TDT4258
Energy Efficient Computer Design
Exercise 1

Lundal, Per Thomas
perthol@stud.ntnu.no

Normann, Kristian
krinorm@stud.ntnu.no

Selvik, Andreas Løve
andrels@stud.ntnu.no

February 14, 2013

Abstract

The purpose of this assignment was to create an assembly program that would turn on the central LED on a given microcontroller, and then allow a user to move the light either left or right.

The reasons for this exercise were for the sake of learning the architecture and general workings of the AVR32 processor, the STK1000 development board and the AT32AP7000 microcontroller, as well as understanding the principles behind and usage of hardware components, parallel I/O, interrupts, debouncing, energy saving and debugging.

The exercise was approached by first reading up on the technical documentation of the instruments and tools, and then dividing the assembly programming up into smaller subtasks.

In the end the final program worked well and according to specifications.

Contents

I Introduction

Energy efficiency is a topic that had garnered much attention lately. With an increasing need for electrical apparatus, saving as much energy as possible is a necessity for allowing our current technological development to continue. Energy can be saved in all aspects of a micro-controller, and every bit counts. However, as hardware components and architectures have been perfected over many years, there is very little to gain here. The system architecture allows for a moderate amount of energy saving, but the most important part is the types of algorithms being used, as the difference between a good and a bad one can be humongous.

Producing customized computer chips for each new development project is expensive and not very flexible. Hence, most start by developing on so called development cards which are specially made for the job by featuring nearly all thinkable features and pins for connecting different components. After development have finished, a computer chip with all unnecessary components removed can then be produced.

The STK1000 is one such development card and will be used for this assignment. It features an AT32AP7000 microcontroller from Atmel containing an AVR32 processor core. AVR32 is a 32-bit RISC processor architecture made especially for use in embedded systems that demands a relatively high amount of performance. It has a high clock frequency compared to those used in many other microcontrollers and features a seven stage pipeline which allows for up to three different instructions to be executed at the same time. The microcontroller also makes use of memory mapped IO which means that interaction with external devices is done through simple memory read and write operations.

A phenomenon that has become a problem with increasing processing speed is bouncing. When a mechanical switch is toggled, the impact between the switch and the holder can cause the button to bounce a little which causes contact to be lost and regained. This has not been a problem in the past because hardware has been too slow to detect it, but modern microcontrollers are so fast that they can react to the event before the switch has stabilized it's position, which can cause multiple events to be detected. To compensate for this, a technique called debouncing is used, which essentially pauses execution for a short time to allow the switch to stabilize.

With all programs comes the need for debugging, unless the programming is flawless, which of course is highly unlikely. Most of the currently available microcontrollers therefore have a built-in debugging module to make this easy. This module allows for step-by-step execution of the program, information about what is stored where at every time and more.

The purpose of this assignment was to create an assembly program for the STK1000 that would turn on the central LED, and then allow a user to move the light left and right with the buttons on the board. Furthermore, it was required to read the button states in an interrupt routine, while processing them in a main loop that would sleep between events.

II Description and methodology

The initial step in solving the task was to read up on the technical details on the AT32AP7000 microcontroller, the STK1000 development-board and the the AVR32 instruction set in order to figure out how to actually solve the assignment. Then a plan of five subsequent steps was made.

Activate LEDs Read button states Make use of interrupts Optimize for energy efficiency Add additional functionality if time allows

The reason for splitting up the assignment into smaller parts was due to debugging and ensuring some measure of incremental success, instead of attempting to solve the whole assignment in one go and ending up debugging for weeks.

II.1 Jumper and cable configuration

The LED's were connected to PIOC by a flat-cable from GPIO pins 16-23(J3) to the LED-pins(J15) on the STK1000 board [1, section 2.4.1]. The buttons were connected to PIOB by flat-cable from GPIO pins 0-7(J1) to the SWITCH-pins(J25) on the STK1000 board [1, section 2.4.1]. On the AT32AP7000 we set the jumpers SW6 and SW4 to GPIO(so that PIOC and PIOB would be connected to GPIO)[1, table 2.3].

II.2 LEDs [unfinished]

As mentioned above, the first goal was to simply light up the LEDs. We used the value 0b11111111 to target all LEDs, and started by writing the value to the Pin Enable Register (PER) and Output Enable Register (OER). Then

After this point, the program entered an infinite loop that did nothing.

The program wrote 0xFF to PIOC's PER and OER memory locations to enable the relevant pins and set them in output mode, and then wrote 0x55 to SODR in order to turn every other LED on.

As mentioned above, our first goal was simply to get the LEDs to light up. The LEDs were connected to GPIO pins 16-23, which is mapped to the 0-7 bits in the PIOC memory location. The program wrote 0xFF to PIOC's PER and OER memory locations to enable the relevant pins and set them in output mode, and then wrote 0x55 to SODR in order to turn every other LED on. At this point, the program entered an infinite loop that did nothing.

KLADD The program structure at this point was really simple; enable the PIO ports associated with the LED and turn them on in a given pattern, then enter an infinite loop that does nothing.

II.3 Buttons [unfinished]

With the LEDs up and running, the next step was to get the buttons under control. This was achieved by connecting the buttons to the PIO 0-7 ports, which are mapped to the PIOB 0-7 pins, enabling these pins as we did for the LEDs, and activating the pull-up resistors by writing 0xFF to PIOB-PUER. Now in the infinite loop, all the LEDs were first cleared and then the inverse of 0xFF button status register , where is bitwise and, was written to the set value register. This resulted in a program that turns on an LED when and only when the corresponding button is depressed.

II.4 Interrupts

Now that the program was able to read button states and manipulate the LEDs, the next step was to implement interrupt handling by following the procedure in section 2.5.2 of the compendium [1].

We started by creating the interrupt routine which stores the button and event states to some specified registers. Except from using a special return command, this is just like a normal routine, but since the interrupt can occur at any time, all registers that is be modified by the routine must first be stored in the memory, and then restored afterwards. A practical way to achieve this is by using a stack. This was accomplished by initializing the Stack Poiner (SP) to the built-in assembler address $_stack, and using load and store instructions which update the S$

Once the routine was functional, the interrupts had to be set up. To accomplish this, the Input Enable Register (IER) is set for the pins connected to the buttons, and the Input Disable Register (IDR) is set for all other pins on PIOB to prevent unwanted events to be triggerred by noise. Afterwards the Exception Vector Base Address (EVBA) is set to zero for simplicity, and an autovector of the exception's address and priority zero is stored in the interrupt controller's Interrupt Priority Register 14 (IPR14), which corresponds to PIOB. Finally, the Global Interrupt Mask (GM) status register is cleared.

The interrupt handler was now fully functional. However, each press generated several events due to bouncing, so the final touch was to add debouncing by waiting for a short while before processing each event.

II.5 Energy optimization

After all the main functionality was in place, it was time to optimize the code where possible to conserve energy. As the program is only supposed to react to interrupts, the first and most obvious step was to put the microcontroller

into a sleep state while waiting for them instead of running in an infinite loop. This allows nearly all parts of the board to be shut down for most of the time, which conserves huge amounts of energy.

Some more energy can be saved by making the code more efficient, i.e. execute faster, which allows the board to re-enter a sleep state earlier. Therefore, the number of instruction was reduced where possible, although there were not many places this could be performed. Branches were also exchanged with conditional instructions where possible, as clearing the whole seven step pipeline can take a lot of time. These actions were taken somewhat into account from the beginning of the assignment, but a second pass did not hurt.

II.6 Additional functionality [unfinished]

A little extra time allowed us to implement some additional functionality. Specifically,

III Results and tests

This assignment's stated task was to create an assembly program for the STK1000 that would turn on the central LED and allow a user to shift the light either left or right by pressing buttons that are read in an interrupt routine. Specifically, this required the light-switching to be placed in the main loop.

There was no formal test-environment for this assignment and the main manner of testing was to visually see if the buttons and LEDs behaved as they should. This meant whether the correct lights were turned on and on time, and were turned off correctly, and on time. Eventually the resulting program behaved satisfyingly to the group-members in terms of response-time as well as fulfilling the assignment's requirements. At that point the program was deemed finished.

Choice of 0x55 enable LEDs thingy? (The value of 0x55 was chosen as an easily recognizable pattern including all the LEDs, as we experienced random lights turning on during the run of invalid code.) Loading the 32bit immidiate.

IV Evaluation of assignment

Some discussion around the manner of the report between the students and the one in charge of the assignment may be prudent. The assignments do not seem to lend themselves very well to the stated format, and one risks falling into the trap of repeating things a lot: That is not good. It is a bad enough habit as it is that some people think that a good skeleton of a report is "say what you're gonna say, say it and say what you've said", without being shoved in such a direction for lack of proper material to put in the different report-slots. I'm also inclined to say that a 'results' section is irrelevant as it is generally a place to report the results of tests and findings; something this assignment strictly speaking lacks.

Kommentere på feil i kompendiet?

V Conclusion

Despite facing occasional difficulties during the implementation, the final code worked according to the specifications of the assignment text. The group also managed to implement some use of sleep-mode for energy-saving.

VI References

References

[1] Computer Architecture and Design Group, *Lab Assignments in TDT4258 Energy Efficient Computer Systems*. Department of Computer and Information Science, NTNU, 2013, `http://www.idi.ntnu.no/emner/tdt4258/_media/kompendium.pdf`.

[2] Atmel. *AVR32 Architecture Document*, 2011, `http://www.idi.ntnu.no/emner/tdt4258/_media/doc32000.pdf`.

[3] Atmel. *AT32AP7000 Preliminary*, 2009, `http://www.idi.ntnu.no/emner/tdt4258/_media/doc32003.pdf`.