NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4258 Energy Efficient Computer Design
Laboratory Report

Exercise 1
GPIO programming in assembly

Group 3:

Tom Meland Pedersen
Morten Alver Normann
Magne Alver Normann

February 8, 2015

Abstract

This document is a lab report describing the first exercise in TDT4258 Energy Efficient Computer Design. The report describes how a simple assembly-program was written and uploaded to the 32-bit ARM Cortex-M3 EFM32GG microcontroller from Silicon Labs.

The goal of the program was to enable the control of 8 LEDs by the use of 8 buttons, all of which were situated on an external circuit board connected to the GPIO pins of the EFM32GG. The program was designed to be energy efficient and this report will discuss the different power optimization techniques possible, and suitable for this application.

The hardware used in this assignment was an EFM32GG DK3750 development board with an external circuit board containing some LEDs and buttons connected to the GPIO port C an A, respectively. The code was cross-compiled using GCC on an Ubuntu workstation and uploaded to the microcontroller using eACommander.

1 Introduction

Microcontrollers are everywhere around us, and we interact with hundreds of them every day. They can be in anything from your electric toothbrush to your car or your cell phone. They are cheap, tiny special purpose computers making our lives easier. The field of embedded designs is an exciting field to be in right now, with the explosion of the Internet of Things (IoT), where more and more things are made to include MCUs and to be connected to the internet. There is a growing demand for MCUs, and along with this grows the demands for their abilities. They need to be fast, reliable and just as important; energy efficient. This document will describe how a simple MCU application, trying to meet these requirements, was implemented, uploaded and tested to an ARM Cortex-M3 32-bit microcontroller.

2 Background and Theory

The EFM32 Giant Gecko is a 32-bit microcontroller from Silicon Labs. It features a 48 MHz ARM Cortex-M3 core, up to 1024 kB flash, 128 kB RAM and up to 93 GPIOs. The Cortex-M3 processor is built on a high-performance processor core, with a 3-stage pipeline Harvard architecture. The microcontroller strives for energy efficiency and has a flexible energy management system with for instance 5 different energy modes. [3]

The assembly language is a low-level programming language with a very strong (often one-to-one) correspondence to the architecture's machine code instructions. For this reason all programming in this assignment has been done in assembly (Thumb-2) to ensure a good understanding of how the program is executed on the hardware.

The EFM32GG utilizes memory mapped I/O. This means that the CPU can access input and output devices the same way it reads and writes to memory locations.

The technique of constantly checking for a certain event is called polling. This technique requires constant CPU attention, without any actual work being done. This results in inefficient use of the processor and id therefor often omitted by the use of interrupts. Interrupts are used to inform the processor of urgent matters. I.e. when a process is ready to execute, or in this case, when actions are performed that the process must respond to. Then the processor will save required information for its current task, handle the interrupt, before continue with the task it was working on before the interrupt. This technique enables the processor to enter sleep mode when idle.

Sleepmodes are energy saving modes for the processor, where the power is saved by turning off unused parts of the processor. The EFM32 Giant Gecko has four different levels of sleep, which is described at page 8 in [3]

The hardware used for this report was an EFM32GG DK3750 development board with an external circuit board containing some LEDs and buttons connected to the GPIO port C and A, respectively. The code was cross-compiled using GNU Compiler Collection (GCC) and a given makefile on an Ubuntu workstation and uploaded to the microcontroller using eACommander.

3 Methodology

3.0.1 Reading and writing to registers and memory

In order to read and write to memory, hexadecimal 32 bit addresses are used. a hexadecimal number is on the form 0xn where n is a value from the range {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f}, each one can be translated to a four digit binary number on the form 0bnnnn where each n is a number in the range {0, 1}. Thus, writing 0xffffffff to a 32 bit register is equal to writing a binary number consisting of 32 ones into the register (setting all the register bits high).

The structure of the ARM 32 bit instructions limit the the values that can be written directly to the registers. All values must be constructible from a 8 bit value rotated an even number of spaces, larger numbers can be constructed by using logical or and masking two numbers with each other. If they are not, special Thumb-2 [2] instructions movw and movt can be used.

The ARM architecture uses indirect register addressing, meaning that all operations on data takes place in registers (not in memory directly), hence all data that we wish to operate on must first be loaded into a register. In order to do this, ldr and str instructions [2] are used. ldr loads the data in a memory location addressed by a 32 bit number into a register given as a argument, the 32 bit address is often stored in a constant and used directly, or preloaded into another register and then offset by a constant to give a memory address relative to the stored address, the syntax for offsetting and general structure can be found in the thumb reference datasheet [2].

In order to change individual or some bits in a register (but not rewrite all) we can use logical operations on multiple registers / values and store the results. As an example, 0xff000000 masked (logical or) with 0x000000ff gives 0xff000000ff.

3.0.2 Enabling GPIO clock in the CMU

One of the most common ways of interfacing a microcontroller with its environment is by utilizing its General Purpose Input/Output pins. These are pins whose settings and status can be controlled by interfacing the GPIO controller by writing the appropriate values to its registers. In this task we used the GPIO pins in order to communicate with the peripherals, mainly the controller and the LEDs on the controller. This required enabling and configuring the GPIO controller and pins, as well as choosing a method of receiving and detecting changes on the pins.

Due to its focus on energy efficiency the Giant Gecko EFM32 has a CMU (Clock Management Unit) which can be used to enable and disable the clock for certain peripherals. In order to use the GPIO controller, its clock must first be enabled by setting the appropriate status registers in the CMU. To do this, we set the 13th bit in the CMU_BASE_HFPERCLKEN0 register to high (logical one) while maintaining the former status of the other bits.

3.0.3 Configuring GPIO pins

Once the clock for the GPIO controller has been enabled, the pins must be configured. In this task we chose to use port A for the output pins and port C for the input pins, both ports have 16 general purpose pins, we will only be using pin 0 to 7 on port C, and pin 8 to 15 of port A. This is because we have eight buttons that serve as inputs and eight leds serving as outputs. The configuration of port A was done by setting the drive strength of the port to low by writing 0x2 to the register referenced by GPIO_PA_CTRL (this sets the current on the output pins to the leds to 20mA), before setting the pins to push-pull and enabling the use of the alternative drive strength (previously configured) by writing 0x55555555 (this is equivalent to eight 0101 binary pin configurations, one for each pin) to GPIO_PA_BASE offset by GPIO_MODEH (MODEH because we are using the eight highest numbered pins 8 - 15). The push-pull configuration enables the output of the pin which can be set by writing to GPIO_PA_DOUT.

In order to properly configure the input pins we wrote 0x33333333 (equivalent to eight 0011 pin configurations, one for each pin) to GPIO_PC_BASE offset by GPIO_MODEL (MODEL because we are using the eight lowest numbered pins 0-7). This sets pin 0 to 7 on port C to input pins and enables pull-down and filtering. The pull-down configuration enables the pull-down resistor of the pins, thus ensuring that the pins do not float (take on arbitrary values) when the buttons are not connected, the resistor circuit pulls the pin low enforcing a low voltage. The filtering is enabled to reduce (filter out) any potential noise on the input pins.

3.0.4 Configuring interrupts

In order to read the button input on port C, we can either poll the input pins continuously using a loop or use interrupts. Because the polling strategy is terribly inefficient with respect to power usage (the processor busy-waits, thereby wasting a lot of time and power doing no useful work) using interrupts is preferable since the processor can sleep while waiting for a interrupt (a change in value) on one of the pins.

To enable interrupts on the MCU we had to first configure the NVIC (Nested Vector Interrupt Controller). The NVIC contains two registers for enabling interrupts, ISER0 and ISER1. By setting the appropriate bits in these registers different interrupts can be enabled, in this case we want to enable interrupts on even and odd GPIO pins (all pins), to do this bit 1 and 11 (0 indexed) in the ISER0 register was set by writing 0x802

to the register. In this case we had to use the movw (move wide) instruction because 0x802 (the hexadecimal representation of a binary number with bit 1 and 11 set to 1) is not representable by rotation.

The NVIC also contains a vector table that has the addresses of the interrupt handlers for all the different interrupt service requests (ISQ) stored. When a ISQ is received, the controller performs a lookup to the vector table and passes control to the interrupt routine by jumping to the handler address stored in the table. In order to perform a specific action on a specific interrupt, the handler should be written and its starting address should be stored in the vector table of the NVIC in the position matching the IRQ we want to handle. This had already been done in the base code skeleton we received for this exercise and all we had to do was fill in the handler routine code.

Once the NVIC has been configured to enable interrupts and the handler routine written and stored in the vector table, we have to configure the GPIO pins so that they will generate interrupts when the state of the pins (high or low) changes. In order to do this, we first wrote 0x22222222 to the GPIO_EXTIPSELL register, this selects the lowest (0-7) port C pins for interrupt generation, then we wrote 0xff to the GPIO_EXTIRISE and GPIO_EXTIFALL (setting all bits to 1), thus stating that interrupts on rising and falling edges for the GPIO pins should be triggered. At this point the flag corresponding to the changing pin in the GPIO_IFC will be set on a change in pin state. Finally, we enabled the interrupt generation for the pins by writing 0xff to the GPIO_IEN register, this makes it so that an interrupt request will be sent on IRQ_GPIO_ODD/EVEN depending on the pin number, thus waking the processor from its sleep and passing control to the handler stored in the IRQ_GPIO_ODD/EVEN entry in the NVIC vector table (same handler for both in this case).

The interrupt handler we chose to implement performed a simple mapping from input to output after clearing the interrupt flags in GPIO_IFC. Clearing the interrupt flags manually is necessary in order to prevent the handler from being called continuously. The mapping was done by reading the status of the input pins (GPIO_DIN on port C), left shifting them 8 bits, and writing the read data to the output pins (GPIO_DOUT on port A). The left shift was necessary since the input pins are numbered 8-15 while the input pins are numbered 0-7, the shift ensures the data is mapped to the right pins.

3.1 Energy optimization

3.1.1 optimizing energy consumption

Power consumption is an important aspect of embedded systems, and especially for those who are battery powered. Low power implies lower cost of operation and smaller battery size, making applications more mobile. It could also mean longer battery life, which is often of great importance to the consumer. Even wall-powered applications may benefit from power efficient designs as this is better for both the environment and your wallet.

The following section will discuss different techniques to lower this applications power consumption.

In order to adjust and optimize the energy consumption of the MCU, the settings of the Energy Management Unit (EMU) registers can be adjusted to best match the task at hand. This allows for running in different energy modes which have different levels of energy consumption, wake-up time and so on.

3.1.2 Reducing static power consumption

Most applications spend significant periods of time in standby mode, and thus the static power consumption contributes greatly to the overall consumption. The static power consumption is the sum of leakage current, current consumed by power management circuits, clocking systems, power regulators, IOs, and so on.

This application does indeed spend most of it's time in an idle state as computations are only needed the exact moment when a button is triggered or released. Because of this, techniques to reduce the static power would most likely have a greater effect on the total power consumption and was explored first.

GPIO leakage

To minimize the energy dissipation in the GPIO-pins, all unused pins should be disabled. Luckily this is done for all pins (except those used for debugging) upon reset. Thus there was no need to implement code doing this.

Disabling RAM blocks

The leakage of the RAM blocks contribute significantly to the over all power consumption in the device, particularly in EM2 and EM3 [1]. The RAM in the EFM32GG is divided into four blocks of 32 KB, where as all is enabled by default. The program for reading the buttons and setting the LEDS is quite small, thus it was quite likely that some RAM blocks could be disabled to further optimize the power consumption. Upon testing it was discovered that the program in fact did not use any of the upper three RAM blocks, hence all three was disabled. As the the current consumption is decreased with approximately 170nA per deactivated RAM block, this should result in an approximate $170nA * 3 = 510nA$ reduction in power consumption, see the result section for actual results.

In order to disable the aforementioned RAM blocks, the appropriate value of 0x7 was written to the EMU_MEMCTRL register according to the Giant Gecko datasheet [3].

Other

The AN0027 Application Note on power optimization from silicon labs suggests that bias current for analog peripherals could be reduced to optimize the power consumption, this is however only applicable to analog peripherals of which this program uses none. The supply voltage greatly effects the power consumption in micro controllers, however the CMOS logic in the EFM32GG is already supplied with 1.8V and are not functional with voltages lower than this. Thus this is already at it's minimum.

3.1.3 Reducing dynamic power consumption

When designing energy efficient applications, it is important to limit the power spent at any time. This means trying to reduce the time spent in higher energy modes.

As the dynamic power consumption increases with the clock speed, reducing it is mostly about optimizing the clocks. The three main ways of doing this is of course to turn unnecessary clocks off, generate or prescale the clocks needed to as low a frequency as possible, and pruning the clock tree (meaning disabling portions of the circuitry so that the flip-flops in them do not have to switch states). This last technique is called clock gating.

When it comes to this application a lot of this is handled by entering the different energy modes. For instance when entering EM2 and lower, the high frequency clocks are disabled, leaving only the low frequency clock peripherals active. Upon reading the datasheet it was discovered that the external interrupts was supported as low as to EM3. Hence our application should be functional even in EM3. It was therefor made an attempt of entering EM3 from EM2 by disabling the low frequency peripheral clock.

Energy Mode 2 (EM2) was enabled by writing 0x0 to the EMU_CTRL register as well as enabling deep sleep by writing 0x6 to the SCR (status control register) thus setting the second and third bits according to the datasheet [3]. Furthermore we allowed the MCU to enter the even more energy saving mode EM3, simply by disabling the low energy peripherals clock, this was done by writing 0 to the CMU_LFCLKSEL register (setting the bits low).

The clock prescaling was not attempted as it would most likely only account for about 50nA reduction, as The GPIO was the only peripheral used by the application. [1]

3.2 Testing

The correctness of the implementation with respect to the requirements of the assignment was tested by uploading the code onto the EFM32GG and pushing the buttons on the gamepad in different combinations. Different versions of the code with varying degrees of power optimization measures taken was tested separately while the power consumption characteristics was monitored on the Ubuntu workstation using eAProfiler. The results can be found under the results section of this report.

4 Results

The behavior of the implementation corresponded with the initial plan. When a button was pushed, the corresponding light was turned on.

4.0.1 Optimizing energy consumption

Since most of the time none buttons are pushed, the most effective way to optimize the energy consumption is to reduce the static energy usage. This was done by setting the microprocessor to deep sleep (EM2) when not in use. Then the current, with no pressed buttons, went from milliampere to microampere, showed in figure 4.2 and 4.1.

The different between EM2 and EM3 is that EM2 uses the low frequency clocks for the peripherals. Therefore, the low frequency clocks were turned off to enter EM3, and the energy consumption was inspected. There was no noticeable improvement in power consumption, which means that either the difference was to small to be noticed or the implementation did not work as planed.

In addition, since three of the RAM blocks was not needed, they were turned off. This reduced the energy consumption to about half, as seen in figure 4.4 and 4.3.

To reduce the energy consumption while buttons were pushed, the drive strength of the lights were set to 1 (low) instead of 2 (high). This means the lights use 0.1 mA instead of 20 mA. The difference can be seen by observing figure 4.5 and 4.6.

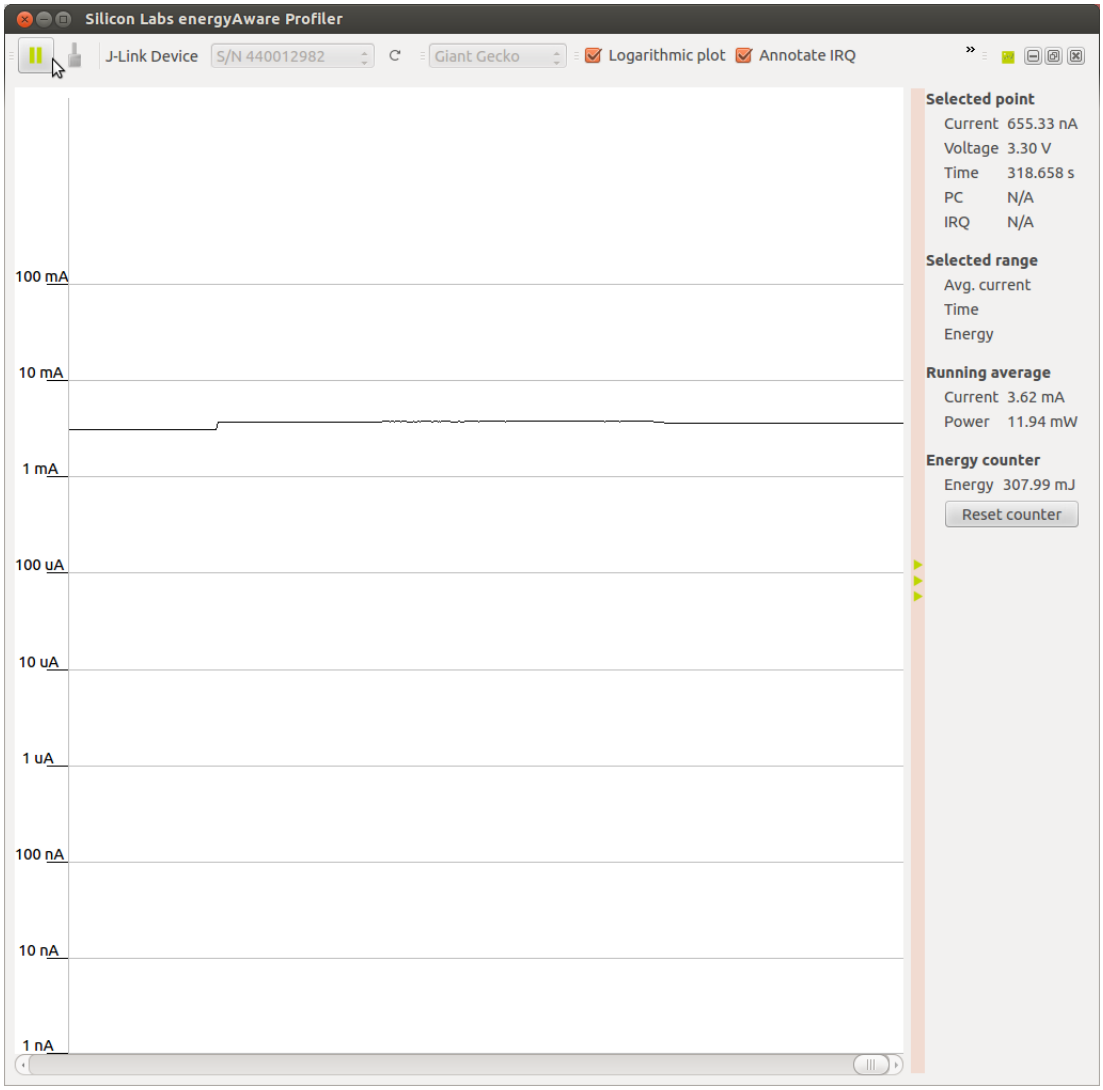

Figure 4.1: Energy consumption without any sleep.

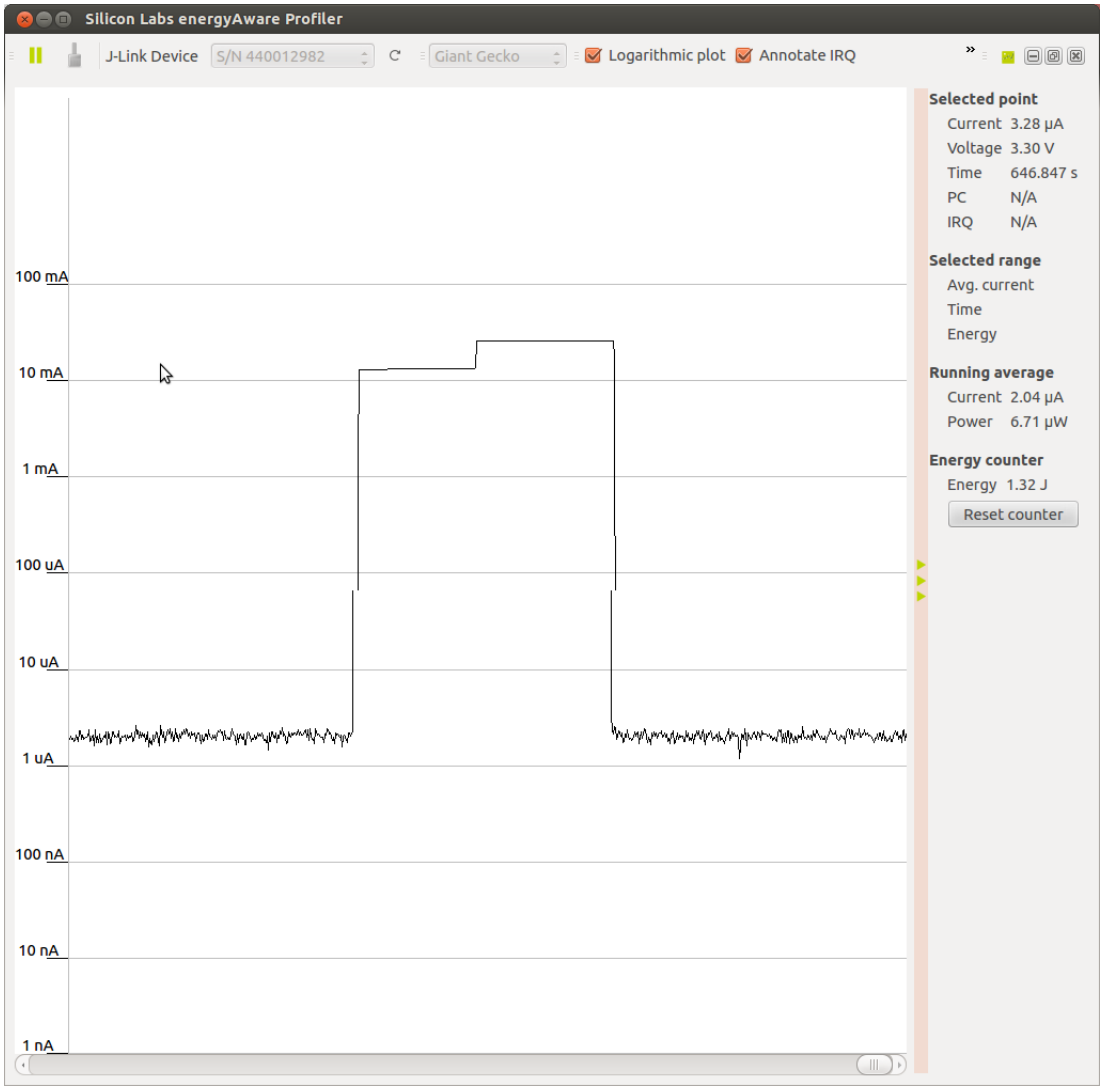

Figure 4.2: Energy consumption with deep sleep.

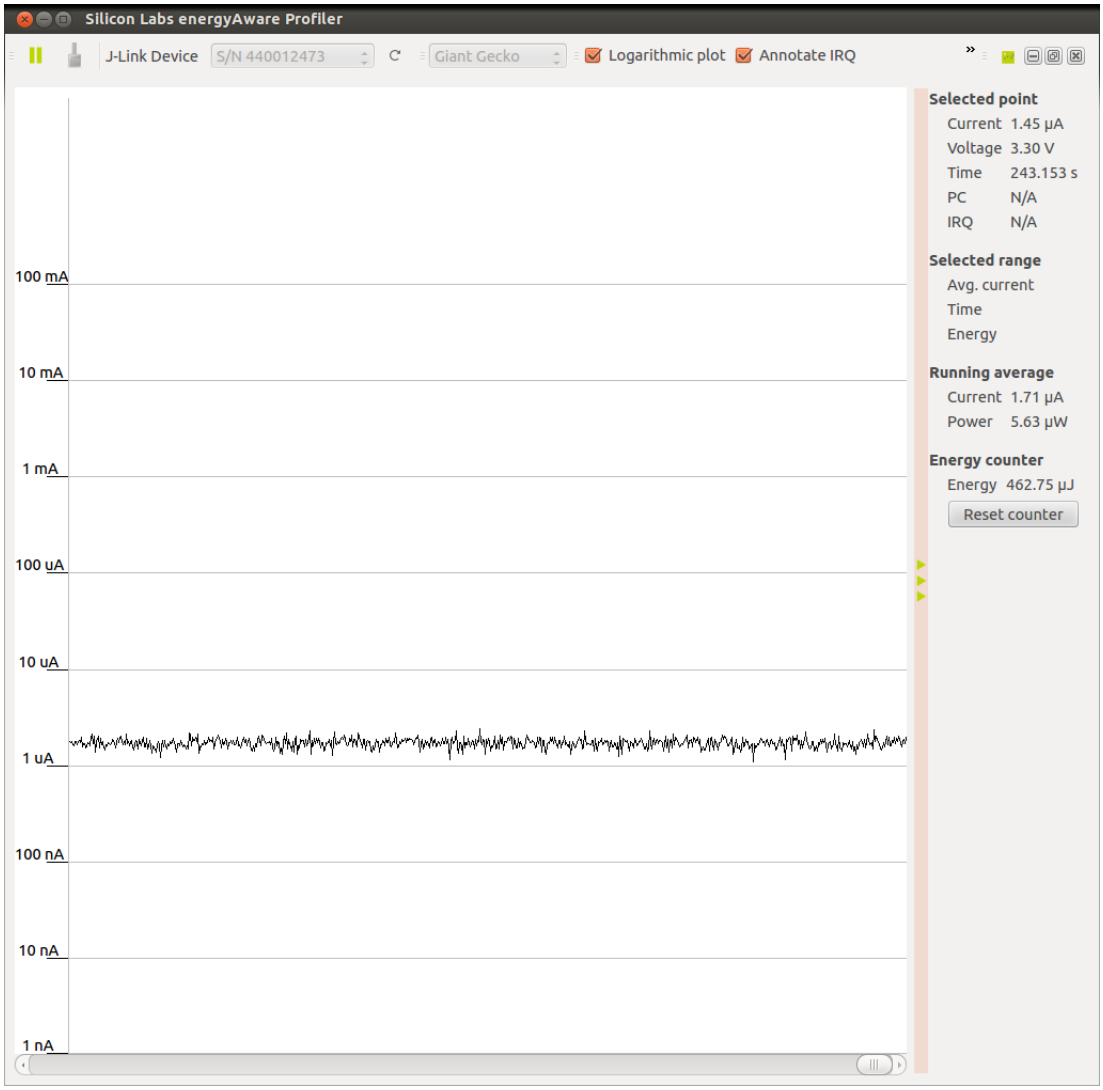

Figure 4.3: Static energy consumption with enabled RAM.

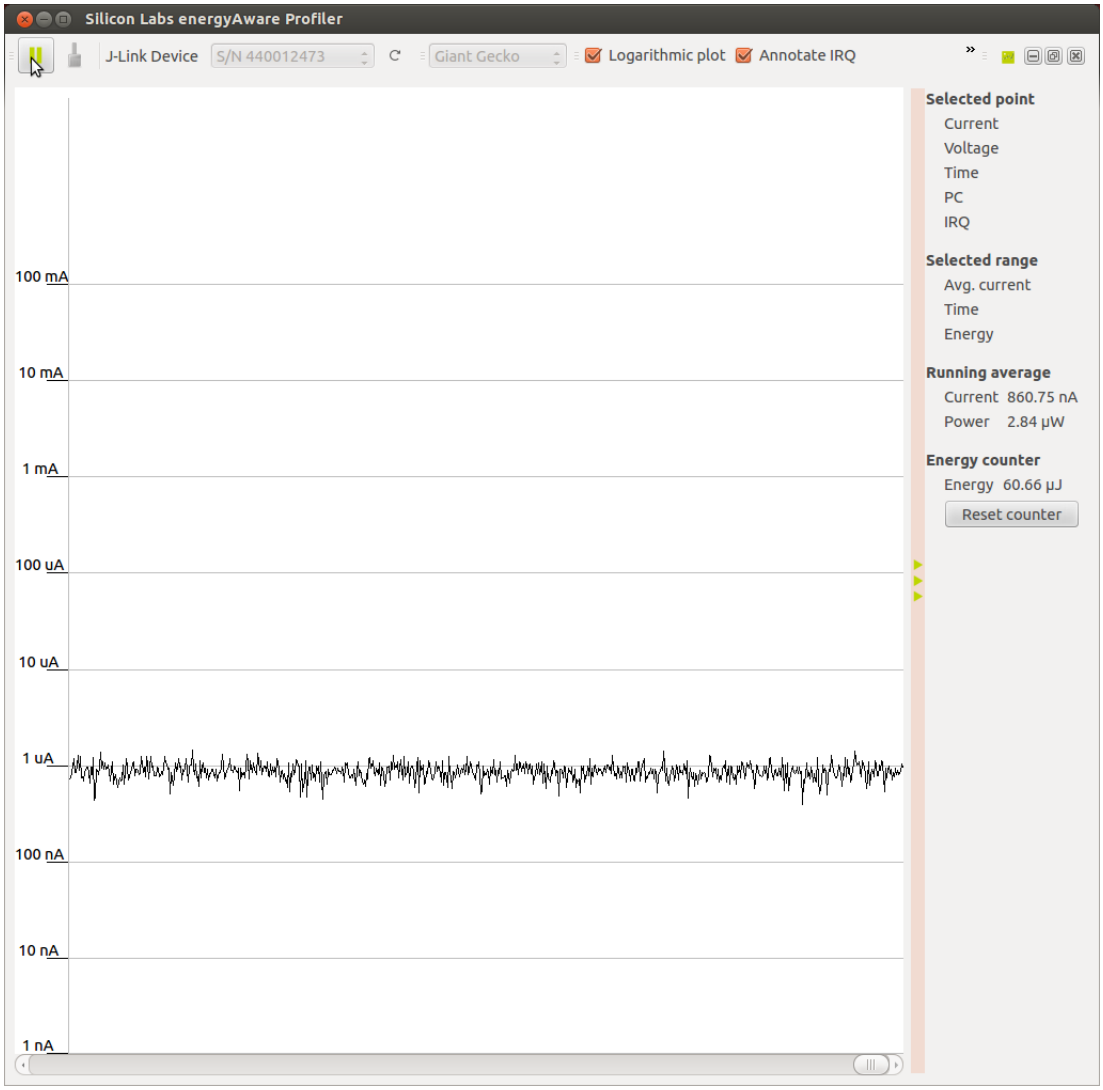

Figure 4.4: Static energy consumption with disabled RAM.

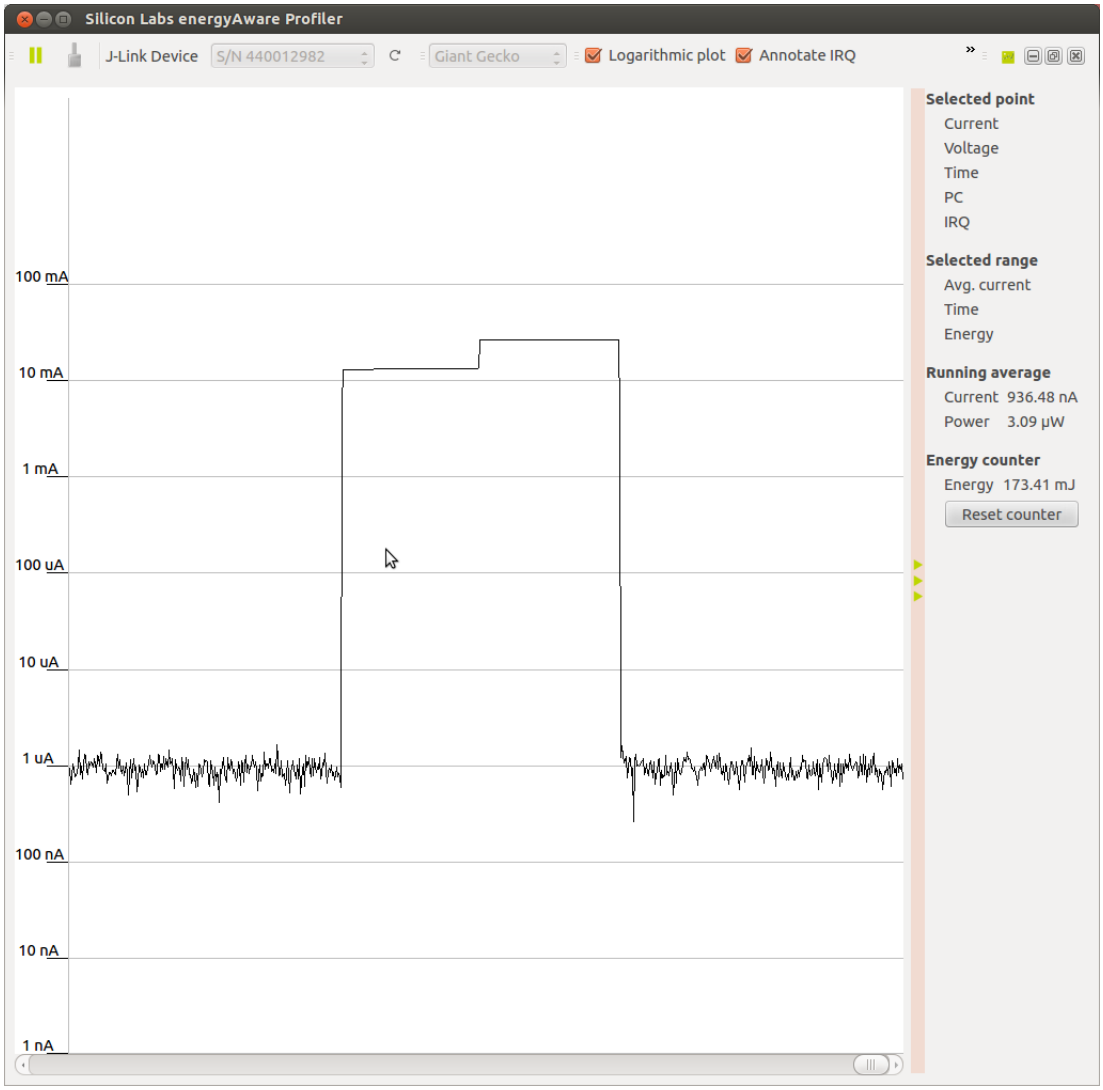

Figure 4.5: Energy consumption with drive strength set to 2

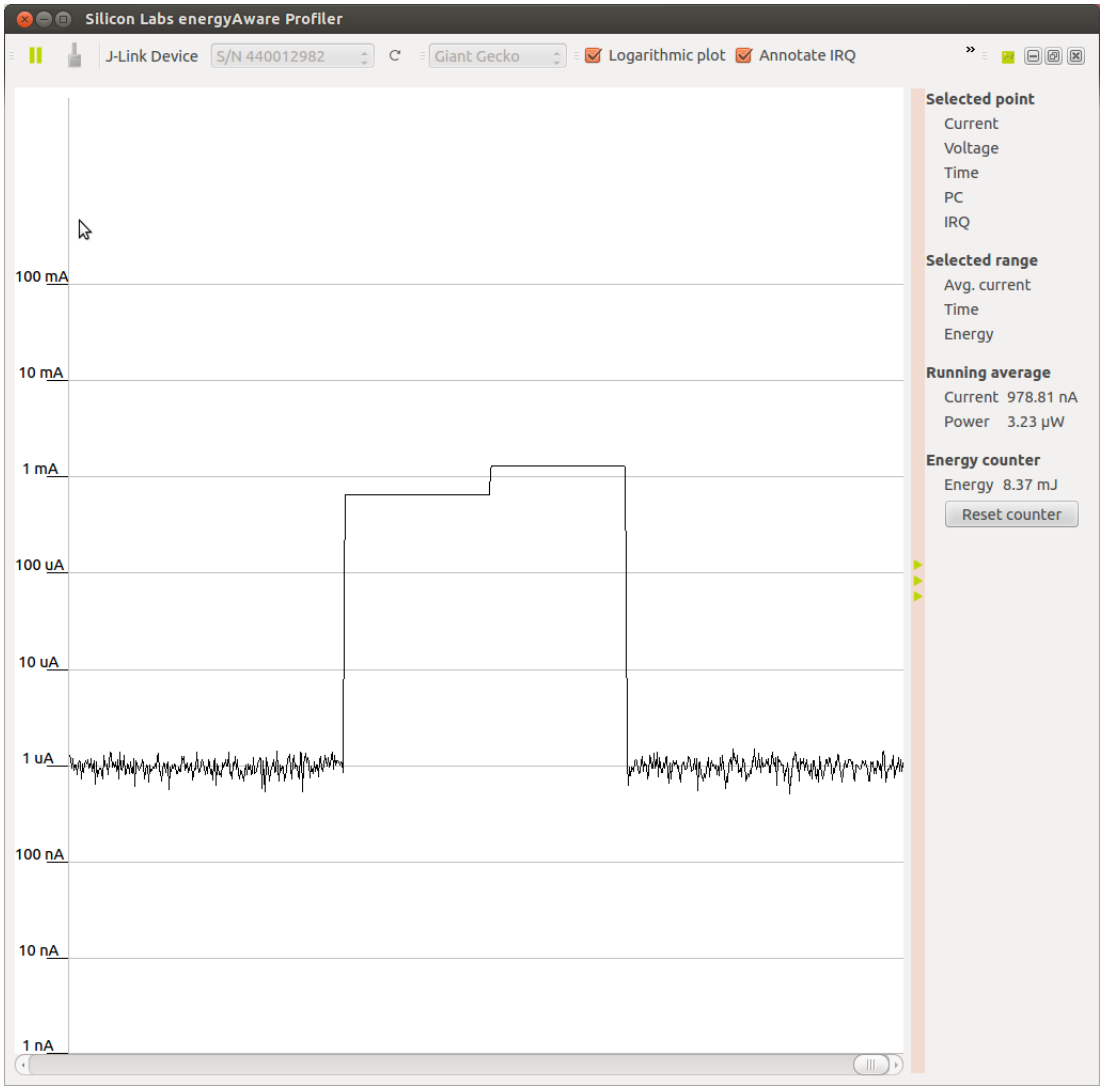

Figure 4.6: Energy consumption with drive strength set to 1.

5 Conclusion

This report has described how a simple assembly program was constructed and how it was optimized with concern to power efficiency. The program was constructed so that when any of the 8 buttons on a gamepad was pressed, a corresponding LED would light up on the gamepad. The program was designed to rely on interrupts in stead of polling, enabling the application to spend most of it's time in sleep mode, waking up whenever a button was pushed or released and then go back to sleep again. The power consumption was plotted and analyzed with different methods of power-optimization. It was discovered that simply by including sleep modes in the design, the power consumption could be reduced by 99.94%. To further reduce the power consumption, unnecessary RAM-blocks were shut off, resulting in another $850nA$ reduction which may not sound like much but with already sleep mode enabled this meant a 50% reduction in power consumption, which for a battery powered application would almost double the battery life time.

5.1 Evaluation of the Assignment

We found the assignment to be interesting and fun. It was fascinating to program so low level and it gave us a better understanding of how code is really executed. We really liked the way the assignment was structured with the compendium telling us what needed to be done, but that we had to find out for ourselves how it should be done. We also liked how all the information needed (datasheets, app notes ++) was made easily accessible to us. Our suggestions of improvement is probably to have better access to a student assistant, either by having one more or by expanding the time he spends at the lab, as the help we were able to get was rather limited. But all in all this was a great assignment and we are looking forward to the next one.

Bibliography

[1] Silicon Labs. An0027 energy optimization, 2013.

[2] Silicon Labs. Arm thub-2 quick refrence sheet, 2013.

[3] Energy Micro. Efm32g reference manual, 2011.