additionally, a run was performed on the Pandaboard ES to evaluate the impact of conditional instructions on power consumption. An instruction set was selected using only the instructions included in gen1 but extended with the conditional versions of the instructions where possible. The instruction set can be found in Appendix T and the first block of the resulting assembler in Appendix U. This program, referred to as gen2, showed a $2 - 3mA$ increase in power consumption in comparison with gen1.

On the Raspberry PI, only the PARSEC suite was used as comparison. The resulting power-virus displays a 42.8% higher power consumption than the second most power consuming program (dedup) in the comparison.

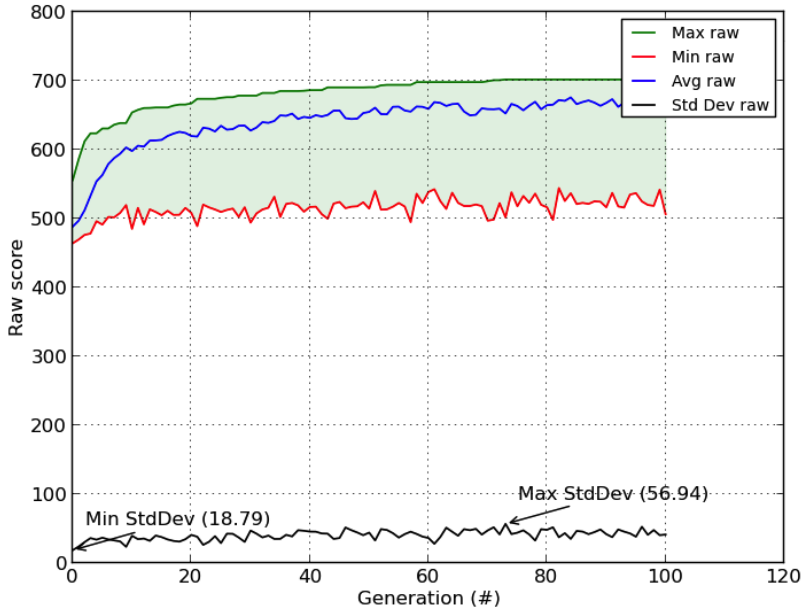



Figure 12: Generation development during evolution of gen1 program for Pandaboard ES. The scores are unscaled fitness values and therefore marked as "raw"

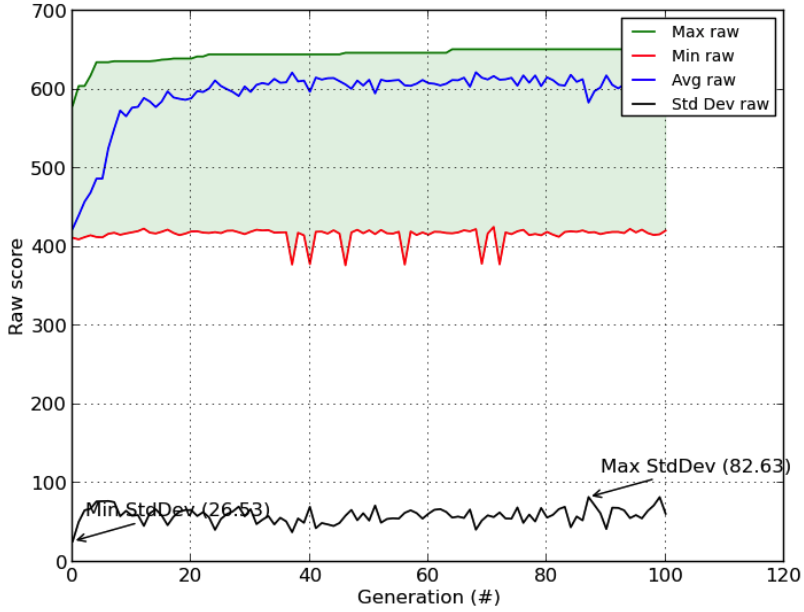



Figure 13: Generation development during evolution of program for Raspberry PI. The scores are unscaled fitness values and therefore marked as "raw"

| Program | | Mean mA |
|---|---|---|
| 1. | board idle | 380.0 |
| 2. | blacksholes | 517.6416 |
| 3. | bodytrack | 504.5744375 |
| 4. | dedup | 551.822133333 |
| 5. | ferret | 530.205941667 |
| 6. | fluidanimate | 533.8024 |
| 7. | freqmine | 550.083426316 |
| 8. | streamcluster | 555.8126 |
| 9. | swaptions | 539.910376923 |
| 10. | x264 | 555.685190909 |
| 11. | burnCortexA9 | 595.918125 |
| 12. | cpuburn | 668.547466667 |
| 13. | gen1 | 695.508533333 |

Table 3: Pandaboard ES power consumption when running programs in the PARSEC benchmark suite and power-viruses. Programs 2 to 10 belong to the PARSEC suite. These are the single core results

| Program | | Mean mA |
|---|---|---|
| 1. | board idle | 380.0 |
| 2. | blacksholes | 642.68475 |
| 3. | bodytrack | 612.147975 |
| 4. | dedup | 672.42475 |
| 5. | ferret | 726.002625 |
| 6. | fluidanimate | 688.507 |
| 7. | freqmine | 678.246556818 |
| 8. | streamcluster | 706.07871875 |
| 9. | swaptions | 718.74635 |
| 10. | x264 | 737.55425 |
| 11. | burnCortexA9 | 849.5467 |
| 12. | cpuburn | 1031.33216667 |
| 13. | gen1 | 1104.2237 |

Table 4: Pandaboard ES power consumption when running programs in the PARSEC benchmark suite and power-viruses. Programs 2 to 10 belong to the PARSEC suite. These are the dual core results

| Program | | Mean mA |
|---|---|---|
| 1. | board idle | 375.656166667 |
| 2. | blacksholes | 412.8264 |
| 3. | bodytrack | 422.804583333 |
| 4. | dedup | 452.399985714 |
| 5. | ferret | 426.655411321 |
| 6. | fluidanimate | 429.63332 |
| 7. | freqmine | 425.802023529 |
| 8. | streamcluster | 438.97835 |
| 9. | swaptions | 419.499093333 |
| 10. | x264 | 429.456785714 |
| 11. | generated program | 646.053333333 |

Table 5: Raspberry PI power consumption when running programs in the PARSEC benchmark suite and power-viruses. Programs 2 to 10 belong to the PARSEC suite

# 7 Discussion

The power consumption of the PARSEC programs fluctuated significantly between runs and the measurements found in the results thus shows the maximum average consumption of the entire region of interest over three to five runs. These measurements are pessimistic in relation to the results presented in this report but shows the difference between real applications and power-viruses.

The use of conditional instructions showed a slight increase in power usage on the Pandaboard ES. The difference was not significant enough to draw any conclusions.

Out of the three code generation methods, the self-coded delivered the best results. Both the chunked and interleaved code generation methods have issues. Both suffers from the inability to change the order of instructions. The chunked code generation issues blocks of the same instruction and the hardware may have issues with scheduling multiple identical instructions simultaneously and thus suffers from a lowered throughput. The self-coded code generation method suits the behavior of the genetic algorithm during crossover as a two-point crossover may swap a piece of code, not some abstracted values but the direct representation of a sequence of instructions. This means that the possibility of swapping entire short power consuming sequences of code is possible. The effect of the block parameter is clearer and has an almost orthogonal effect on the total number of instructions in the resulting program when using the self-coded code generation method. It is however, still possible for instruction genes to assume a value representing no instruction, essentially reducing the size of the code block by one.

The settings used for the genetic algorithm may appear as picked without any specific motivation. This is partially true and typically no configuration of the genetic algorithms suits all problems. Thus it is up to the designer to find suitable settings. The values used for generating the results were decided on through experimentation. Comparison with related work SYMPO and MAMPO shows that the values used in this thesis are quite similar to the ones used in SYMPO and MAMPO. The population size is much larger in this thesis due to the quite extensive parameter ranges. When initializing the genetic algorithm, most if not all possible values should be contained in the original population. The population size was approximated to contain all values of each gene with a reasonably high probability. After convergence, when all chromosomes are nearly identical or identical, the behavior of the genetic algorithm shows more of a random hill-climbing behavior. This is likely caused by the somewhat higher than regular mutation rate. During this stage the last percent of performance is found. One could let the genetic algorithm stop after convergence, but allowing it to run for another $20 - 30$ or more generations may increase the final score.

An experiment was conducted to evaluate the micro-benchmarks as a method of selecting good instruction candidates. By performing two runs, one run using instructions coupled with the worst scores in the micro-benchmarks and one using instructions coupled with the highest score. In both instruction sets, only one instruction was chosen from each micro-benchmark category. Both runs had an identical setup except for the chosen instructions. The result of these two runs can be found in Figure 14.

As can be seen in Figure 14, the difference is significant. The instruction set coupled with high scores shows a 13.4% higher result than the instruction set

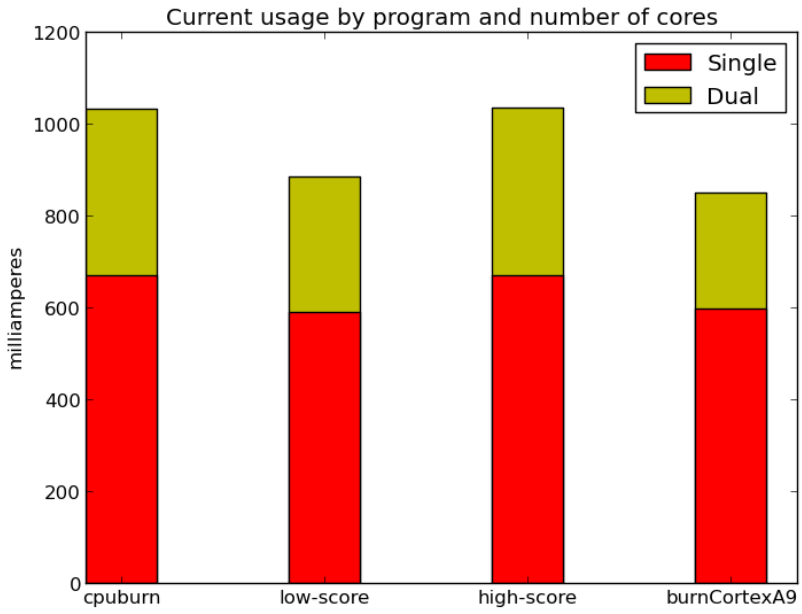



Figure 14: Total Pandaboard consumption of low scoring instructions in comparison with high scoring instructions, both single- and dual-core. Both experiments ran for 200 generations in the genetic algorithm using the same setup except for the chosen instructions.

coupled with low scores for the single core version of the program and 16.7% for the dual core versions. The program created using low score instructions is on par with burnCortexA9 and the power-virus created using high score instructions is on par with cpuburn. The results from this test indicates that the instructions included in the instruction set has impact on the power consumption of the generated power-virus. The method proposed for selection of instructions in this report is simple and, if adhered to, appears to have positive impact on the results. In circumstances where knowledge is limited about a system, this can be used as a guideline for creating an instruction set.

The micro-benchmark as proposed in this thesis does not consider the interaction between different instructions. In what ways does a multiplication interfere with an addition in the hardware? What effects does a NEON load instruction have if issued directly before or directly after a "regular" load? The micro-benchmark does not answer any of these questions and the possibilities of instruction interaction are many. The genetic algorithm, needs to sort out these problems and create programs consisting of suitable instructions in a suitable mix. By allowing the AI to solve these issues, less detailed knowledge is required from the person generating the power-virus.

The use of actual hardware for generating power-viruses does not come without issues. The measured power consumption varies depending on temperature and preferably, the hardware should be contained in a controlled environment. However, this thesis shows that using actual hardware for power-virus generation is possible. Reproduction of the exact values collected during this thesis is unlikely to succeed due to environmental circumstances and inaccuracy in the measurement equipment. Comparisons with the results in this thesis should be done using the relative differences between programs rather than the absolute values.

During experimentation on the Pandaboard ES, programs slightly better than the programs displayed in the results were created by mistake. The better scoring programs were generated using malfunctioning algorithms and better results were never reached using patched versions. This shows that the results here are less than optimal and that better results may be obtained. The broken algorithms were used for assigning registers to instructions and inserting branches. The broken register allocation had edge cases where it would always assign the last (or highest) register, completely ignoring parameters regarding register distance. Furthermore, the it applied different behavior for different "kinds" of register allocations (even/odd/ranges/singles). The code that handled branches typically inserted a branch followed by one or multiple NOPs. It would initially count each one of the NOPs as an executed instruction and update register distances even though the NOPs were never to be executed. Thus for every branch the register allocation state was effectively reset. Note that the results of this thesis were generated using the patched versions of these algorithms.

# 8 Conclusion

The main conclusions of this thesis are as follows: (I) The genetic algorithm can be used for successful generation of power-viruses on ARM hardware and the generated programs surpass the hand-written ones considered in this thesis. If further developed, better results can be expected. (II) It is possible to use actual Pandaboard ES and Raspberry PI hardware for power-virus generation on both respective boards; fluctuations in current measurements has been noted but did not pose an obstacle during the generation process of power-viruses. (III) As has been shown, the hardware required for collecting measurements during the power-virus generation can be purchased at a low cost. (IV) Finally, the micro-benchmarks performed in this thesis indicate that some instructions consume more power than others and that using the most power consuming instructions results in better power-viruses on the Pandaboard ES and Raspberry PI.

The approach used for generating power-viruses in this thesis greatly reduces the amount of knowledge required by the person generating the power-virus compared to manually designing equivalent programs. The time required for generating power-viruses was approximated to five hours for the final programs as seen in Section 6, and the process runs without supervision once started.

# 9 Future work

This section presents subjects for possible future work related to the work done in this thesis. Two versions of Thumb exists: Thumb and Thumb-2. Thumb consists of 16-bit long instructions and Thumb-2 consists of both 16-bit and 32-bit instructions. The functionality between the regular ARM mode instruction set and the Thumb sets are mostly overlapping, but differences exist. Due to the smaller instruction size, Thumb allows to pack more instructions in less memory. Question is, a higher power consumption be gained by using the Thumb instruction sets when generating a power-virus?

It would be interesting to perform analysis of many programs with a high power-consumption and try to find re-appearing characteristics. How are they composed? What happens if instruction a is replaced with instruction b?

A computer consists of more hardware than a CPU and memory. This thesis focused on the CPU and memory but how does the approach work for other hardware, e.g. GPUs?

# References


[1] arm.com. Arm1176jzf-s technical reference manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf, 2009. [Online; accessed 14-07-2013].

[2] arm.com. Neon - arm. http://www.arm.com/products/processors/technologies/neon.php, 2013. [Online; accessed 04-09-2013.

[3] arm.org. Company profile. http://www.arm.com/about/company-profile/index.php, 2013. [Online; accessed 25-06-2013].

[4] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] Dominik Brodowski and Nico Golde. Governors in the linux kernel. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt. [Online; accessed 04-06-2013].

[6] Christian S. Perone et. al. Pyevolve 0.6rc1. http://pyevolve.sourceforge.net/0_6rc1/, 2010. [Online; accessed 11-09-2013].

[7] Raspberry Pi Foundation. Faqs. http://www.raspberrypi.org/faqs. [Online; accessed 25-06-2013].

[8] K. Ganesan and L.K. John. Maximum multicore power (mampo) - an automatic multithreaded synthetic power virus generation framework for multicore systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011.

[9] Karthik Ganesan, Jungho Jo, W. Lloyd Bircher, Dimitris Kaseridis, Zhibin Yu, and Lizy K. John. System-level max power (sympo): a systematic approach for escalating system-level power consumption using synthetic benchmarks. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 19–28, New York, NY, USA, 2010. ACM.

[10] J.L. Hennessy, D.A. Patterson, and K. Asanović. *Computer Architecture: A Quantitative Approach*. Computer Architecture: A Quantitative Approach. Morgan Kaufmann/Elsevier, 2012.

[11] Grégory Herrero. burncortexa9. http://bazaar.launchpad.net/~ubuntu-branches/ubuntu/precise/cpuburn/precise/view/head:/ARM/burnCortexA9.s, 2010. [Online; accessed 15-08-2013].

[12] Texas Instruments. Ina219 datasheet. http://www.ti.com/lit/ds/symlink/ina219.pdf, 2011. [Online; accessed 22-05-2013].

[13] Texas Instruments. Omap4460 multimedia device technical reference manual. http://www.ti.com/lit/ug/swpu235z/swpu235z.pdf, 2013. [Online; accessed 10-06-2013].

[14] Sean Luke. *Essentials of Metaheuristics*. Lulu, 2009. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[15] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[16] Pandaboard.org. Omap4460 pandaboard es system reference manual. `http://pandaboard.org/sites/default/files/board_reference/ES/Panda_Board_Spec_DOC-21054_REV0_1.pdf`, 2011. [Online; accessed 10-06-2013].

[17] Siarhei Siamashka. ssvb-cpuburn-a9.s. `http://cloud.github.com/downloads/ssvb/ssvb.github.com/ssvb-cpuburn-a9.S`, 2012. [Online; accessed 12-06-2013].

# Appendix A Pandaboard benchmark (VFP ALU)

Table 6: Pandaboard micro-benchmark listing for instructions in the VFP ALU class

| Milliamperes | Instruction |
|---|---|
| 466.6 | vmov rd, vfpss |
| 478.95 | vmov.f32 vfpsd, #0.5 |
| 481.9 | vmov.f64 vfpdd, #0.5 |
| 483.8 | vcmp.f64 vfpdd, #0 |
| 484.0 | vcvtb.f32.f16 vfpsd, vfpss |
| 484.65 | vcmp.f32 vfpsd, #0 |
| 486.2 | vmov.f32 vfpsd, vfpss |
| 486.2 | vneg.f32 vfpsd, vfpss |
| 487.2 | vcmp.f32 vfpsd, vfpss |
| 487.4 | vcvtt.f32.f16 vfpsd, vfpss |
| 487.85 | vabs.f32 vfpsd, vfpss |
| 487.85 | vneg.f64 vfpdd, vfpds |
| 487.9 | vcmp.f64 vfpdd, vfpds |
| 488.3 | vcvt.s32.f64 vfpsd, vfpds |
| 488.3 | vcvtr.s32.f32 vfpsd, vfpss |
| 488.5 | vmov.f64 vfpdd, vfpds |
| 488.7 | vcvt.s32.f32 vfpsd, vfpss |
| 488.7 | vcvtr.u32.f32 vfpsd, vfpss |
| 488.9 | vcvt.f64.s32 vfpdd, vfpss |
| 489.15 | vcvt.u32.f32 vfpsd, vfpss |
| 489.55 | vabs.f64 vfpdd, vfpds |
| 489.6 | vcvtr.u32.f64 vfpsd, vfpds |
| 490.2 | vcvt.f32.f64 vfpsd, vfpds |
| 490.2 | vcvt.u32.f64 vfpsd, vfpds |
| 490.4 | vcvtb.f16.f32 vfpsd, vfpss |
| 490.4 | vcvtr.s32.f64 vfpsd, vfpds |
| 491.05 | vcvt.f64.f32 vfpdd, vfpss |
| 491.3 | vcvt.f64.u32 vfpdd, vfpss |
| 491.5 | vcvtt.f16.f32 vfpsd, vfpss |
| 492.1 | vcvt.f32.s32 vfpsd, vfpss |
| 492.5 | vcvt.f32.u32 vfpsd, vfpss |
| 493.0 | vmov vfpsd, rs |
| 495.35 | vsub.f32 vfpsd, vfpss, vfpss |
| 497.0 | vadd.f32 vfpsd, vfpss, vfpss |
| 497.05 | vsub.f64 vfpdd, vfpds, vfpds |
| 497.65 | vadd.f64 vfpdd, vfpds, vfpds |

**End of Table 6**

# Appendix B Pandaboard benchmark (VFP DIV/SQRT)

Table 7: Pandaboard micro-benchmark listing for instructions in the VFP DIV/SQRT class

| Milliamperes | Instruction |
|---|---|
| 446.95 | vsqrt.f64 vfpdd, vfpds |
| 449.5 | vsqrt.f32 vfpsd, vfpss |
| 450.65 | vdiv.f64 vfpdd, vfpds, vfpds |
| 451.6 | vdiv.f32 vfpsd, vfpss, vfpss |

**End of Table 7**

# Appendix C Pandaboard benchmark (VFP MUL)

Table 8: Pandaboard micro-benchmark listing for instructions in the VFP MUL class

| Milliamperes | Instruction |
|---|---|
| 484.0 | vmul.f64 vfpdd, vfpds, vfpds |
| 485.7 | vnmul.f64 vfpdd, vfpds, vfpds |
| 488.5 | vnmls.f64 vfpdd, vfpds, vfpds |
| 489.15 | vnmla.f64 vfpdd, vfpds, vfpds |
| 490.4 | vmla.f64 vfpdd, vfpds, vfpds |
| 492.75 | vmls.f64 vfpdd, vfpds, vfpds |
| 498.95 | vmul.f32 vfpsd, vfpss, vfpss |
| 502.35 | vnmul.f32 vfpsd, vfpss, vfpss |
| 508.95 | vmla.f32 vfpsd, vfpss, vfpss |
| 510.0 | vnmls.f32 vfpsd, vfpss, vfpss |
| 510.2 | vmls.f32 vfpsd, vfpss, vfpss |
| 511.5 | vnmla.f32 vfpsd, vfpss, vfpss |

**End of Table 8**

# Appendix D Pandaboard benchmark (MEM)

Table 9: Pandaboard micro-benchmark listing for instructions in the MEM class

| Milliamperes | Instruction |
|---|---|
| 441.4 | ldrexb rd, mem |
| 441.8 | ldrex rd, mem |
| 442.05 | ldrexh rd, mem |
| 443.95 | pld mem |
| 445.85 | ldrexd rd, rd, mem |
| 446.35 | pli _run |
| 447.4 | pli mem |
| 481.9 | strd rd, rd, mem |
| 483.2 | stm memreg, reglistsrc5 |
| 486.35 | strb rd, mem |
| 487.0 | strh rd, mem |
| 489.95 | stm memreg, reglistsrc3 |
| 490.4 | str rd, mem |
| 491.05 | ldrd rd, rd, mem |
| 491.3 | stm memreg, reglistsrc1 |
| 492.55 | stm memreg, reglistsrc4 |
| 502.75 | stm memreg, reglistsrc2 |
| 503.2 | ldrb rd, mem |
| 504.05 | ldr rd, mem |
| 504.7 | ldrh rd, mem |
| 505.15 | ldrsb rd, mem |
| 505.8 | ldrsh rd, mem |
| 508.3 | ldm memreg, reglistdst1 |
| 508.95 | strb rd, memstep |
| 510.05 | strh rd, memstep |
| 511.05 | ldm memreg, reglistdst5 |
| 511.7 | strd rd, rd, memstep |
| 512.1 | strd rd, rd, mem, #32 |
| 512.55 | str rd, memstep |
| 512.75 | ldrd rd, rd, mem, #32 |
| 513.8 | ldrd rd, rd, memstep |
| 515.5 | ldm memreg, reglistdst3 |
| 517.05 | strh rd, mem, #32 |
| 517.7 | strb rd, mem, #32 |
| 519.2 | str rd, mem, #32 |
| 521.3 | ldm memreg, reglistdst4 |
| 525.1 | ldrb rd, memstep |
| 525.95 | ldrsb rd, memstep |
| 526.6 | ldrsh rd, memstep |
| 527.3 | ldrh rd, memstep |
| 527.5 | ldm memreg, reglistdst2 |
| 527.9 | ldr rd, memstep |
| 528.3 | ldrsb rd, mem, #32 |

**Table 9 - continues on next page**

**Table 9 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 532.2 | ldrb rd, mem, #32 |
| 532.6 | ldrsh rd, mem, #32 |
| 534.5 | ldrh rd, mem, #32 |
| 535.6 | ldr rd, mem, #32 |

**End of Table 9**

# Appendix E Pandaboard benchmark (ALU)

Table 10: Pandaboard micro-benchmark listing for instructions in the ALU class

| Milliamperes | Instruction |
|---|---|
| 463.15 | tst rd, rs, asr rs |
| 463.75 | tst rd, rs, ror rs |
| 463.8 | cmn rd, rs, asr rs |
| 464.0 | cmn rd, #128 |
| 464.0 | teq rd, rs, lsl rs |
| 464.0 | teq rd, rs, ror rs |
| 464.2 | cmp rd, rs, asr rs |
| 464.4 | cmp rd, rs, lsr rs |
| 464.4 | cmp rd, rs, ror rs |
| 464.4 | mvn rd, rs, asr rs |
| 464.4 | teq rd, rs, lsr rs |
| 464.4 | tst rd, rs, lsl rs |
| 464.4 | tst rd, rs, lsr rs |
| 464.6 | cmn rd, rs, lsl rs |
| 464.6 | cmn rd, rs, ror rs |
| 464.8 | cmn rd, rs, lsr rs |
| 464.8 | cmp rd, rs, lsl rs |
| 464.8 | teq rd, rs, asr rs |
| 465.5 | cmp rd, #128 |
| 465.9 | mvn rd, rs, ror rs |
| 466.35 | bic rd, rs, asr rs |
| 466.55 | and rd, rs, ror rs |
| 466.6 | and rd, rs, lsr rs |
| 466.6 | mvn rd, rs, lsr rs |
| 466.8 | mvn rd, rs, lsl rs |
| 467.0 | add rd, rs, asr rs |
| 467.0 | orr rd, rs, asr rs |
| 467.2 | eor rd, rs, asr rs |
| 467.4 | and rd, rs, lsl rs |
| 467.6 | add rd, rs, lsr rs |
| 467.6 | bic rd, rs, ror rs |
| 467.8 | and rd, rs, asr rs |
| 467.8 | eor rd, rs, ror rs |
| 467.85 | sbc rd, rs, asr rs |
| 468.05 | eor rd, rs, lsr rs |
| 468.05 | sub rd, rs, ror rs |
| 468.05 | tst rd, #128 |
| 468.25 | adc rd, rs, asr rs |
| 468.3 | add rd, rs, ror rs |
| 468.3 | orr rd, rs, ror rs |
| 468.45 | sub rd, rs, asr rs |
| 468.5 | adc rd, rs, ror rs |
| 468.7 | add rd, rs, lsl rs |

**Table 10 - continues on next page**

**Table 10 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 468.7 | orr rd, rs, lsr rs |
| 468.7 | sbc rd, rs, ror rs |
| 468.9 | eor rd, rs, lsl rs |
| 469.1 | adc rd, rs, lsr rs |
| 469.1 | bic rd, rs, lsl rs |
| 469.1 | bic rd, rs, lsr rs |
| 469.1 | orr rd, rs, lsl rs |
| 469.1 | rsb rd, rs, lsl rs |
| 469.1 | rsb rd, rs, lsr rs |
| 469.1 | rsb rd, rs, ror rs |
| 469.1 | rsc rd, rs, lsl rs |
| 469.1 | sbc rd, rs, lsl rs |
| 469.1 | sub rd, rs, lsr rs |
| 469.3 | rsc rd, rs, lsr rs |
| 469.3 | sbc rd, rs, lsr rs |
| 469.35 | adc rd, rs, lsl rs |
| 469.5 | rsc rd, rs, ror rs |
| 469.55 | rsb rd, rs, asr rs |
| 469.55 | teq rd, #128 |
| 469.75 | rsc rd, rs, asr rs |
| 469.95 | sub rd, rs, lsl rs |
| 470.4 | usat16 rd, #15, rs |
| 471.05 | usat rd, #31, rs, lsl #30 |
| 471.65 | usat rd, #31, rs |
| 472.3 | cmn rd, rs |
| 473.35 | cmp rd, rs |
| 474.65 | mov rd, #128 |
| 475.05 | mov rd, #128 |
| 475.5 | ssat rd, #31, rs |
| 475.5 | ssat16 rd, #15, rs |
| 475.7 | usat rd, #31, rs, asr #30 |
| 475.95 | mvn rd, #128 |
| 476.4 | uxtab rd, rs, rs |
| 476.8 | uxtab rd, rs, rs, ror #16 |
| 478.05 | uxtah rd, rs, rs |
| 478.1 | ssat rd, #31, rs, lsl #30 |
| 478.3 | sxtab rd, rs, rs |
| 478.3 | uxtah rd, rs, rs, ror #16 |
| 478.7 | teq rd, rs |
| 479.35 | uxtab16 rd, rs, rs |
| 479.55 | tst rd, rs |
| 479.8 | uxtab16 rd, rs, rs, ror #16 |
| 480.0 | sxtab16 rd, rs, rs |
| 480.2 | sxtah rd, rs, rs |
| 480.6 | ssub8 rd, rs, rs |
| 480.8 | ssat rd, #31, rs, asr #30 |
| 481.05 | sxtab rd, rs, rs, ror #16 |

**Table 10 - continues on next page**

**Table 10 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 481.25 | sxtah rd, rs, rs, ror #16 |
| 481.45 | sadd8 rd, rs, rs |
| 481.9 | usub16 rd, rs, rs |
| 482.1 | qsub rd, rs, rs |
| 482.1 | sxtab16 rd, rs, rs, ror #16 |
| 482.3 | and rd, #128 |
| 482.3 | sadd16 rd, rs, rs |
| 483.2 | uadd16 rd, rs, rs |
| 483.4 | ssub16 rd, rs, rs |
| 483.6 | usub8 rd, rs, rs |
| 484.0 | uadd8 rd, rs, rs |
| 484.2 | usad8 rd, rs, rs |
| 485.75 | qadd rd, rs, rs |
| 487.4 | usax rd, rs, rs |
| 487.45 | uasx rd, rs, rs |
| 488.05 | qdadd rd, rs, rs |
| 488.3 | movt rd, #128 |
| 488.7 | sasx rd, rs, rs |
| 488.9 | ssax rd, rs, rs |
| 489.35 | clz rd, rs |
| 490.6 | qdsub rd, rs, rs |
| 494.25 | ubfx rd, rs, #16, #8 |
| 494.7 | add rd, rs, #128 |
| 494.9 | add rd, rs, #1024 |
| 495.3 | bfc rd, #16, #8 |
| 495.3 | bic rd, #128 |
| 495.5 | rsb rd, rs, #128 |
| 495.75 | orr rd, #128 |
| 495.75 | uxtb rd, rs |
| 495.95 | adc rd, rs, #128 |
| 495.95 | eor rd, #128 |
| 496.4 | adc rd, rs, #1024 |
| 496.4 | rsc rd, rs, #128 |
| 496.4 | sub rd, rs, #128 |
| 496.8 | uxtb rd, rs, ror #16 |
| 497.0 | sub rd, rs, #1024 |
| 497.2 | usada8 rd, rs, rs, rs |
| 497.2 | uxtb16 rd, rs, ror #16 |
| 498.3 | uxth rd, rs, ror #16 |
| 498.9 | sbc rd, rs, #128 |
| 498.9 | uxtb16 rd, rs |
| 498.95 | uxth rd, rs |
| 499.4 | mov rd, rs |
| 500.2 | mvn rd, rs |
| 501.5 | rev rd, rs |
| 502.75 | rbit rd, rs |
| 503.4 | rev16 rd, rs |

**Table 10 - continues on next page**

**Table 10 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 504.9 | sxth rd, rs |
| 505.1 | sxtb rd, rs, ror #16 |
| 505.3 | sxtb rd, rs |
| 505.55 | revsh rd, rs |
| 505.55 | sxth rd, rs, ror #16 |
| 506.6 | sxtb16 rd, rs, ror #16 |
| 506.8 | sxtb16 rd, rs |
| 507.5 | pkhtb rd, rs, rs |
| 507.5 | sbfx rd, rs, #16, #8 |
| 507.9 | orr rd, rs |
| 508.1 | uhsub8 rd, rs, rs |
| 508.3 | and rd, rs |
| 508.7 | bic rd, rs |
| 510.4 | uqadd16 rd, rs, rs |
| 510.65 | qsub8 rd, rs, rs |
| 511.05 | qadd8 rd, rs, rs |
| 511.1 | eor rd, rs |
| 511.1 | sel rd, rs, rs |
| 511.3 | shsub8 rd, rs, rs |
| 511.7 | uqsub8 rd, rs, rs |
| 512.35 | pkhbt rd, rs, rs, lsl #16 |
| 512.6 | qadd16 rd, rs, rs |
| 512.8 | uhsub16 rd, rs, rs |
| 513.0 | uhadd8 rd, rs, rs |
| 513.2 | shsub16 rd, rs, rs |
| 513.45 | pkhbt rd, rs, rs |
| 513.6 | sub rd, rs, rs |
| 513.8 | mov rd, rs, lsl rs |
| 513.85 | shadd8 rd, rs, rs |
| 513.85 | uqadd8 rd, rs, rs |
| 514.05 | lsl rd, rs, rs |
| 514.05 | qsub16 rd, rs, rs |
| 514.05 | sbc rd, rs, rs |
| 514.3 | lsr rd, rs, rs |
| 514.3 | rsc rd, rs, rs |
| 514.5 | adc rd, rs, rs |
| 515.1 | bfi rd, rs, #16, #8 |
| 515.3 | uqsub16 rd, rs, rs |
| 515.75 | uhadd16 rd, rs, rs |
| 515.95 | mov rd, rs, lsr rs |
| 516.0 | asr rd, rs, rs |
| 516.0 | rsb rd, rs, rs |
| 516.4 | shadd16 rd, rs, rs |
| 517.7 | add rd, rs, rs |
| 517.9 | uqsax rd, rs, rs |
| 518.55 | mov rd, rs, asr rs |
| 518.55 | qsax rd, rs, rs |

**Table 10 - continues on next page**

**Table 10 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 519.0 | uhsax rd, rs, rs |
| 520.25 | mov rd, rs, ror rs |
| 520.9 | shsax rd, rs, rs |
| 521.3 | ror rd, rs, rs |
| 521.3 | shasx rd, rs, rs |
| 522.6 | uhasx rd, rs, rs |
| 522.75 | pkhtb rd, rs, rs, asr #16 |
| 523.85 | qasx rd, rs, rs |
| 524.5 | uqasx rd, rs, rs |

**End of Table 10**

# Appendix F Pandaboard benchmark (MUL)

Table 11: Pandaboard micro-benchmark listing for instructions in the MUL class

| Milliamperes | Instruction |
|---|---|
| 462.3 | mul rd, rs, rs |
| 463.1 | smmul rd, rs, rs |
| 475.7 | smull rd, rd, rs, rs |
| 475.95 | umull rd, rd, rs, rs |
| 477.2 | mla rd, rs, rs, rs |
| 477.65 | smmls rd, rs, rs, rs |
| 478.1 | mls rd, rs, rs, rs |
| 478.1 | smmla rd, rs, rs, rs |
| 481.0 | smlalbt rd, rd, rs, rs |
| 482.1 | smulbt rd, rs, rs |
| 482.5 | smlaltb rd, rd, rs, rs |
| 484.4 | smlsld rd, rd, rs, rs |
| 484.45 | smlald rd, rd, rs, rs |
| 485.1 | smultb rd, rs, rs |
| 487.0 | umaal rd, rd, rs, rs |
| 487.45 | smuad rd, rs, rs |
| 487.65 | umlal rd, rd, rs, rs |
| 488.3 | smlal rd, rd, rs, rs |
| 488.3 | smulwb rd, rs, rs |
| 490.6 | smulwt rd, rs, rs |
| 492.75 | smusd rd, rs, rs |
| 494.25 | smlabt rd, rs, rs, rs |
| 494.9 | smlawt rd, rs, rs, rs |
| 498.1 | smlawb rd, rs, rs, rs |
| 498.5 | smlatb rd, rs, rs, rs |
| 501.7 | smlad rd, rs, rs, rs |
| 502.35 | smlsd rd, rs, rs, rs |

**End of Table 11**

# Appendix G Pandaboard benchmark (NEON ALU)

Table 12: Pandaboard micro-benchmark listing for instructions in the NEON ALU class

| Milliamperes | Instruction |
|---|---|
| 475.9 | vtbl.8 vfpdd, doublelistdst4, vfpds |
| 476.15 | vtbl.8 vfpdd, quadlistdst2, vfpds |
| 476.6 | vtbl.8 vfpdd, doublelistdst3, vfpds |
| 476.6 | vtbx.8 vfpdd, doublelistdst4, vfpds |
| 477.0 | vtbx.8 vfpdd, doublelistdst3, vfpds |
| 478.25 | vtbx.8 vfpdd, quadlistdst2, vfpds |
| 481.0 | vdup.8 vfpdd, vfpds[1] |
| 481.9 | vceq.f32 vfpdd, vfpds, #0 |
| 482.7 | vadd.f32 vfpqd, vfpqs, vfpqs |
| 482.95 | vcgt.f32 vfpqd, vfpqs, vfpqs |
| 483.15 | vmax.f32 vfpqd, vfpqs, vfpqs |
| 483.4 | vceq.f32 vfpqd, vfpqs, #0 |
| 483.4 | vmovn.i64 vfpdd, vfpqs |
| 483.6 | vcvt.u32.f32 vfpqd, vfpqs, #32 |
| 483.8 | vmovn.i32 vfpdd, vfpqs |
| 483.8 | vneg.f32 vfpdd, vfpds |
| 484.2 | vabd.f32 vfpqd, vfpqs, vfpqs |
| 484.2 | vcvt.u32.f32 vfpqd, vfpqs |
| 484.2 | vmin.f32 vfpqd, vfpqs, vfpqs |
| 484.4 | vcvt.s32.f32 vfpqd, vfpqs |
| 484.65 | vabs.f32 vfpdd, vfpds |
| 484.85 | vdup.8 vfpqd, vfpds[1] |
| 485.1 | vabs.f32 vfpqd, vfpqs |
| 485.3 | vcvt.f32.s32 vfpqd, vfpqs |
| 485.3 | vcvt.s32.f32 vfpqd, vfpqs, #32 |
| 485.3 | vneg.f32 vfpqd, vfpqs |
| 485.5 | vacgt.f32 vfpqd, vfpqs, vfpqs |
| 485.5 | vcvt.f32.u32 vfpqd, vfpqs |
| 485.75 | vzip.32 vfpqd, vfpqd |
| 486.2 | vcvt.f32.s32 vfpqd, vfpqs, #32 |
| 486.4 | vcvt.f32.u32 vfpqd, vfpqs, #32 |
| 486.6 | vzip.8 vfpqd, vfpqd |
| 486.8 | vtbl.8 vfpdd, doublelistdst2, vfpds |
| 486.8 | vuzp.32 vfpqd, vfpqd |
| 487.0 | vcvt.f32.f16 vfpqd, vfpds |
| 487.45 | vzip.16 vfpqd, vfpqd |
| 487.65 | vtbx.8 vfpdd, doublelistdst2, vfpds |
| 487.85 | vabd.f32 vfpqd, vfpqs, vfpqs |
| 487.9 | vmovn.i16 vfpdd, vfpqs |
| 488.1 | vceq.f32 vfpdd, vfpds, vfpds |
| 488.1 | vneg.f64 vfpdd, vfpds |
| 488.5 | vpmin.f32 vfpdd, vfpds, vfpds |

**Table 12 - continues on next page**

**Table 12 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 488.5 | vuzp.8 vfpqd, vfpqd |
| 488.7 | vshl.u16 vfpqd, vfpqd, vfpqd |
| 488.9 | vmov.i32 vfpqd, #12 |
| 488.9 | vrshr.u8 vfpdd, vfpds, #4 |
| 488.9 | vshl.u8 vfpqd, vfpqd, vfpqd |
| 488.9 | vuzp.16 vfpqd, vfpqd |
| 489.1 | vmin.f32 vfpdd, vfpds, vfpds |
| 489.15 | vtbl.8 vfpdd, doublelistdst1, vfpds |
| 489.35 | vrshr.u16 vfpdd, vfpds, #4 |
| 489.35 | vsub.f32 vfpqd, vfpqs, vfpqs |
| 489.6 | vabd.f32 vfpdd, vfpds, vfpds |
| 489.6 | vsub.f32 vfpdd, vfpds, vfpds |
| 489.8 | vbic.i16 vfpdd, #0xab00 |
| 489.8 | vmov.i16 vfpqd, #12 |
| 490.0 | vabd.f32 vfpdd, vfpds, vfpds |
| 490.0 | vbic.i32 vfpdd, #0xab000000 |
| 490.2 | vshr.u8 vfpdd, vfpds, #4 |
| 490.4 | vadd.f32 vfpdd, vfpds, vfpds |
| 490.4 | vmax.f32 vfpdd, vfpds, vfpds |
| 490.4 | vtbx.8 vfpdd, doublelistdst1, vfpds |
| 490.6 | vacgt.f32 vfpdd, vfpds, vfpds |
| 490.6 | vclz.s32 vfpqd, vfpqs |
| 490.6 | vmov.f32 vfpqd, #-0.328125 |
| 490.6 | vrshr.s16 vfpdd, vfpds, #4 |
| 490.6 | vrshr.s8 vfpdd, vfpds, #4 |
| 490.8 | vcgt.f32 vfpdd, vfpds, vfpds |
| 490.85 | vqrshl.u8 vfpqd, vfpqd, vfpqd |
| 490.85 | vshr.s16 vfpdd, vfpds, #4 |
| 491.05 | vbif vfpqd, vfpqs, vfpqs |
| 491.25 | vbit vfpqd, vfpqs, vfpqs |
| 491.25 | vpadd.f32 vfpdd, vfpds, vfpds |
| 491.3 | vbic.i64 vfpdd, #0x000000ab000000ab |
| 491.3 | vcls.s32 vfpqd, vfpqs |
| 491.5 | vshr.u16 vfpdd, vfpds, #4 |
| 491.65 | vcls.s32 vfpdd, vfpds |
| 491.7 | vacge.f32 vfpdd, vfpds, vfpds |
| 491.7 | vsli.32 vfpdd, vfpds, #4 |
| 491.7 | vsli.64 vfpdd, vfpds, #4 |
| 491.9 | vbic.i8 vfpdd, #0x00 |
| 491.9 | vclz.s16 vfpdd, vfpds |
| 491.9 | vclz.s32 vfpdd, vfpds |
| 491.9 | vshr.s8 vfpdd, vfpds, #4 |
| 491.9 | vuzp.16 vfpdd, vfpdd |
| 491.9 | vuzp.32 vfpdd, vfpdd |
| 492.1 | vext.8 vfpdd, vfpds, vfpds, #4 |
| 492.1 | vneg.s8 vfpdd, vfpds |
| 492.1 | vqrshl.s16 vfpqd, vfpqd, vfpqd |

**Table 12 - continues on next page**

**Table 12 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 492.1 | vswp vfpdd, vfpds |
| 492.1 | vzip.32 vfpdd, vfpdd |
| 492.3 | vcge.f32 vfpdd, vfpds, vfpds |
| 492.3 | vclz.s8 vfpdd, vfpds |
| 492.3 | vpmax.f32 vfpdd, vfpds, vfpds |
| 492.3 | vsri.8 vfpdd, vfpds, #4 |
| 492.3 | vtrn.8 vfpdd, vfpdd |
| 492.3 | vzip.16 vfpdd, vfpdd |
| 492.35 | vmov.i8 vfpqd, #12 |
| 492.35 | vshr.s32 vfpdd, vfpds, #4 |
| 492.5 | vshr.u32 vfpdd, vfpds, #4 |
| 492.5 | vshr.u64 vfpdd, vfpds, #4 |
| 492.5 | vshrn.i64 vfpdd, vfpqs, #4 |
| 492.55 | vcnt.8 vfpdd, vfpds |
| 492.55 | vpaddl.s8 vfpdd, vfpds |
| 492.55 | vpaddl.u8 vfpdd, vfpds |
| 492.55 | vtrn.16 vfpdd, vfpdd |
| 492.75 | vclz.s16 vfpqd, vfpqs |
| 492.75 | vcvt.f16.f32 vfpdd, vfpqs |
| 492.75 | vrshr.s64 vfpdd, vfpds, #4 |
| 492.75 | vrshr.u32 vfpdd, vfpds, #4 |
| 492.75 | vrshr.u64 vfpdd, vfpds, #4 |
| 492.75 | vrshrn.i64 vfpdd, vfpqs, #4 |
| 492.95 | vsli.16 vfpdd, vfpds, #4 |
| 492.95 | vsri.8 vfpqd, vfpqs, #4 |
| 493.0 | vrsra.u16 vfpdd, vfpds, #4 |
| 493.2 | vabs.s16 vfpdd, vfpds |
| 493.2 | vqrshl.u16 vfpqd, vfpqd, vfpqd |
| 493.2 | vsri.16 vfpdd, vfpds, #4 |
| 493.4 | vand.f32 vfpqd, vfpqs, vfpqs |
| 493.4 | vcls.s16 vfpdd, vfpds |
| 493.4 | vneg.s32 vfpdd, vfpds |
| 493.4 | vrev64.32 vfpqd, vfpqs |
| 493.4 | vrshr.s32 vfpdd, vfpds, #4 |
| 493.4 | vshl.s16 vfpqd, vfpqd, vfpqd |
| 493.4 | vshr.s64 vfpdd, vfpds, #4 |
| 493.4 | vsli.8 vfpdd, vfpds, #4 |
| 493.4 | vtrn.32 vfpdd, vfpdd |
| 493.6 | vcls.s16 vfpqd, vfpqs |
| 493.6 | vclz.s8 vfpqd, vfpqs |
| 493.6 | vneg.s16 vfpdd, vfpds |
| 493.6 | vuzp.8 vfpdd, vfpdd |
| 493.8 | vrev32.16 vfpqd, vfpqs |
| 494.0 | vabs.s8 vfpdd, vfpds |
| 494.0 | vmax.u16 vfpqd, vfpqs, vfpqs |
| 494.0 | vzip.8 vfpdd, vfpdd |
| 494.2 | vcls.s8 vfpqd, vfpqs |

**Table 12 - continues on next page**

**Table 12 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 494.2 | vmax.s16 vfpqd, vfpqs, vfpqs |
| 494.2 | vmax.u32 vfpqd, vfpqs, vfpqs |
| 494.2 | vqabs.s8 vfpdd, vfpds |
| 494.2 | vrsra.u8 vfpdd, vfpds, #4 |
| 494.2 | vshrn.i32 vfpdd, vfpqs, #4 |
| 494.25 | vqrshl.s64 vfpqd, vfpqd, vfpqd |
| 494.25 | vshl.s8 vfpqd, vfpqd, vfpqd |
| 494.25 | vswp vfpqd, vfpqs |
| 494.45 | vbic vfpqd, vfpqs, vfpqs |
| 494.45 | vcls.s8 vfpdd, vfpds |
| 494.45 | vhadd.u16 vfpqd, vfpqs, vfpqs |
| 494.45 | vmax.s8 vfpqd, vfpqs, vfpqs |
| 494.45 | vmovl.u16 vfpqd, vfpds |
| 494.45 | vmovl.u32 vfpqd, vfpds |
| 494.45 | vrev16.8 vfpqd, vfpqs |
| 494.45 | vrshrn.i32 vfpdd, vfpqs, #4 |
| 494.45 | vshll.u32 vfpqd, vfpdd, #4 |
| 494.65 | vqshrn.s16 vfpdd, vfpqs, #4 |
| 494.7 | vabs.s32 vfpdd, vfpds |
| 494.7 | vbic.i32 vfpqd, #0xab000000 |
| 494.7 | vhadd.s16 vfpqd, vfpqs, vfpqs |
| 494.7 | vmov.i64 vfpqd, #0xff0000ff0000ffff |
| 494.7 | vqshl.u16 vfpqd, vfpqd, #4 |
| 494.7 | vshll.u16 vfpqd, vfpdd, #4 |
| 494.85 | vshl.u32 vfpqd, vfpqd, vfpqd |
| 494.9 | vmax.s32 vfpqd, vfpqs, vfpqs |
| 494.9 | vmin.u8 vfpqd, vfpqs, vfpqs |
| 494.9 | vqshl.s16 vfpqd, vfpqd, #4 |
| 494.9 | vqshrn.u16 vfpdd, vfpqs, #4 |
| 495.1 | vext.8 vfpqd, vfpqs, vfpqs, #4 |
| 495.1 | vhadd.s8 vfpqd, vfpqs, vfpqs |
| 495.1 | vorr vfpqd, vfpqs, vfpqs |
| 495.1 | vqmovn.s16 vfpdd, vfpqs |
| 495.1 | vqrshl.s32 vfpqd, vfpqd, vfpqd |
| 495.1 | vshll.s16 vfpqd, vfpdd, #4 |
| 495.1 | vshll.s32 vfpqd, vfpdd, #4 |
| 495.1 | vshll.u8 vfpqd, vfpdd, #4 |
| 495.1 | vshr.u8 vfpqd, vfpqs, #4 |
| 495.1 | vsli.32 vfpqd, vfpqs, #4 |
| 495.1 | vsri.64 vfpdd, vfpds, #4 |
| 495.3 | vcnt.8 vfpqd, vfpqs |
| 495.3 | vmax.u8 vfpqd, vfpqs, vfpqs |
| 495.3 | vmin.s8 vfpqd, vfpqs, vfpqs |
| 495.3 | vpaddl.s32 vfpdd, vfpds |
| 495.3 | vqrshrun.s16 vfpdd, vfpqs, #4 |
| 495.3 | vqshlu.s16 vfpqd, vfpqd, #4 |
| 495.3 | vrshr.u8 vfpqd, vfpqs, #4 |

**Table 12 - continues on next page**

**Table 12 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 495.3 | vsri.32 vfpdd, vfpds, #4 |
| 495.3 | vtrn.16 vfpqd, vfpqd |
| 495.35 | vrsra.s8 vfpdd, vfpds, #4 |
| 495.5 | vmovl.s16 vfpqd, vfpds |
| 495.5 | vmovl.u8 vfpqd, vfpds |
| 495.5 | vqabs.s16 vfpdd, vfpds |
| 495.5 | vqrshl.s8 vfpqd, vfpqd, vfpqd |
| 495.5 | vqrshrun.s32 vfpdd, vfpqs, #4 |
| 495.5 | vqshl.u8 vfpqd, vfpqd, #4 |
| 495.55 | vceq.f32 vfpqd, vfpqs, vfpqs |
| 495.75 | vhadd.u8 vfpqd, vfpqs, vfpqs |
| 495.75 | vmin.s16 vfpqd, vfpqs, vfpqs |
| 495.75 | vpaddl.u32 vfpdd, vfpds |
| 495.75 | vrsra.s16 vfpdd, vfpds, #4 |
| 495.75 | vshl.i8 vfpqd, vfpqd, #4 |
| 495.75 | vsra.s8 vfpdd, vfpds, #4 |
| 495.95 | vqshrn.s32 vfpdd, vfpqs, #4 |
| 495.95 | vsli.8 vfpqd, vfpqs, #4 |
| 496.0 | vshl.u8 vfpdd, vfpdd, vfpdd |
| 496.0 | vsra.u8 vfpdd, vfpds, #4 |
| 496.0 | vsri.16 vfpqd, vfpqs, #4 |
| 496.0 | vsri.32 vfpqd, vfpqs, #4 |
| 496.15 | vbic.i16 vfpqd, #0xab00 |
| 496.2 | vhadd.u32 vfpqd, vfpqs, vfpqs |
| 496.2 | vmin.s32 vfpqd, vfpqs, vfpqs |
| 496.2 | vmin.u16 vfpqd, vfpqs, vfpqs |
| 496.2 | vmin.u32 vfpqd, vfpqs, vfpqs |
| 496.2 | vmovl.s32 vfpqd, vfpds |
| 496.2 | vmvn.i64 vfpqd, #0xff0000ff0000ffff |
| 496.2 | vpaddl.s16 vfpdd, vfpds |
| 496.2 | vqshrn.u32 vfpdd, vfpqs, #4 |
| 496.2 | vshll.s8 vfpqd, vfpdd, #4 |
| 496.2 | vsli.64 vfpqd, vfpqs, #4 |
| 496.35 | vpaddl.u16 vfpdd, vfpds |
| 496.4 | vabd.u32 vfpqd, vfpqs, vfpqs |
| 496.4 | vhadd.s32 vfpqd, vfpqs, vfpqs |
| 496.4 | vqabs.s32 vfpdd, vfpds |
| 496.4 | vqmovun.s16 vfpdd, vfpqs |
| 496.4 | vshl.s64 vfpqd, vfpqd, vfpqd |
| 496.4 | vsra.s16 vfpdd, vfpds, #4 |
| 496.4 | vsra.u16 vfpdd, vfpds, #4 |
| 496.4 | vtrn.32 vfpqd, vfpqd |
| 496.4 | vtrn.8 vfpqd, vfpqd |
| 496.6 | vrshrn.i16 vfpdd, vfpqs, #4 |
| 496.8 | vacge.f32 vfpqd, vfpqs, vfpqs |
| 496.8 | vbic.i64 vfpqd, #0x000000ab000000ab |
| 496.8 | vcge.f32 vfpqd, vfpqs, vfpqs |

**Table 12 - continues on next page**

**Table 12 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 496.8 | vcgt.s16 vfpqd, vfpqs, vfpqs |
| 496.8 | vqmovun.s32 vfpdd, vfpqs |
| 496.8 | vqrshl.s8 vfpdd, vfpdd, vfpdd |
| 496.8 | vqrshl.u8 vfpdd, vfpdd, vfpdd |
| 496.8 | vshl.s32 vfpqd, vfpqd, vfpqd |
| 497.0 | vorr.i32 vfpqd, #0xab000000 |
| 497.0 | vorr.i64 vfpqd, #0xab000000ab000000 |
| 497.2 | vmovl.s8 vfpqd, vfpds |
| 497.2 | vqrshl.u32 vfpqd, vfpqd, vfpqd |
| 497.2 | vqshlu.s32 vfpqd, vfpqd, #4 |
| 497.25 | vabd.s16 vfpqd, vfpqs, vfpqs |
| 497.25 | vabd.s32 vfpqd, vfpqs, vfpqs |
| 497.25 | vabd.u16 vfpqd, vfpqs, vfpqs |
| 497.25 | vaddl.u8 vfpqd, vfpds, vfpds |
| 497.25 | vorr.i16 vfpqd, #0x00ab |
| 497.25 | vshl.i32 vfpqd, vfpqd, #4 |
| 497.25 | vshrn.i16 vfpdd, vfpqs, #4 |
| 497.45 | vpadal.u8 vfpdd, vfpds |
| 497.45 | vqshl.s8 vfpqd, vfpqd, #4 |
| 497.45 | vqshl.u32 vfpqd, vfpqd, #4 |
| 497.45 | vshl.i16 vfpqd, vfpqd, #4 |
| 497.45 | vshl.i64 vfpqd, vfpqd, #4 |
| 497.65 | vceq.i32 vfpqd, vfpqs, #0 |
| 497.7 | vaddl.s32 vfpqd, vfpds, vfpds |
| 497.7 | vaddl.u16 vfpqd, vfpds, vfpds |
| 497.7 | vqrshl.u16 vfpdd, vfpdd, vfpdd |
| 497.7 | vshl.u64 vfpqd, vfpqd, vfpqd |
| 497.7 | vsra.u32 vfpdd, vfpds, #4 |
| 497.85 | vqmovn.s32 vfpdd, vfpqs |
| 497.9 | vabdl.s16 vfpqd, vfpds, vfpds |
| 497.9 | vabdl.s8 vfpqd, vfpds, vfpds |
| 497.9 | vqshl.s32 vfpqd, vfpqd, #4 |
| 497.9 | vqshlu.s8 vfpqd, vfpqd, #4 |
| 497.9 | vsli.16 vfpqd, vfpqs, #4 |
| 498.1 | vabd.u16 vfpdd, vfpds, vfpds |
| 498.1 | vabd.u8 vfpqd, vfpqs, vfpqs |
| 498.1 | vabdl.u16 vfpqd, vfpds, vfpds |
| 498.1 | vaddl.s16 vfpqd, vfpds, vfpds |
| 498.1 | vaddl.u32 vfpqd, vfpds, vfpds |
| 498.1 | vorr.i8 vfpqd, #0x00 |
| 498.1 | vpaddl.s8 vfpqd, vfpqs |
| 498.1 | vpaddl.u8 vfpqd, vfpqs |
| 498.1 | vshl.s16 vfpdd, vfpdd, vfpdd |
| 498.1 | vshl.s8 vfpdd, vfpdd, vfpdd |
| 498.1 | vshr.s16 vfpqd, vfpqs, #4 |
| 498.1 | vsra.u64 vfpdd, vfpds, #4 |
| 498.1 | vsri.64 vfpqd, vfpqs, #4 |

**Table 12 - continues on next page**

**Table 12 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 498.3 | vbic.i8 vfpqd, #0x00 |
| 498.3 | vneg.s8 vfpqd, vfpqs |
| 498.3 | vrshr.s16 vfpqd, vfpqs, #4 |
| 498.3 | vrsra.u64 vfpdd, vfpds, #4 |
| 498.3 | vshr.s8 vfpqd, vfpqs, #4 |
| 498.5 | vabd.s16 vfpdd, vfpds, vfpds |
| 498.5 | vabd.s8 vfpdd, vfpds, vfpds |
| 498.5 | vaddl.s8 vfpqd, vfpds, vfpds |
| 498.5 | vhsub.u16 vfpdd, vfpds, vfpds |
| 498.5 | vmvn.i8 vfpqd, #12 |
| 498.5 | vqrshl.s16 vfpdd, vfpdd, vfpdd |
| 498.55 | vrsra.s64 vfpdd, vfpds, #4 |
| 498.55 | vshl.u16 vfpdd, vfpdd, vfpdd |
| 498.7 | vabd.s32 vfpdd, vfpds, vfpds |
| 498.7 | vabdl.u8 vfpqd, vfpds, vfpds |
| 498.7 | vqrshl.u64 vfpqd, vfpqd, vfpqd |
| 498.7 | vrshr.u16 vfpqd, vfpqs, #4 |
| 498.7 | vshl.s32 vfpdd, vfpdd, vfpdd |
| 498.9 | vshr.u16 vfpqd, vfpqs, #4 |
| 498.9 | vsra.s64 vfpdd, vfpds, #4 |
| 498.95 | vbic vfpdd, vfpds, vfpds |
| 498.95 | vceq.i16 vfpqd, vfpqs, #0 |
| 498.95 | vshl.s64 vfpdd, vfpdd, vfpdd |
| 499.15 | vabd.s8 vfpqd, vfpqs, vfpqs |
| 499.15 | vabd.u32 vfpdd, vfpds, vfpds |
| 499.15 | vcgt.s8 vfpqd, vfpqs, vfpqs |
| 499.15 | vdup.16 vfpdd, rs |
| 499.15 | vhadd.u8 vfpdd, vfpds, vfpds |
| 499.15 | vrshr.s8 vfpqd, vfpqs, #4 |
| 499.15 | vrsra.s32 vfpdd, vfpds, #4 |
| 499.15 | vrsra.u32 vfpdd, vfpds, #4 |
| 499.15 | vsra.s32 vfpdd, vfpds, #4 |
| 499.35 | vpadal.s8 vfpdd, vfpds |
| 499.35 | vqrshl.s64 vfpdd, vfpdd, vfpdd |
| 499.4 | vabdl.s32 vfpqd, vfpds, vfpds |
| 499.4 | vabdl.u32 vfpqd, vfpds, vfpds |
| 499.4 | vceq.i8 vfpqd, vfpqs, #0 |
| 499.4 | vqrshl.u32 vfpdd, vfpdd, vfpdd |
| 499.55 | vhsub.s16 vfpdd, vfpds, vfpds |
| 499.6 | vabd.u8 vfpdd, vfpds, vfpds |
| 499.6 | vqabs.s8 vfpqd, vfpqs |
| 499.6 | vqrshl.s32 vfpdd, vfpdd, vfpdd |
| 499.6 | vrshr.s32 vfpqd, vfpqs, #4 |
| 499.8 | vmvn.i16 vfpqd, #12 |
| 499.8 | vqshrn.s64 vfpdd, vfpqs, #4 |
| 499.8 | vshl.u32 vfpdd, vfpdd, vfpdd |
| 499.8 | vshr.u64 vfpqd, vfpqs, #4 |

**Table 12 - continues on next page**

**Table 12 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 499.8 | vtst.16 vfpqd, vfpqs |
| 500.0 | vadd.i8 vfpdd, vfpds, vfpds |
| 500.0 | vdup.32 vfpdd, rs |
| 500.0 | vorr.i16 vfpdd, #0x00ab |
| 500.0 | vqrshl.u64 vfpdd, vfpdd, vfpdd |
| 500.0 | vrshr.u32 vfpqd, vfpqs, #4 |
| 500.0 | vshr.s64 vfpqd, vfpqs, #4 |
| 500.0 | vshr.u32 vfpqd, vfpqs, #4 |
| 500.0 | vsub.i16 vfpdd, vfpds, vfpds |
| 500.0 | vsub.i32 vfpdd, vfpds, vfpds |
| 500.0 | vsub.i64 vfpdd, vfpds, vfpds |
| 500.0 | vsub.i8 vfpdd, vfpds, vfpds |
| 500.0 | vtst.32 vfpqd, vfpqs |
| 500.2 | vhadd.s8 vfpdd, vfpds, vfpds |
| 500.2 | vmin.u32 vfpdd, vfpds, vfpds |
| 500.2 | vorr.i32 vfpdd, #0xab000000 |
| 500.2 | vorr.i64 vfpdd, #0xab000000ab000000 |
| 500.4 | vabs.s32 vfpqd, vfpqs |
| 500.4 | vcgt.s32 vfpqd, vfpqs, vfpqs |
| 500.4 | vcgt.u8 vfpqd, vfpqs, vfpqs |
| 500.4 | vmin.u16 vfpdd, vfpds, vfpds |
| 500.4 | vrshr.u64 vfpqd, vfpqs, #4 |
| 500.4 | vshl.u64 vfpdd, vfpdd, vfpdd |
| 500.4 | vshr.s32 vfpqd, vfpqs, #4 |
| 500.6 | vmvn.i32 vfpqd, #12 |
| 500.6 | vqshrn.u64 vfpdd, vfpqs, #4 |
| 500.6 | vrshr.s64 vfpqd, vfpqs, #4 |
| 500.85 | vmin.s32 vfpdd, vfpds, vfpds |
| 500.85 | vmin.s8 vfpdd, vfpds, vfpds |
| 500.85 | vmvn vfpdd, vfpds |
| 500.85 | vneg.s16 vfpqd, vfpqs |
| 500.85 | vqrshrun.s64 vfpdd, vfpqs, #4 |
| 501.05 | vcgt.u16 vfpqd, vfpqs, vfpqs |
| 501.1 | vabs.s16 vfpqd, vfpqs |
| 501.1 | vabs.s8 vfpqd, vfpqs |
| 501.1 | vceq.i32 vfpdd, vfpds, #0 |
| 501.1 | vhsub.u32 vfpdd, vfpds, vfpds |
| 501.1 | vmin.s16 vfpdd, vfpds, vfpds |
| 501.1 | vneg.s32 vfpqd, vfpqs |
| 501.1 | vqmovun.s64 vfpdd, vfpqs |
| 501.3 | vqabs.s16 vfpqd, vfpqs |
| 501.3 | vqabs.s32 vfpqd, vfpqs |
| 501.3 | vqmovn.s64 vfpdd, vfpqs |
| 501.3 | vqshl.u64 vfpqd, vfpqd, #4 |
| 501.5 | vhsub.s32 vfpdd, vfpds, vfpds |
| 501.5 | vhsub.u8 vfpdd, vfpds, vfpds |
| 501.5 | vpaddl.u32 vfpqd, vfpqs |

**Table 12 - continues on next page**

**Table 12 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 501.5 | vraddhn.i16 vfpdd, vfpqs, vfpqs |
| 501.7 | vbsl vfpqd, vfpqs, vfpqs |
| 501.7 | vmax.u8 vfpdd, vfpds, vfpds |
| 501.7 | vmin.u8 vfpdd, vfpds, vfpds |
| 501.7 | vpadd.i8 vfpdd, vfpds, vfpds |
| 501.9 | vadd.i16 vfpdd, vfpds, vfpds |
| 501.9 | vhsub.s8 vfpdd, vfpds, vfpds |
| 501.9 | vqshlu.s64 vfpqd, vfpqd, #4 |
| 502.1 | vtst.8 vfpqd, vfpqs |
| 502.3 | vadd.i64 vfpdd, vfpds, vfpds |
| 502.3 | vorr.i8 vfpdd, #0x00 |
| 502.35 | vhadd.s16 vfpdd, vfpds, vfpds |
| 502.35 | vrsubhn.i16 vfpdd, vfpqs, vfpqs |
| 502.35 | vtst.16 vfpdd, vfpds |
| 502.55 | vhadd.u32 vfpdd, vfpds, vfpds |
| 502.55 | vmax.u16 vfpdd, vfpds, vfpds |
| 502.55 | vmax.u32 vfpdd, vfpds, vfpds |
| 502.55 | vpaddl.s32 vfpqd, vfpqs |
| 502.55 | vrsra.u8 vfpqd, vfpqs, #4 |
| 502.75 | vhadd.u16 vfpdd, vfpds, vfpds |
| 502.75 | vmax.s16 vfpdd, vfpds, vfpds |
| 502.8 | vbif vfpdd, vfpds, vfpds |
| 502.8 | vmax.s8 vfpdd, vfpds, vfpds |
| 502.95 | vceq.i16 vfpdd, vfpds, #0 |
| 503.0 | vadd.i32 vfpdd, vfpds, vfpds |
| 503.0 | vceq.i8 vfpdd, vfpds, #0 |
| 503.0 | vpaddl.u16 vfpqd, vfpqs |
| 503.2 | vand.f32 vfpdd, vfpds, vfpds |
| 503.2 | vhadd.s32 vfpdd, vfpds, vfpds |
| 503.4 | vaba.u16 vfpqd, vfpqs, vfpqs |
| 503.4 | vmax.s32 vfpdd, vfpds, vfpds |
| 503.4 | vsra.u8 vfpqd, vfpqs, #4 |
| 503.6 | vpadal.u16 vfpdd, vfpds |
| 503.6 | vpaddl.s16 vfpqd, vfpqs |
| 503.6 | vpmin.u16 vfpdd, vfpds, vfpds |
| 503.8 | vsubl.u16 vfpqd, vfpds, vfpds |
| 504.0 | vaba.u8 vfpqd, vfpqs, vfpqs |
| 504.0 | vrsra.u16 vfpqd, vfpqs, #4 |
| 504.0 | vsubhn.i16 vfpdd, vfpqs, vfpqs |
| 504.05 | vsubl.s8 vfpqd, vfpds, vfpds |
| 504.25 | vaddhn.i16 vfpdd, vfpqs, vfpqs |
| 504.25 | vrsra.s8 vfpqd, vfpqs, #4 |
| 504.25 | vsra.s8 vfpqd, vfpqs, #4 |
| 504.25 | vsubl.s16 vfpqd, vfpds, vfpds |
| 504.45 | vaba.s8 vfpqd, vfpqs, vfpqs |
| 504.45 | vrsra.s16 vfpqd, vfpqs, #4 |
| 504.45 | vtst.32 vfpdd, vfpds |

**Table 12 - continues on next page**

**Table 12 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 504.5 | vpadd.i32 vfpdd, vfpds, vfpds |
| 504.5 | vpmin.u8 vfpdd, vfpds, vfpds |
| 504.9 | vpadal.s16 vfpdd, vfpds |
| 504.9 | vpmax.s8 vfpdd, vfpds, vfpds |
| 504.9 | vsra.u16 vfpqd, vfpqs, #4 |
| 504.9 | vsub.i8 vfpqd, vfpqs, vfpqs |
| 504.9 | vsubl.u8 vfpqd, vfpds, vfpds |
| 505.1 | vpadal.u8 vfpqd, vfpqs |
| 505.3 | vhsub.s8 vfpqd, vfpqs, vfpqs |
| 505.3 | vsra.s16 vfpqd, vfpqs, #4 |
| 505.35 | vpmin.s32 vfpdd, vfpds, vfpds |
| 505.55 | vbsl vfpdd, vfpds, vfpds |
| 505.55 | vpmin.u32 vfpdd, vfpds, vfpds |
| 505.75 | vaba.s32 vfpqd, vfpqs, vfpqs |
| 505.8 | veor vfpdd, vfpds, vfpds |
| 505.95 | vadd.i32 vfpqd, vfpqs, vfpqs |
| 505.95 | veor vfpqd, vfpqs, vfpqs |
| 506.0 | vbit vfpdd, vfpds, vfpds |
| 506.0 | vpadal.u32 vfpdd, vfpds |
| 506.2 | vaba.u32 vfpqd, vfpqs, vfpqs |
| 506.2 | vpadal.s8 vfpqd, vfpqs |
| 506.4 | vadd.i16 vfpqd, vfpqs, vfpqs |
| 506.4 | vmov vfpdd, vfpds |
| 506.4 | vpadd.i16 vfpdd, vfpds, vfpds |
| 506.4 | vpmin.s16 vfpdd, vfpds, vfpds |
| 506.4 | vsra.u64 vfpqd, vfpqs, #4 |
| 506.6 | vaba.s16 vfpqd, vfpqs, vfpqs |
| 506.6 | vaba.s8 vfpdd, vfpds, vfpds |
| 506.6 | vaba.u32 vfpdd, vfpds, vfpds |
| 506.6 | vadd.i8 vfpqd, vfpqs, vfpqs |
| 506.6 | vhsub.s16 vfpqd, vfpqs, vfpqs |
| 506.6 | vhsub.u16 vfpqd, vfpqs, vfpqs |
| 506.6 | vorn vfpdd, vfpds, vfpds |
| 506.6 | vrsra.s64 vfpqd, vfpqs, #4 |
| 506.6 | vsub.i16 vfpqd, vfpqs, vfpqs |
| 506.8 | vadd.i64 vfpqd, vfpqs, vfpqs |
| 506.8 | vaddhn.i32 vfpdd, vfpqs, vfpqs |
| 506.8 | vaddhn.i64 vfpdd, vfpqs, vfpqs |
| 506.8 | vraddhn.i32 vfpdd, vfpqs, vfpqs |
| 506.8 | vrsra.s32 vfpqd, vfpqs, #4 |
| 506.8 | vrsra.u32 vfpqd, vfpqs, #4 |
| 506.8 | vrsra.u64 vfpqd, vfpqs, #4 |
| 506.8 | vrsubhn.i32 vfpdd, vfpqs, vfpqs |
| 506.8 | vsra.s64 vfpqd, vfpqs, #4 |
| 507.0 | vaba.u8 vfpdd, vfpds, vfpds |
| 507.0 | vceq.i32 vfpdd, vfpds, vfpds |
| 507.0 | vcgt.u16 vfpdd, vfpds, vfpds |

**Table 12 - continues on next page**

**Table 12 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 507.0 | vhsub.u8 vfpqd, vfpqs, vfpqs |
| 507.0 | vraddhn.i64 vfpdd, vfpqs, vfpqs |
| 507.0 | vsra.u32 vfpqd, vfpqs, #4 |
| 507.0 | vsub.i32 vfpqd, vfpqs, vfpqs |
| 507.0 | vsubhn.i64 vfpdd, vfpqs, vfpqs |
| 507.05 | vaba.s16 vfpdd, vfpds, vfpds |
| 507.05 | vpadal.s32 vfpdd, vfpds |
| 507.05 | vrsubhn.i64 vfpdd, vfpqs, vfpqs |
| 507.05 | vsubhn.i32 vfpdd, vfpqs, vfpqs |
| 507.25 | vorr vfpdd, vfpds, vfpds |
| 507.25 | vsubl.s32 vfpqd, vfpds, vfpds |
| 507.25 | vtst.8 vfpdd, vfpds |
| 507.45 | vsra.s32 vfpqd, vfpqs, #4 |
| 507.5 | vaba.s32 vfpdd, vfpds, vfpds |
| 507.5 | vaba.u16 vfpdd, vfpds, vfpds |
| 507.7 | vaddw.u8 vfpqd, vfpqs, vfpds |
| 507.7 | vsub.i64 vfpqd, vfpqs, vfpqs |
| 507.7 | vsubl.u32 vfpqd, vfpds, vfpds |
| 508.1 | vcgt.u8 vfpdd, vfpds, vfpds |
| 508.3 | vmov vfpqd, vfpqs |
| 508.5 | vpmax.u8 vfpdd, vfpds, vfpds |
| 508.95 | vabal.u16 vfpqd, vfpds, vfpds |
| 508.95 | vpmin.s8 vfpdd, vfpds, vfpds |
| 509.4 | vabal.u8 vfpqd, vfpds, vfpds |
| 509.4 | vmvn vfpqd, vfpqs |
| 509.6 | vsubw.u8 vfpqd, vfpqs, vfpds |
| 510.2 | vpmax.s32 vfpdd, vfpds, vfpds |
| 510.4 | vaddw.s8 vfpqd, vfpqs, vfpds |
| 510.4 | vdup.16 vfpqd, rs |
| 510.45 | vcgt.s16 vfpdd, vfpds, vfpds |
| 510.65 | vaddw.s16 vfpqd, vfpqs, vfpds |
| 510.65 | vcgt.u32 vfpqd, vfpqs, vfpqs |
| 510.9 | vpmax.u32 vfpdd, vfpds, vfpds |
| 510.9 | vsubw.s16 vfpqd, vfpqs, vfpds |
| 511.1 | vaddw.u16 vfpqd, vfpqs, vfpds |
| 511.3 | vaddw.s32 vfpqd, vfpqs, vfpds |
| 511.3 | vceq.i16 vfpdd, vfpds, vfpds |
| 511.3 | vpmax.s16 vfpdd, vfpds, vfpds |
| 511.5 | vaddw.u32 vfpqd, vfpqs, vfpds |
| 511.5 | vcgt.s8 vfpdd, vfpds, vfpds |
| 511.5 | vcgt.u32 vfpdd, vfpds, vfpds |
| 511.5 | vsubw.s32 vfpqd, vfpqs, vfpds |
| 511.5 | vsubw.u16 vfpqd, vfpqs, vfpds |
| 511.5 | vsubw.u32 vfpqd, vfpqs, vfpds |
| 511.7 | vsubw.s8 vfpqd, vfpqs, vfpds |
| 512.6 | vceq.i8 vfpdd, vfpds, vfpds |
| 512.6 | vpadal.u16 vfpqd, vfpqs |

**Table 12 - continues on next page**

**Table 12 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 513.0 | vcgt.s32 vfpdd, vfpds, vfpds |
| 513.4 | vabal.u32 vfpqd, vfpds, vfpds |
| 513.6 | vpadal.s16 vfpqd, vfpqs |
| 513.8 | vdup.32 vfpqd, rs |
| 513.85 | vpmax.u16 vfpdd, vfpds, vfpds |
| 514.9 | vpadal.u32 vfpqd, vfpqs |
| 515.1 | vcge.s16 vfpdd, vfpds, vfpds |
| 515.5 | vcge.u16 vfpdd, vfpds, vfpds |
| 515.5 | vpadal.s32 vfpqd, vfpqs |
| 515.75 | vorn vfpqd, vfpqs, vfpqs |
| 515.95 | vcge.s8 vfpdd, vfpds, vfpds |
| 516.0 | vcge.u8 vfpdd, vfpds, vfpds |
| 517.9 | vcge.s32 vfpdd, vfpds, vfpds |
| 517.9 | vcge.u32 vfpdd, vfpds, vfpds |
| 520.7 | vcge.u16 vfpqd, vfpqs, vfpqs |
| 520.7 | vhsub.u32 vfpqd, vfpqs, vfpqs |
| 521.1 | vcge.u8 vfpqd, vfpqs, vfpqs |
| 521.1 | vhsub.s32 vfpqd, vfpqs, vfpqs |
| 526.2 | vcge.u32 vfpqd, vfpqs, vfpqs |
| 541.3 | vcge.s8 vfpqd, vfpqs, vfpqs |
| 542.6 | vceq.i32 vfpqd, vfpqs, vfpqs |
| 544.5 | vcge.s16 vfpqd, vfpqs, vfpqs |
| 544.95 | vceq.i16 vfpqd, vfpqs, vfpqs |
| 544.95 | vcge.s32 vfpqd, vfpqs, vfpqs |
| 545.15 | vceq.i8 vfpqd, vfpqs, vfpqs |

**End of Table 12**

# Appendix H Pandaboard benchmark (NEON DIV/SQRT)

Table 13: Pandaboard micro-benchmark listing for instructions in the NEON DIV/SQRT class

| Milliamperes | Instruction |
|---|---|
| 483.8 | vrsqrte.f32 vfpdd, vfpds |
| 484.0 | vrsqrte.f32 vfpqd, vfpqs |
| 489.15 | vrsqrte.u32 vfpdd, vfpds |
| 490.0 | vrsqrte.u32 vfpqd, vfpqs |
| 491.25 | vrsqrts.f32 vfpdd, vfpds |
| 495.55 | vrsqrts.f32 vfpqd, vfpqs |

**End of Table 13**

# Appendix I Pandaboard benchmark (NEON MEM)

Table 14: Pandaboard micro-benchmark listing for instructions in the NEON MEM class

| Milliamperes | Instruction |
|---|---|
| 463.55 | vst1.64 doublelistdst4, [memreg:128] |
| 463.8 | vst2.16 doublelistdst4, [memreg:128] |
| 464.0 | vst2.8 doublelistdst4, [memreg:256] |
| 464.0 | vst2.8 doublelistdst4, [memreg:64] |
| 464.0 | vst4.16 doublelistdst4, [memreg:128] |
| 464.0 | vst4.32 doublelistdst4, [memreg:64] |
| 464.2 | vst1.8 doublelistdst4, [memreg:128] |
| 464.2 | vst2.16 doublelistdst4, [memreg:256] |
| 464.2 | vst4.32 doublelistdst4, [memreg:128] |
| 464.4 | vst1.16 doublelistdst4, [memreg:64] |
| 464.4 | vst1.32 doublelistdst4, [memreg:128] |
| 464.4 | vst1.64 doublelistdst4, [memreg:64] |
| 464.4 | vst1.8 doublelistdst4, [memreg:64] |
| 464.4 | vst2.32 doublelistdst4, [memreg:256] |
| 464.45 | vst4.16 doublelistdst4, [memreg:64] |
| 464.8 | vst1.32 doublelistdst4, [memreg:256] |
| 464.8 | vst2.32 doublelistdst4, [memreg:128] |
| 464.8 | vst4.8 doublelistdst4, [memreg:64] |
| 464.85 | vst1.64 doublelistdst4, [memreg:256] |
| 465.05 | vst1.32 doublelistdst4, [memreg:64] |
| 465.05 | vst4.8 doublelistdst4, [memreg:256] |
| 465.25 | vst1.16 doublelistdst4, [memreg:128] |
| 465.3 | vst2.16 doublelistdst4, [memreg:64] |
| 465.3 | vst4.16 doublelistdst4, [memreg:256] |
| 465.5 | vst2.8 doublelistdst4, [memreg:128] |
| 473.6 | vst4.32 doublelistdstidx40, [memreg:64] |
| 474.6 | vst4.32 doublelistdstidx40, [memreg:128] |
| 475.05 | vst1.16 doublelistdst3, [memreg:64] |
| 475.5 | vst3.32 doublelistdstidx30, [memreg] |
| 475.7 | vst4.32 doublelistdstidx40, [memreg] |
| 475.75 | vst1.32 doublelistdst3, [memreg:64] |
| 475.75 | vst1.64 doublelistdst2, [memreg:64] |
| 475.9 | vst1.16 doublelistdst2, [memreg:64] |
| 475.9 | vst1.64 doublelistdst2, [memreg:128] |
| 476.15 | vst1.64 doublelistdst3, [memreg:64] |
| 476.6 | vst2.16 doublelistdst2, [memreg:128] |
| 476.8 | vst1.8 doublelistdst3, [memreg:64] |
| 477.2 | vst1.16 doublelistdst2, [memreg:128] |
| 477.2 | vst1.32 doublelistdst2, [memreg:64] |
| 477.2 | vst1.8 doublelistdst2, [memreg:64] |
| 477.2 | vst2.32 doublelistdst2, [memreg:64] |
| 477.25 | vst2.8 doublelistdst2, [memreg:64] |

**Table 14 - continues on next page**

**Table 14 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 477.4 | vst1.32 doublelistdst2, [memreg:128] |
| 477.4 | vst2.16 doublelistdst2, [memreg:64] |
| 477.65 | vst2.32 doublelistdst2, [memreg:128] |
| 477.65 | vst2.8 doublelistdst2, [memreg:128] |
| 478.3 | vst1.8 doublelistdst2, [memreg:128] |
| 478.5 | vst1.8 doublelistdst4, [memreg:256] |
| 478.5 | vst4.32 doublelistdstidx40, alignedmemstep128 |
| 478.7 | vst3.32 doublelistdst3, [memreg:64] |
| 478.9 | vst1.16 doublelistdst4, [memreg:256] |
| 478.9 | vst3.8 doublelistdst3, [memreg:64] |
| 478.9 | vst4.32 doublelistdstidx40, alignedmemstep64 |
| 479.35 | vst3.16 doublelistdst3, [memreg:64] |
| 480.4 | vst1.64 doublelistdst3, alignedmemstep64 |
| 480.6 | vst1.32 doublelistdst3, alignedmemstep64 |
| 480.6 | vst1.8 doublelistdst3, alignedmemstep64 |
| 480.8 | vst1.16 doublelistdst3, alignedmemstep64 |
| 481.05 | vst4.32 doublelistdstidx40, memstep |
| 481.45 | vst1.8 doublelistdstidx10, [memreg] |
| 481.5 | vst4.32 doublelistdst4, [memreg:256] |
| 481.7 | vst1.16 doublelistdst2, alignedmemstep128 |
| 481.7 | vst1.16 doublelistdstidx10, [memreg] |
| 481.9 | vst1.32 doublelistdst2, alignedmemstep128 |
| 481.9 | vst3.32 doublelistdstidx30, memstep |
| 481.9 | vst3.8 doublelistdst3, alignedmemstep64 |
| 482.1 | vst1.32 doublelistdst2, alignedmemstep64 |
| 482.1 | vst1.64 doublelistdst2, alignedmemstep128 |
| 482.1 | vst3.32 doublelistdst3, alignedmemstep64 |
| 482.3 | vst2.32 doublelistdst2, alignedmemstep128 |
| 482.3 | vst2.8 doublelistdst2, alignedmemstep128 |
| 482.3 | vst3.16 doublelistdst3, alignedmemstep64 |
| 482.5 | vst2.16 doublelistdst2, alignedmemstep64 |
| 482.55 | vst2.16 doublelistdst2, alignedmemstep128 |
| 482.55 | vst2.32 doublelistdst4, [memreg:64] |
| 482.7 | vst1.64 doublelistdst2, alignedmemstep64 |
| 482.7 | vst1.8 doublelistdst2, alignedmemstep128 |
| 482.7 | vst2.32 doublelistdst2, alignedmemstep64 |
| 482.95 | vst1.16 doublelistdst2, alignedmemstep64 |
| 483.15 | vst4.8 doublelistdst4, [memreg:128] |
| 483.2 | vst1.8 doublelistdst2, alignedmemstep64 |
| 483.2 | vst2.8 doublelistdst2, alignedmemstep64 |
| 483.6 | vst1.16 doublelistdstidx10, [memreg:16] |
| 484.4 | vst1.8 doublelistdst4, alignedmemstep128 |
| 484.4 | vst2.8 doublelistdstidx20, [memreg] |
| 484.85 | vst2.8 doublelistdstidx20, [memreg:16] |
| 485.1 | vst1.32 doublelistdstidx10, [memreg] |
| 485.3 | vld4.8 doublelistdste4, [memreg] |
| 485.3 | vst4.32 doublelistdst4, alignedmemstep128 |

**Table 14 - continues on next page**

**Table 14 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 485.5 | vld3.8 doublelistdste3, [memreg] |
| 485.7 | vst1.32 doublelistdstidx10, [memreg:32] |
| 485.7 | vst2.16 doublelistdst4, alignedmemstep256 |
| 487.0 | vst3.8 doublelistdstidx30, [memreg] |
| 487.65 | vst2.16 doublelistdstidx20, [memreg:32] |
| 487.65 | vst2.32 doublelistdst4, alignedmemstep256 |
| 488.1 | vst2.16 doublelistdstidx20, [memreg] |
| 488.3 | vst4.8 doublelistdst4, alignedmemstep128 |
| 488.75 | vst2.32 doublelistdstidx20, [memreg] |
| 488.9 | vld4.16 doublelistdste4, [memreg] |
| 488.9 | vst2.8 doublelistdst4, alignedmemstep128 |
| 489.15 | vst4.8 doublelistdstidx40, [memreg] |
| 489.6 | vst4.16 doublelistdst4, alignedmemstep128 |
| 489.6 | vst4.16 doublelistdstidx40, [memreg] |
| 489.8 | vld3.16 doublelistdste3, [memreg] |
| 489.8 | vst1.32 doublelistdst4, alignedmemstep64 |
| 489.8 | vst2.32 doublelistdstidx20, [memreg:64] |
| 490.0 | vst1.16 doublelistdst1, [memreg:64] |
| 490.0 | vst1.8 doublelistdst4, alignedmemstep256 |
| 490.0 | vst2.16 doublelistdst4, alignedmemstep64 |
| 490.0 | vst2.8 doublelistdst4, alignedmemstep64 |
| 490.0 | vst4.8 doublelistdstidx40, [memreg:32] |
| 490.2 | vst3.16 doublelistdstidx30, [memreg] |
| 490.4 | vst1.64 doublelistdst1, [memreg:64] |
| 490.4 | vst1.8 doublelistdst1, [memreg:64] |
| 490.4 | vst4.16 doublelistdstidx40, [memreg:64] |
| 490.4 | vst4.32 doublelistdst4, alignedmemstep256 |
| 490.6 | vst4.8 doublelistdst4, alignedmemstep64 |
| 490.8 | vld1.16 doublelistdste1, [memreg] |
| 490.8 | vst2.16 doublelistdst4, alignedmemstep128 |
| 491.05 | vst1.32 doublelistdst1, [memreg:64] |
| 491.05 | vst1.64 doublelistdst4, alignedmemstep128 |
| 491.5 | vst2.32 doublelistdst4, alignedmemstep128 |
| 491.5 | vst4.16 doublelistdst4, alignedmemstep256 |
| 491.5 | vst4.32 doublelistdst4, alignedmemstep64 |
| 491.65 | vld1.32 doublelistdste1, [memreg] |
| 491.7 | vst4.8 doublelistdst4, alignedmemstep256 |
| 491.9 | vst2.32 doublelistdst4, alignedmemstep64 |
| 492.3 | vst1.16 doublelistdst4, alignedmemstep128 |
| 492.3 | vst4.16 doublelistdst4, alignedmemstep64 |
| 492.35 | vst1.32 doublelistdst4, alignedmemstep128 |
| 492.5 | vst2.8 doublelistdst4, alignedmemstep256 |
| 492.55 | vst1.64 doublelistdst4, alignedmemstep256 |
| 492.75 | vld4.8 doublelistdste4, memstep |
| 493.2 | vst1.16 doublelistdst4, alignedmemstep64 |
| 493.4 | vld2.8 doublelistdste2, [memreg] |
| 493.4 | vst1.32 doublelistdst4, alignedmemstep256 |

**Table 14 - continues on next page**

**Table 14 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 493.4 | vst1.8 doublelistdst4, alignedmemstep64 |
| 493.8 | vst1.64 doublelistdst4, alignedmemstep64 |
| 494.45 | vld2.16 doublelistdste2, [memreg] |
| 494.45 | vst1.16 doublelistdst4, alignedmemstep256 |
| 494.9 | vld3.8 doublelistdste3, memstep |
| 496.6 | vld4.8 doublelistdstidx40, [memreg:32] |
| 497.0 | vld1.32 doublelistdste2, [memreg] |
| 497.0 | vld4.16 doublelistdste4, memstep |
| 497.9 | vld1.16 doublelistdste2, [memreg] |
| 497.9 | vld4.8 doublelistdste4, [memreg:32] |
| 498.7 | vld2.32 doublelistdste2, [memreg] |
| 499.15 | vld4.8 doublelistdstidx40, [memreg] |
| 499.4 | vld3.16 doublelistdste3, memstep |
| 499.8 | vld3.8 doublelistdstidx30, [memreg] |
| 500.2 | vst1.8 doublelistdstidx10, memstep |
| 500.65 | vld3.16 doublelistdstidx30, [memreg] |
| 501.1 | vld4.16 doublelistdstidx40, [memreg:64] |
| 501.5 | vst1.32 doublelistdstidx10, alignedmemstep32 |
| 501.7 | vld4.16 doublelistdstidx40, [memreg] |
| 502.1 | vst1.16 doublelistdstidx10, memstep |
| 502.3 | vst2.16 doublelistdstidx20, alignedmemstep32 |
| 502.35 | vld4.16 doublelistdste4, [memreg:64] |
| 502.55 | vst2.8 doublelistdstidx20, alignedmemstep16 |
| 502.8 | vst1.16 doublelistdstidx10, alignedmemstep16 |
| 502.8 | vst1.32 doublelistdstidx10, memstep |
| 503.2 | vld1.16 doublelistdste1, memstep |
| 503.4 | vst1.64 doublelistdst1, alignedmemstep64 |
| 504.25 | vst2.8 doublelistdstidx20, memstep |
| 504.25 | vst4.8 doublelistdstidx40, alignedmemstep32 |
| 504.7 | vst2.16 doublelistdstidx20, memstep |
| 504.9 | vst2.32 doublelistdstidx20, memstep |
| 504.9 | vst4.16 doublelistdstidx40, alignedmemstep64 |
| 505.3 | vld1.32 doublelistdste1, memstep |
| 505.3 | vst1.16 doublelistdst1, alignedmemstep64 |
| 505.8 | vld2.8 doublelistdste2, memstep |
| 505.8 | vst2.32 doublelistdstidx20, alignedmemstep64 |
| 506.0 | vst3.8 doublelistdstidx30, memstep |
| 506.2 | vld4.8 doublelistdstidx40, alignedmemstep32 |
| 506.2 | vst4.8 doublelistdstidx40, memstep |
| 506.6 | vld4.8 doublelistdstidx40, memstep |
| 506.6 | vst1.32 doublelistdst1, alignedmemstep64 |
| 506.8 | vld1.32 doublelistdste2, memstep |
| 507.05 | vld1.16 doublelistdste2, memstep |
| 507.7 | vst1.8 doublelistdst1, alignedmemstep64 |
| 508.1 | vld3.32 doublelistdste3, [memreg] |
| 508.3 | vld2.16 doublelistdste2, memstep |
| 508.75 | vld4.8 doublelistdste4, alignedmemstep32 |

**Table 14 - continues on next page**

**Table 14 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 509.6 | vst3.16 doublelistdstidx30, memstep |
| 509.8 | vst4.16 doublelistdstidx40, memstep |
| 510.0 | vld1.16 doublelistdste1, [memreg:16] |
| 510.45 | vld1.8 doublelistdste1, [memreg] |
| 510.65 | vld4.16 doublelistdstidx40, alignedmemstep64 |
| 510.85 | vld4.16 doublelistdstidx40, memstep |
| 510.85 | vld4.32 doublelistdste4, [memreg] |
| 511.1 | vld3.8 doublelistdstidx30, memstep |
| 512.55 | vld1.16 doublelistdste2, [memreg:16] |
| 512.6 | vld2.32 doublelistdste2, memstep |
| 513.6 | vld1.8 doublelistdste2, [memreg] |
| 514.3 | vld3.16 doublelistdstidx30, memstep |
| 514.7 | vld4.16 doublelistdste4, alignedmemstep64 |
| 515.3 | vld1.32 doublelistdste1, [memreg:32] |
| 517.05 | vld1.16 doublelistdstidx10, [memreg] |
| 517.5 | vld1.32 doublelistdste2, [memreg:32] |
| 518.1 | vld3.32 doublelistdstidx30, [memreg] |
| 518.5 | vld1.32 doublelistdstidx10, [memreg] |
| 518.5 | vld1.8 doublelistdstidx10, [memreg] |
| 519.4 | vld3.32 doublelistdste3, memstep |
| 519.8 | vld4.32 doublelistdstidx40, [memreg:128] |
| 520.05 | vld4.32 doublelistdstidx40, [memreg:64] |
| 520.45 | vld1.64 doublelistdst1, [memreg:64] |
| 520.65 | vld1.8 doublelistdst1, [memreg:64] |
| 520.85 | vld1.16 doublelistdst1, [memreg:64] |
| 521.1 | vld1.32 doublelistdst1, [memreg:64] |
| 521.5 | vld4.32 doublelistdste4, memstep |
| 522.15 | vld3.8 doublelistdst3, [memreg:64] |
| 522.15 | vld4.32 doublelistdstidx40, [memreg] |
| 522.6 | vld3.16 doublelistdst3, [memreg:64] |
| 523.0 | vld1.32 doublelistdstidx10, [memreg:32] |
| 523.2 | vld1.16 doublelistdstidx10, [memreg:16] |
| 523.4 | vld3.32 doublelistdst3, [memreg:64] |
| 524.9 | vld2.8 doublelistdst4, [memreg:128] |
| 525.35 | vld1.64 doublelistdst4, [memreg:64] |
| 525.35 | vld1.8 doublelistdst4, [memreg:128] |
| 525.35 | vld2.8 doublelistdst4, [memreg:64] |
| 525.55 | vld1.64 doublelistdst4, [memreg:128] |
| 525.55 | vld1.8 doublelistdst4, [memreg:64] |
| 525.55 | vld2.8 doublelistdstidx20, [memreg] |
| 525.75 | vld1.16 doublelistdst4, [memreg:256] |
| 525.8 | vld2.16 doublelistdstidx20, [memreg:32] |
| 525.8 | vld4.32 doublelistdste4, [memreg:128] |
| 526.0 | vld1.64 doublelistdst4, [memreg:256] |
| 526.2 | vld1.8 doublelistdst4, [memreg:256] |
| 526.2 | vld2.16 doublelistdst4, [memreg:64] |
| 526.2 | vld2.8 doublelistdst4, [memreg:256] |

**Table 14 - continues on next page**

**Table 14 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 526.4 | vld4.32 doublelistdst4, [memreg:128] |
| 526.4 | vld4.32 doublelistdste4, [memreg:64] |
| 526.6 | vld4.32 doublelistdst4, [memreg:256] |
| 526.85 | vld4.32 doublelistdstidx40, alignedmemstep128 |
| 526.85 | vld4.32 doublelistdstidx40, alignedmemstep64 |
| 527.05 | vld2.16 doublelistdst4, [memreg:128] |
| 527.05 | vld2.16 doublelistdst4, [memreg:256] |
| 527.1 | vld1.32 doublelistdst4, [memreg:256] |
| 527.3 | vld2.32 doublelistdst4, [memreg:256] |
| 527.3 | vld4.16 doublelistdst4, [memreg:256] |
| 527.5 | vld1.16 doublelistdst4, [memreg:128] |
| 527.5 | vld4.32 doublelistdst4, [memreg:64] |
| 527.7 | vld1.16 doublelistdst4, [memreg:64] |
| 527.7 | vld1.32 doublelistdst4, [memreg:128] |
| 527.9 | vld1.32 doublelistdst4, [memreg:64] |
| 527.9 | vld2.32 doublelistdst4, [memreg:128] |
| 527.9 | vld2.32 doublelistdst4, [memreg:64] |
| 527.9 | vld4.16 doublelistdst4, [memreg:128] |
| 527.9 | vld4.8 doublelistdst4, [memreg:128] |
| 528.8 | vld1.8 doublelistdste2, memstep |
| 529.2 | vld4.16 doublelistdst4, [memreg:64] |
| 529.2 | vld4.8 doublelistdst4, [memreg:64] |
| 529.4 | vld4.8 doublelistdst4, [memreg:256] |
| 530.25 | vld2.8 doublelistdstidx20, [memreg:16] |
| 530.45 | vld1.8 doublelistdste1, memstep |
| 530.9 | vld4.16 doublelistdst4, alignedmemstep64 |
| 531.3 | vld2.16 doublelistdstidx20, [memreg] |
| 531.3 | vld3.16 doublelistdst3, alignedmemstep64 |
| 531.3 | vld3.8 doublelistdst3, alignedmemstep64 |
| 531.3 | vldr vfpsd, mem |
| 532.15 | vld2.16 doublelistdst4, alignedmemstep256 |
| 532.2 | vld2.16 doublelistdst4, alignedmemstep128 |
| 532.4 | vld1.8 doublelistdst4, alignedmemstep128 |
| 532.4 | vld2.16 doublelistdst4, alignedmemstep64 |
| 532.4 | vld3.32 doublelistdst3, alignedmemstep64 |
| 532.6 | vld1.32 doublelistdst4, alignedmemstep64 |
| 532.6 | vld2.32 doublelistdst4, alignedmemstep128 |
| 532.8 | vld1.32 doublelistdste1, alignedmemstep32 |
| 532.8 | vld1.8 doublelistdst4, alignedmemstep256 |
| 532.8 | vld1.8 doublelistdst4, alignedmemstep64 |
| 532.8 | vld2.8 doublelistdst4, alignedmemstep128 |
| 533.0 | vld1.32 doublelistdst4, alignedmemstep256 |
| 533.0 | vld1.64 doublelistdst4, alignedmemstep64 |
| 533.0 | vld2.8 doublelistdst4, alignedmemstep256 |
| 533.2 | vld1.64 doublelistdst4, alignedmemstep128 |
| 533.25 | vld1.16 doublelistdste1, alignedmemstep16 |
| 533.4 | vld1.32 doublelistdst4, alignedmemstep128 |

**Table 14 - continues on next page**

**Table 14 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 533.4 | vld2.32 doublelistdst4, alignedmemstep256 |
| 533.4 | vld2.32 doublelistdst4, alignedmemstep64 |
| 533.45 | vld1.64 doublelistdst4, alignedmemstep256 |
| 533.65 | vld1.16 doublelistdst4, alignedmemstep256 |
| 533.65 | vld4.32 doublelistdst4, alignedmemstep256 |
| 533.85 | vld1.16 doublelistdste2, alignedmemstep16 |
| 533.9 | vld2.8 doublelistdst4, alignedmemstep64 |
| 534.1 | vld1.16 doublelistdst4, alignedmemstep128 |
| 534.1 | vld4.32 doublelistdst4, alignedmemstep128 |
| 534.1 | vld4.8 doublelistdst4, alignedmemstep64 |
| 534.3 | vld4.32 doublelistdst4, alignedmemstep64 |
| 534.5 | vld1.16 doublelistdst4, alignedmemstep64 |
| 534.7 | vld3.32 doublelistdstidx30, memstep |
| 534.7 | vld4.16 doublelistdst4, alignedmemstep128 |
| 534.7 | vld4.16 doublelistdst4, alignedmemstep256 |
| 534.95 | vld4.8 doublelistdst4, alignedmemstep128 |
| 535.1 | vld4.8 doublelistdst4, alignedmemstep256 |
| 535.15 | vld4.32 doublelistdstidx40, memstep |
| 535.55 | vld1.32 doublelistdste2, alignedmemstep32 |
| 536.4 | vld1.32 doublelistdst3, [memreg:64] |
| 536.65 | vld2.32 doublelistdstidx20, [memreg] |
| 537.3 | vld1.64 doublelistdst1, alignedmemstep64 |
| 537.7 | vld2.32 doublelistdstidx20, [memreg:64] |
| 537.9 | vld4.32 doublelistdste4, alignedmemstep128 |
| 538.15 | vld4.32 doublelistdste4, alignedmemstep64 |
| 538.8 | vld1.16 doublelistdst3, [memreg:64] |
| 539.2 | vld1.16 doublelistdst1, alignedmemstep64 |
| 539.4 | vld1.8 doublelistdst1, alignedmemstep64 |
| 539.6 | vld1.8 doublelistdstidx10, memstep |
| 539.8 | vld1.8 doublelistdst3, [memreg:64] |
| 540.05 | vld1.32 doublelistdstidx10, memstep |
| 540.5 | vld1.16 doublelistdstidx10, memstep |
| 540.5 | vld1.64 doublelistdst3, [memreg:64] |
| 540.9 | vld1.32 doublelistdstidx10, alignedmemstep32 |
| 541.1 | vld1.16 doublelistdstidx10, alignedmemstep16 |
| 541.3 | vld1.32 doublelistdst1, alignedmemstep64 |
| 543.7 | vld1.32 doublelistdst2, [memreg:128] |
| 543.7 | vld1.32 doublelistdst2, [memreg:64] |
| 543.7 | vld1.8 doublelistdst2, [memreg:64] |
| 544.05 | vld2.8 doublelistdstidx20, memstep |
| 544.3 | vld2.16 doublelistdstidx20, memstep |
| 544.3 | vld2.8 doublelistdst2, [memreg:64] |
| 544.5 | vld2.32 doublelistdst2, [memreg:128] |
| 544.55 | vld2.8 doublelistdstidx20, alignedmemstep16 |
| 544.95 | vld1.16 doublelistdst2, [memreg:64] |
| 545.15 | vld2.16 doublelistdstidx20, alignedmemstep32 |
| 545.15 | vld2.8 doublelistdst2, [memreg:128] |

**Table 14 - continues on next page**

**Table 14 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 545.35 | vld1.8 doublelistdst2, [memreg:128] |
| 545.4 | vld2.16 doublelistdst2, [memreg:64] |
| 545.4 | vld2.32 doublelistdst2, [memreg:64] |
| 545.55 | vldr vfpdd, mem |
| 545.8 | vld1.16 doublelistdst2, [memreg:128] |
| 545.8 | vld2.16 doublelistdst2, [memreg:128] |
| 546.7 | vld1.32 doublelistdst3, alignedmemstep64 |
| 546.9 | vld1.8 doublelistdst3, alignedmemstep64 |
| 547.1 | vld1.64 doublelistdst3, alignedmemstep64 |
| 547.7 | vld1.64 doublelistdst2, [memreg:128] |
| 547.9 | vld1.16 doublelistdst3, alignedmemstep64 |
| 548.6 | vld1.64 doublelistdst2, [memreg:64] |
| 552.4 | vld2.32 doublelistdstidx20, alignedmemstep64 |
| 552.4 | vld2.32 doublelistdstidx20, memstep |
| 559.2 | vld1.8 doublelistdst2, alignedmemstep128 |
| 559.25 | vld1.8 doublelistdst2, alignedmemstep64 |
| 559.4 | vld1.32 doublelistdst2, alignedmemstep64 |
| 559.4 | vld1.64 doublelistdst2, alignedmemstep64 |
| 559.45 | vld1.64 doublelistdst2, alignedmemstep128 |
| 559.65 | vld2.32 doublelistdst2, alignedmemstep128 |
| 559.65 | vld2.8 doublelistdst2, alignedmemstep128 |
| 559.9 | vld2.16 doublelistdst2, alignedmemstep128 |
| 559.9 | vld2.16 doublelistdst2, alignedmemstep64 |
| 559.9 | vld2.32 doublelistdst2, alignedmemstep64 |
| 559.9 | vld2.8 doublelistdst2, alignedmemstep64 |
| 560.5 | vld1.16 doublelistdst2, alignedmemstep128 |
| 560.5 | vld1.32 doublelistdst2, alignedmemstep128 |
| 560.9 | vld1.16 doublelistdst2, alignedmemstep64 |

**End of Table 14**

# Appendix J Pandaboard benchmark (NEON MUL)

Table 15: Pandaboard micro-benchmark listing for instructions in the NEON MUL class

| Milliamperes | Instruction |
|---|---|
| 475.7 | vmul.i32 vfpqd, vfpqs, scalarsrc0 |
| 477.0 | vmul.i32 vfpqd, vfpqs, vfpqs |
| 477.6 | vqdmulh.s32 vfpqd, vfpqs, scalarsrc0 |
| 478.5 | vqdmulh.s32 vfpqd, vfpqs, vfpqs |
| 480.6 | vmla.i32 vfpqd, vfpqs, scalarsrc0 |
| 482.95 | vmul.f32 vfpqd, vfpqs, scalarsrc0 |
| 483.4 | vrecps.f32 vfpdd, vfpds |
| 483.8 | vmul.f32 vfpqd, vfpqs, vfpqs |
| 483.8 | vrecpe.f32 vfpdd, vfpds |
| 484.85 | vmul.f32 vfpdd, vfpds, scalarsrc0 |
| 485.05 | vmla.f32 vfpdd, vfpds, scalarsrc0 |
| 485.7 | vmls.f32 vfpdd, vfpds, scalarsrc0 |
| 486.6 | vrecpe.f32 vfpqd, vfpqs |
| 487.0 | vmls.i32 vfpqd, vfpqs, scalarsrc0 |
| 487.45 | vmla.i32 vfpqd, vfpqs, vfpqs |
| 487.9 | vmls.f32 vfpdd, vfpds, vfpds |
| 488.1 | vmul.i16 vfpqd, vfpqs, vfpqs |
| 488.3 | vmla.f32 vfpdd, vfpds, vfpds |
| 488.7 | vmul.p8 vfpqd, vfpqs, vfpqs |
| 488.9 | vrecpe.u32 vfpdd, vfpds |
| 489.55 | vqdmulh.s16 vfpqd, vfpqs, vfpqs |
| 489.6 | vmul.i16 vfpqd, vfpqs, scalarsrc0 |
| 489.8 | vrecps.f32 vfpqd, vfpqs |
| 490.0 | vmla.f32 vfpqd, vfpqs, vfpqs |
| 490.0 | vmls.f32 vfpqd, vfpqs, vfpqs |
| 490.0 | vrecpe.u32 vfpqd, vfpqs |
| 490.65 | vmul.i8 vfpqd, vfpqs, vfpqs |
| 490.8 | vmla.f32 vfpqd, vfpqs, scalarsrc0 |
| 490.8 | vmls.i32 vfpqd, vfpqs, vfpqs |
| 491.05 | vmul.f32 vfpdd, vfpds, vfpds |
| 491.05 | vqdmulh.s16 vfpqd, vfpqs, scalarsrc0 |
| 492.3 | vmls.f32 vfpqd, vfpqs, scalarsrc0 |
| 493.4 | vqdmulh.s32 vfpdd, vfpds, scalarsrc0 |
| 495.1 | vmls.i16 vfpqd, vfpqs, scalarsrc0 |
| 495.3 | vmul.i32 vfpdd, vfpds, scalarsrc0 |
| 496.2 | vmla.i16 vfpqd, vfpqs, scalarsrc0 |
| 498.5 | vmla.i32 vfpdd, vfpds, scalarsrc0 |
| 498.9 | vqdmull.s32 vfpqd, vfpds, scalarsrc0 |
| 500.2 | vmul.i8 vfpdd, vfpds, vfpds |
| 500.6 | vmls.i8 vfpqd, vfpqs, vfpqs |
| 500.65 | vmul.i16 vfpdd, vfpds, scalarsrc0 |
| 501.1 | vqdmulh.s16 vfpdd, vfpds, scalarsrc0 |

**Table 15 - continues on next page**

**Table 15 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 501.5 | vmull.u8 vfpqd, vfpds, vfpds |
| 501.9 | vmull.s8 vfpqd, vfpds, vfpds |
| 502.1 | vqdmull.s16 vfpqd, vfpds, vfpds |
| 502.35 | vmul.i32 vfpdd, vfpds, vfpds |
| 502.55 | vmla.i8 vfpqd, vfpqs, vfpqs |
| 502.55 | vmls.i16 vfpqd, vfpqs, vfpqs |
| 502.8 | vmul.p8 vfpdd, vfpds, vfpds |
| 502.95 | vqdmull.s16 vfpqd, vfpds, scalarsrc0 |
| 503.0 | vmull.u32 vfpqd, vfpds, vfpds |
| 503.2 | vqdmull.s32 vfpqd, vfpds, vfpds |
| 503.4 | vmull.s32 vfpqd, vfpds, vfpds |
| 503.6 | vmull.s16 vfpqd, vfpds, vfpds |
| 503.6 | vmull.u16 vfpqd, vfpds, vfpds |
| 503.8 | vmla.i16 vfpqd, vfpqs, vfpqs |
| 504.25 | vmls.i32 vfpdd, vfpds, scalarsrc0 |
| 504.25 | vqdmlsl.s16 vfpqd, vfpds, vfpds |
| 504.25 | vqdmulh.s32 vfpdd, vfpds, vfpds |
| 504.5 | vqdmlal.s16 vfpqd, vfpds, vfpds |
| 505.1 | vqdmulh.s16 vfpdd, vfpds, vfpds |
| 505.3 | vmls.i8 vfpdd, vfpds, vfpds |
| 505.3 | vmls.u8 vfpdd, vfpds, vfpds |
| 505.8 | vmla.i16 vfpdd, vfpds, scalarsrc0 |
| 505.8 | vmla.i8 vfpdd, vfpds, vfpds |
| 506.0 | vmul.i16 vfpdd, vfpds, vfpds |
| 506.6 | vqdmlal.s32 vfpqd, vfpds, scalarsrc0 |
| 506.8 | vmla.i32 vfpdd, vfpds, vfpds |
| 506.85 | vmls.i16 vfpdd, vfpds, scalarsrc0 |
| 507.05 | vmlal.s8 vfpqd, vfpds, vfpds |
| 507.45 | vmla.i16 vfpdd, vfpds, vfpds |
| 507.45 | vmlal.u8 vfpqd, vfpds, vfpds |
| 507.5 | vmls.i16 vfpdd, vfpds, vfpds |
| 507.7 | vmlsl.u8 vfpqd, vfpds, vfpds |
| 508.1 | vmls.u16 vfpdd, vfpds, vfpds |
| 508.3 | vmlsl.s8 vfpqd, vfpds, vfpds |
| 508.95 | vmls.u32 vfpdd, vfpds, vfpds |
| 509.6 | vmlal.u16 vfpqd, vfpds, vfpds |
| 509.8 | vmls.u32 vfpdd, vfpds, vfpds |
| 510.2 | vmlsl.s16 vfpqd, vfpds, vfpds |
| 510.4 | vmlal.s16 vfpqd, vfpds, vfpds |
| 510.4 | vqdmlsl.s32 vfpqd, vfpds, scalarsrc0 |
| 510.65 | vmls.i32 vfpdd, vfpds, vfpds |
| 510.65 | vmlsl.u16 vfpqd, vfpds, vfpds |
| 510.65 | vqdmlsl.s32 vfpqd, vfpds, vfpds |
| 512.35 | vmlal.s32 vfpqd, vfpds, vfpds |
| 513.0 | vmlal.u32 vfpqd, vfpds, vfpds |
| 513.0 | vqdmlal.s16 vfpqd, vfpds, scalarsrc0 |
| 513.4 | vqdmlsl.s16 vfpqd, vfpds, scalarsrc0 |

**Table 15 - continues on next page**

**Table 15 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 514.05 | vmlsl.u32 vfpqd, vfpds, vfpds |
| 515.55 | vmlsl.s32 vfpqd, vfpds, vfpds |
| 515.95 | vqdmlal.s32 vfpqd, vfpds, vfpds |

**End of Table 15**

# Appendix K Raspberry PI benchmark (ALU)

Table 16: Raspberry PI micro-benchmark listing for instructions in the ALU class

| Milliamperes | Instruction |
|---|---|
| 398.4 | nop |
| 398.8 | mvn rd, rs, lsl rs |
| 399.45 | orr rd, rs, lsl rs |
| 399.7 | tst rd, #128 |
| 399.85 | orr rd, rs, ror rs |
| 400.1 | mrs rd, SPSR |
| 400.3 | tst rd, rs, lsl rs |
| 400.5 | mov rd, #128 |
| 400.5 | mvn rd, #128 |
| 400.7 | lsl rd, rs, rs |
| 400.95 | and rd, #128 |
| 400.95 | and rd, rs, lsl rs |
| 400.95 | teq rd, rs, asr rs |
| 401.15 | cmp rd, rs, ror rs |
| 401.35 | sbc rd, rs, asr rs |
| 401.4 | cmn rd, rs, lsl rs |
| 401.8 | and rd, rs, ror rs |
| 401.8 | rsb rd, rs, ror rs |
| 401.8 | sub rd, rs, lsr rs |
| 401.8 | usat rd, #31, rs, lsl #30 |
| 402.0 | lsr rd, rs, rs |
| 402.2 | cmn rd, rs, lsr rs |
| 402.2 | cmp rd, rs, asr rs |
| 402.2 | cmp rd, rs, lsr rs |
| 402.2 | eor rd, rs, asr rs |
| 402.2 | mrs rd, CPSR |
| 402.2 | orr rd, rs, lsr rs |
| 402.2 | rsb rd, rs, lsl rs |
| 402.25 | mov rd, rs, lsl rs |
| 402.4 | cmn rd, rs, asr rs |
| 402.4 | mvn rd, rs, asr rs |
| 402.4 | tst rd, rs, lsr rs |
| 402.45 | mvn rd, rs, lsr rs |
| 402.45 | sub rd, rs, ror rs |
| 402.6 | add rd, rs, lsl rs |
| 402.6 | asr rd, rs, rs |
| 402.6 | mov rd, rs, asr rs |
| 402.6 | mov rd, rs, lsr rs |
| 402.6 | sbc rd, rs, #128 |
| 402.6 | tst rd, rs, asr rs |
| 402.65 | orr rd, rs, asr rs |
| 402.85 | bic rd, rs, asr rs |

**Table 16 - continues on next page**

**Table 16 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 402.85 | mov rd, #128 |
| 402.85 | mov rd, rs, ror rs |
| 402.85 | ror rd, rs, rs |
| 402.85 | rsc rd, rs, #128 |
| 403.05 | teq rd, #128 |
| 403.05 | tst rd, rs, ror rs |
| 403.05 | uxth rd, rs, ror #16 |
| 403.1 | and rd, rs, lsr rs |
| 403.1 | cmp rd, rs, lsl rs |
| 403.1 | mvn rd, rs, ror rs |
| 403.1 | rsb rd, rs, #128 |
| 403.1 | teq rd, rs, lsl rs |
| 403.1 | teq rd, rs, lsr rs |
| 403.25 | cmn rd, rs, ror rs |
| 403.3 | bic rd, rs, lsr rs |
| 403.3 | rsb rd, rs, lsr rs |
| 403.5 | add rd, rs, asr rs |
| 403.5 | bic rd, rs, ror rs |
| 403.5 | rsb rd, rs, asr rs |
| 403.5 | rsc rd, rs, asr rs |
| 403.5 | sub rd, rs, asr rs |
| 403.5 | sub rd, rs, lsl rs |
| 403.5 | sxth rd, rs, ror #16 |
| 403.5 | teq rd, rs, ror rs |
| 403.5 | uxtb16 rd, rs, ror #16 |
| 403.7 | adc rd, rs, lsr rs |
| 403.7 | add rd, rs, lsr rs |
| 403.7 | and rd, rs, asr rs |
| 403.7 | eor rd, rs, lsl rs |
| 403.7 | eor rd, rs, ror rs |
| 403.7 | rsc rd, rs, lsl rs |
| 403.7 | sbc rd, rs, lsl rs |
| 403.7 | sxtb16 rd, rs, ror #16 |
| 403.9 | adc rd, rs, asr rs |
| 403.9 | adc rd, rs, lsl rs |
| 403.9 | add rd, rs, ror rs |
| 403.9 | eor rd, rs, lsr rs |
| 403.9 | rsc rd, rs, lsr rs |
| 403.9 | rsc rd, rs, ror rs |
| 403.9 | sbc rd, rs, lsr rs |
| 403.9 | sbc rd, rs, ror rs |
| 403.95 | bic rd, #128 |
| 403.95 | uxtb rd, rs |
| 404.15 | bic rd, rs, lsl rs |
| 404.15 | mov rd, rs |
| 404.55 | adc rd, rs, ror rs |
| 404.55 | sxtb16 rd, rs |

**Table 16 - continues on next page**

**Table 16 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 404.55 | uxtb rd, rs, ror #16 |
| 404.55 | uxtb16 rd, rs |
| 404.75 | adc rd, rs, #128 |
| 404.95 | usat16 rd, #15, rs |
| 405.0 | cmp rd, #128 |
| 405.0 | ssat rd, #31, rs, lsl #30 |
| 405.0 | uxth rd, rs |
| 405.2 | clz rd, rs |
| 405.4 | cmn rd, #128 |
| 405.4 | cmn rd, rs |
| 405.4 | teq rd, rs |
| 405.6 | add rd, rs, #128 |
| 405.6 | usat rd, #31, rs, asr #30 |
| 405.8 | sub rd, rs, #1024 |
| 405.8 | sxtb rd, rs, ror #16 |
| 405.85 | rev rd, rs |
| 406.0 | mvn rd, rs |
| 406.0 | usat rd, #31, rs |
| 406.0 | uxtab rd, rs, rs, ror #16 |
| 406.05 | add rd, rs, #1024 |
| 406.05 | ssat rd, #31, rs |
| 406.25 | cmp rd, rs |
| 406.25 | rrx rd, rs |
| 406.25 | ssat rd, #31, rs, asr #30 |
| 406.25 | sub rd, rs, #128 |
| 406.45 | orr rd, #128 |
| 406.45 | revsh rd, rs |
| 406.5 | cpy rd, rs |
| 406.5 | eor rd, #128 |
| 406.65 | adc rd, rs, #1024 |
| 406.65 | pkhbt rd, rs, rs, lsl #16 |
| 406.9 | ssat16 rd, #15, rs |
| 406.9 | ssubaddx rd, rs, rs |
| 406.9 | sub rd, rs, rs |
| 406.9 | sxth rd, rs |
| 407.1 | orr rd, rs |
| 407.1 | qdsub rd, rs, rs |
| 407.1 | tst rd, rs |
| 407.3 | sxtab16 rd, rs, rs, ror #16 |
| 407.35 | eor rd, rs |
| 407.35 | sxtb rd, rs |
| 407.35 | usubaddx rd, rs, rs |
| 407.55 | pkhtb rd, rs, rs, asr #16 |
| 407.75 | qaddsubx rd, rs, rs |
| 407.75 | sxtab rd, rs, rs, ror #16 |
| 407.75 | uxtab16 rd, rs, rs, ror #16 |
| 407.8 | and rd, rs |

**Table 16 - continues on next page**

**Table 16 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 408.0 | qsub16 rd, rs, rs |
| 408.0 | rev16 rd, rs |
| 408.0 | ssax rd, rs, rs |
| 408.15 | uhadd16 rd, rs, rs |
| 408.15 | usad8 rd, rs, rs |
| 408.2 | bic rd, rs |
| 408.2 | uhsub16 rd, rs, rs |
| 408.2 | uqsub8 rd, rs, rs |
| 408.4 | ssub16 rd, rs, rs |
| 408.4 | uadd16 rd, rs, rs |
| 408.4 | uhasx rd, rs, rs |
| 408.6 | usada8 rd, rs, rs, rs |
| 408.85 | qadd rd, rs, rs |
| 409.0 | shsubaddx rd, rs, rs |
| 409.0 | uhadd8 rd, rs, rs |
| 409.0 | uhsub8 rd, rs, rs |
| 409.0 | uxtab rd, rs, rs |
| 409.0 | uxtab16 rd, rs, rs |
| 409.25 | qadd8 rd, rs, rs |
| 409.25 | saddsubx rd, rs, rs |
| 409.25 | uhsax rd, rs, rs |
| 409.25 | uhsubaddx rd, rs, rs |
| 409.25 | uqadd16 rd, rs, rs |
| 409.25 | uqadd8 rd, rs, rs |
| 409.25 | uqsub16 rd, rs, rs |
| 409.25 | uxtah rd, rs, rs, ror #16 |
| 409.45 | sadd8 rd, rs, rs |
| 409.45 | sxtah rd, rs, rs, ror #16 |
| 409.45 | usub8 rd, rs, rs |
| 409.5 | add rd, rs, rs |
| 409.5 | qsubaddx rd, rs, rs |
| 409.5 | rsb rd, rs, rs |
| 409.5 | shadd16 rd, rs, rs |
| 409.5 | shsub16 rd, rs, rs |
| 409.5 | uaddsubx rd, rs, rs |
| 409.5 | uasx rd, rs, rs |
| 409.5 | uqaddsubx rd, rs, rs |
| 409.5 | usax rd, rs, rs |
| 409.5 | uxtah rd, rs, rs |
| 409.65 | ssub8 rd, rs, rs |
| 409.7 | adc rd, rs, rs |
| 409.7 | pkhtb rd, rs, rs |
| 409.7 | qadd16 rd, rs, rs |
| 409.7 | sadd16 rd, rs, rs |
| 409.7 | shadd8 rd, rs, rs |
| 409.7 | shaddsubx rd, rs, rs |
| 409.7 | shasx rd, rs, rs |

**Table 16 - continues on next page**

**Table 16 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 409.7 | shsub8 rd, rs, rs |
| 409.7 | uadd8 rd, rs, rs |
| 409.7 | uqsubaddx rd, rs, rs |
| 409.9 | pkhbt rd, rs, rs |
| 409.9 | qdadd rd, rs, rs |
| 409.9 | qsub rd, rs, rs |
| 409.9 | rsc rd, rs, rs |
| 409.9 | sbc rd, rs, rs |
| 409.9 | sel rd, rs, rs |
| 409.9 | shsax rd, rs, rs |
| 409.9 | uhaddsubx rd, rs, rs |
| 409.9 | uqasx rd, rs, rs |
| 409.9 | uqsax rd, rs, rs |
| 410.1 | qasx rd, rs, rs |
| 410.1 | sasx rd, rs, rs |
| 410.1 | sxtab rd, rs, rs |
| 410.1 | sxtab16 rd, rs, rs |
| 410.1 | sxtah rd, rs, rs |
| 410.1 | usub16 rd, rs, rs |
| 410.3 | qsub8 rd, rs, rs |
| 410.5 | qsax rd, rs, rs |

**End of Table 16**

# Appendix L Raspberry PI benchmark (MEM)

Table 17: Raspberry PI micro-benchmark listing for instructions in the MEM class

| Milliamperes | Instruction |
|---|---|
| 392.85 | pld mem |
| 417.55 | ldm memreg, reglistdst5 |
| 419.9 | ldrexh rd, mem |
| 420.1 | ldrh rd, mem |
| 420.95 | ldm memreg, reglistdst3 |
| 420.95 | ldrb rd, mem |
| 420.95 | ldrexb rd, mem |
| 421.4 | ldrsb rd, mem |
| 421.6 | ldr rd, memstep |
| 421.6 | ldr rd, mem |
| 421.6 | ldrex rd, mem |
| 421.6 | ldrsh rd, mem |
| 422.2 | ldm memreg, reglistdst1 |
| 422.2 | ldrh rd, memstep |
| 422.45 | ldrb rd, memstep |
| 422.65 | ldrsh rd, memstep |
| 424.15 | ldrsb rd, memstep |
| 424.35 | ldrsb rd, mem, #32 |
| 424.4 | ldr rd, mem, #32 |
| 424.4 | ldrb rd, mem, #32 |
| 424.4 | ldrh rd, mem, #32 |
| 424.4 | ldrsh rd, mem, #32 |
| 425.6 | ldm memreg, reglistdst4 |
| 427.8 | ldm memreg, reglistdst2 |
| 429.9 | ldrd rd, rd, mem |
| 430.3 | ldrexd rd, rd, mem |
| 431.8 | ldrd rd, rd, mem, #32 |
| 432.7 | ldrd rd, rd, memstep |
| 511.1 | strb rd, mem |
| 517.25 | strh rd, mem |
| 521.5 | strb rd, memstep |
| 522.35 | strb rd, mem, #32 |
| 525.75 | str rd, mem |
| 527.9 | stm memreg, reglistsrc1 |
| 528.55 | strh rd, memstep |
| 529.8 | strh rd, mem, #32 |
| 540.05 | str rd, memstep |
| 540.05 | str rd, mem, #32 |
| 540.7 | stm memreg, reglistsrc5 |
| 545.8 | stm memreg, reglistsrc3 |
| 551.35 | stm memreg, reglistsrc2 |
| 552.6 | strd rd, rd, mem |

**Table 17 - continues on next page**

**Table 17 - continued from previous page**

| Milliamperes | Instruction |
|---|---|
| 560.5 | strd rd, rd, memstep |
| 563.25 | strd rd, rd, mem, #32 |
| 594.6 | stm memreg, reglistsrc4 |

**End of Table 17**

# Appendix M Raspberry PI benchmark (MUL)

Table 18: Raspberry PI micro-benchmark listing for instructions in the MUL class

| Milliamperes | Instruction |
|---|---|
| 401.35 | smull rd, rd, rs, rs |
| 401.8 | umaal rd, rd, rs, rs |
| 401.8 | umull rd, rd, rs, rs |
| 402.0 | umaal rd, rd, rs, rs |
| 402.2 | smlal rd, rd, rs, rs |
| 402.65 | smmla rd, rs, rs, rs |
| 402.65 | umlal rd, rd, rs, rs |
| 403.05 | smmls rd, rs, rs, rs |
| 403.95 | smmulr rd, rs, rs |
| 404.8 | mul rd, rs, rs |
| 404.8 | smmlsr rd, rs, rs, rs |
| 404.95 | smlaldx rd, rd, rs, rs |
| 405.2 | smmlar rd, rs, rs, rs |
| 405.4 | mla rd, rs, rs, rs |
| 405.4 | smlaltb rd, rd, rs, rs |
| 405.6 | smlalbt rd, rd, rs, rs |
| 405.6 | smmul rd, rs, rs |
| 406.25 | smlsldx rd, rd, rs, rs |
| 406.7 | smlald rd, rd, rs, rs |
| 406.9 | smlsdx rd, rs, rs, rs |
| 407.1 | smlsld rd, rd, rs, rs |
| 407.35 | smultb rd, rs, rs |
| 408.2 | smlad rd, rs, rs, rs |
| 408.4 | smusd rd, rs, rs |
| 408.6 | smulbt rd, rs, rs |
| 408.8 | smlabt rd, rs, rs, rs |
| 408.8 | smulwb rd, rs, rs |
| 409.0 | smladx rd, rs, rs, rs |
| 409.05 | smlsd rd, rs, rs, rs |
| 409.05 | smultt rd, rs, rs |
| 409.7 | smlawt rd, rs, rs, rs |
| 409.7 | smulbb rd, rs, rs |
| 409.9 | smlatb rd, rs, rs, rs |
| 410.5 | smuad rd, rs, rs |
| 410.5 | smulwt rd, rs, rs |
| 410.5 | smusdx rd, rs, rs |
| 410.7 | smuadx rd, rs, rs |
| 411.2 | smlawb rd, rs, rs, rs |

**End of Table 18**

# Appendix N Raspberry PI benchmark (VFP ALU)

Table 19: Raspberry PI micro-benchmark listing for instructions in the VFP ALU class

| Milliamperes | Instruction |
|---|---|
| 398.8 | vcvtr.s32.f32 vfpsd, vfpss |
| 400.3 | vcvt.u32.f32 vfpsd, vfpss |
| 401.4 | vcvt.f32.s32 vfpsd, vfpss |
| 401.6 | vcvt.f32.u32 vfpsd, vfpss |
| 401.8 | vcvtr.u32.f64 vfpsd, vfpds |
| 402.0 | vcvtr.u32.f32 vfpsd, vfpss |
| 402.2 | vcvt.f64.f32 vfpdd, vfpss |
| 402.45 | vadd.f64 vfpdd, vfpds, vfpds |
| 402.85 | vcvt.f64.s32 vfpdd, vfpss |
| 402.85 | vcvt.s32.f32 vfpsd, vfpss |
| 403.5 | vcmp.f32 vfpsd, #0 |
| 403.7 | vcvtr.s32.f64 vfpsd, vfpds |
| 403.9 | vcmp.f64 vfpdd, #0 |
| 403.9 | vcvt.f64.u32 vfpdd, vfpss |
| 404.3 | vcvt.u32.f64 vfpsd, vfpds |
| 404.75 | vcvt.s32.f64 vfpsd, vfpds |
| 405.2 | vmov.f64 vfpdd, vfpds |
| 406.0 | vcmp.f32 vfpsd, vfpss |
| 406.0 | vmov.f32 vfpsd, vfpss |
| 406.0 | vneg.f32 vfpsd, vfpss |
| 406.25 | vsub.f64 vfpdd, vfpds, vfpds |
| 406.45 | vabs.f32 vfpsd, vfpss |
| 406.5 | vcmp.f64 vfpdd, vfpds |
| 406.7 | vabs.f64 vfpdd, vfpds |
| 406.9 | vneg.f64 vfpdd, vfpds |
| 408.6 | vadd.f32 vfpsd, vfpss, vfpss |
| 408.6 | vsub.f32 vfpsd, vfpss, vfpss |
| 411.6 | vmov rd, vfpss |
| 413.3 | vmov vfpsd, rs |
| 414.8 | vmov rd, rd, vfpds |
| 415.0 | vmov rd, rs, vfpss, vfpss |
| 415.0 | vmov vfpdd, rs, rs |
| 415.4 | vmov vfpsd, vfpsd, rs, rs |
| 418.0 | vcvt.f32.f64 vfpsd, vfpds |

**End of Table 19**

# Appendix O Raspberry PI benchmark (VFP MEM)

Table 20: Raspberry PI micro-benchmark listing for instructions in the VFP MEM class

| Milliamperes | Instruction |
|---|---|
| 418.6 | vldm memreg, singlelistdst5 |
| 422.45 | vldm memreg, singlelistdst3 |
| 422.45 | vldr vfpsd, mem |
| 423.1 | vldm memreg, singlelistdst1 |
| 425.0 | vldm memreg, doublelistdst2 |
| 425.85 | vldm memreg, singlelistdst4 |
| 427.8 | vldr vfpdd, mem |
| 430.8 | vldm memreg, doublelistdst1 |
| 430.8 | vldm memreg, singlelistdst2 |
| 537.05 | vstr vfpsd, mem |
| 537.3 | vstm memreg, singlelistsrc1 |
| 537.7 | vstr vfpss, mem |
| 549.65 | vstm memreg, singlelistsrc5 |
| 549.85 | vstm memreg, singlelistsrc3 |
| 556.5 | vstm memreg, singlelistsrc2 |
| 559.4 | vstm memreg, doublelistsrc1 |
| 596.7 | vstm memreg, singlelistsrc4 |
| 597.35 | vstm memreg, doublelistsrc2 |

**End of Table 20**

# Appendix P Raspberry PI benchmark (VFP MUL)

Table 21: Raspberry PI micro-benchmark listing for instructions in the VFP MUL class

| Milliamperes | Instruction |
|---|---|
| 406.65 | vmul.f32 vfpsd, vfpss, vfpss |
| 409.25 | vnmul.f32 vfpsd, vfpss, vfpss |
| 409.85 | vmls.f32 vfpsd, vfpss, vfpss |
| 410.3 | vmla.f32 vfpsd, vfpss, vfpss |
| 411.6 | vmul.f64 vfpdd, vfpds, vfpds |
| 414.8 | vnmls.f32 vfpsd, vfpss, vfpss |
| 415.0 | vnmla.f32 vfpsd, vfpss, vfpss |
| 415.0 | vnmls.f64 vfpdd, vfpds, vfpds |
| 415.45 | vnmul.f64 vfpdd, vfpds, vfpds |
| 416.9 | vmla.f64 vfpdd, vfpds, vfpds |
| 417.35 | vmls.f64 vfpdd, vfpds, vfpds |
| 417.6 | vnmla.f64 vfpdd, vfpds, vfpds |

**End of Table 21**

# Appendix Q Raspberry PI benchmark (VFP DIV)

Table 22: Raspberry PI micro-benchmark listing for instructions in the VFP DIV class

| Milliamperes | Instruction |
|---|---|
| 390.1 | vsqrt.f64 vfpdd, vfpds |
| 390.3 | vsqrt.f32 vfpsd, vfpss |
| 409.5 | vdiv.f32 vfpsd, vfpss, vfpss |
| 414.6 | vdiv.f64 vfpdd, vfpds, vfpds |

**End of Table 22**

# Appendix R Pandaboard ES Gen1 instruction set

```
vadd.f64     {vfpdd}, {vfpds}, {vfpds}
vmov         {vfpsd}, {rs}
vdiv.f32     {vfpsd}, {vfpss}, {vfpss}
vnmla.f32    {vfpsd}, {vfpss}, {vfpss}
vceq.i8      {vfpqd}, {vfpqs}, {vfpqs}
vhsub.s32    {vfpqd}, {vfpqs}, {vfpqs}
vorn {vfpqd} {vfpqs}, {vfpqs}
vqdmlal.s32  {vfpqd}, {vfpds}, {vfpds}
vrsqrts.f32  {vfpdd}, {vfpds}
vld1.16      {doublelistdst2}, {alignedmemstep64}
vldr         {vfpdd}, {mem}
uqasx        {rd}, {rs}, {rs}
pkhtb        {rd}, {rs}, {rs}, asr #16
add          {rd}, {rs}, {rs}
smlsd        {rd}, {rs}, {rs}, {rs}
b            {label}
```

# Appendix S Pandaboard ES Gen1 first block source-code

```
vhsub.s32  q0, q1, q2
smlsd      r0, r1, r2, r3
vldr       d6, [lr]
vorn       q4, q5, q6
pkhtb      r4, r5, r6, asr #16
b          <label0>
--- 336 NOPs ---
label0:
uqasx     r7, r8, r9
vldr      d7, [lr]
uqasx     sl, fp, r0
vhsub.s32 q7, q8, q9
add       r1, r2, r3
vld1.16   {d0-d1}, [lr :64], ip
smlsd     r4, r5, r6, r7
b         <label1>
---- 336 NOPs ---
label1:
vorn      q4, q5, q6
pkhtb     r8, r9, r0, asr #16
vld1.16   {d5-d6}, [lr :64], ip
```

# Appendix T Pandaboard ES Gen2 instruction set

```
vhsub.s32 {vfpqd}, {vfpqs}, {vfpqs}
vorn      {vfpqd}, {vfpqs}, {vfpqs}
vld1.16   {doublelistdst2}, {alignedmemstep64}
vldr      {vfpdd}, {mem}
vldreq    {vfpdd}, {mem}
uqasx     {rd}, {rs}, {rs}
uqasxeq   {rd}, {rs}, {rs}
pkhtb     {rd}, {rs}, {rs}, asr #16
pkhtbeq   {rd}, {rs}, {rs}, asr #16
add       {rd}, {rs}, {rs}
addeq     {rd}, {rs}, {rs}
smlsd     {rd}, {rs}, {rs}, {rs}
smlsdeq   {rd}, {rs}, {rs}, {rs}
b         {label}
beq       {label}
```

# Appendix U Pandaboard ES Gen2 first block sourcecode

```
pkhtb     r0, r1, r2, asr #16
b         <label0>
--- 496 NOPs ---
label0:
vldr      d0, [lr]
b         <label1>
--- 496 NOPs ---
label1:
pkhtbeq   r3, r4, r5, asr #16
vhsub.s32 q1, q2, q3
uqasx     r6, r7, r8
vld1.16   {d8-d9}, [lr :64], ip
add       r9, sl, fp
uqasxeq   r0, r1, r2
vorn      q0, q5, q6
vldr      d14, [lr]
vhsub.s32 q1, q2, q3
add       r6, r7, r8
vld1.16   {d8-d9}, [lr :64], ip
smlsdeq   r9, sl, fp, r3
pkhtbeq   r0, r1, r2, asr #16
vorn      q0, q5, q6
```

# Appendix V Raspberry PI instruction set

```
qsax          {rd}, {rs}, {rs}
sel           {rd}, {rs}, {rs}
pkhtb         {rd}, {rs}, {rs}
stm           {memreg}, {reglistsrc4}
strd          {regsevendst2}, {memstep}
smlawb        {rd}, {rs}, {rs}, {rs}
vcvt.f32.f64 {vfpsd}, {vfpds}
vmov          {adjvfpsd2}, {rs}, {rs}
vsub.f32      {vfpsd}, {vfpss}, {vfpss}
vdiv.f64      {vfpdd}, {vfpds}, {vfpds}
vstm          {memreg}, {doublelistsrc2}
vnmla.f64     {vfpdd}, {vfpds}, {vfpds}
b             {label}
```

# Appendix W Raspberry PI first block source-code

```
strd      r0, [lr], ip
b         <label0>
--- 896 NOPs ---
label0:
qsax      r2, r3, r4
vsub.f32  s0, s1, s2
b         <label1>
--- 896 NOPs ---
label1:
vstmia    lr, {d2-d3}
sel       r5, r6, r7
vmov      s8, s9, r8, r9
sel       sl, fp, r0
qsax      r1, r2, r3
b         <label2>
--- 896 NOPs ---
label2:
b         <label3>
--- 896 NOPs ---
label3:
vstmia    lr, {d5-d6}
sel       r4, r5, r6
vsub.f32  s3, s14, s15
vmov      s0, s1, r7, r8
stm       lr, {r0, r9, sl, fp}
b         <label4>
--- 896 NOPs ---
label4:
```