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 15, 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.

This was an introduction to the architecture and general workings of the AVR32 processor, the STK1000 development board and the AT32AP7000 microcontroller, as well as the principles behind and usage of hardware components, parallel I/O, interrupts, debouncing, energy efficiency 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 incremental 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. However, combining the required use of interrupts with putting the microcontroller in sleep mode during the main-loop, will save some energy from the hardware perspective.

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 [3] microcontroller from Atmel containing an AVR32 [2] 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.

II Description and methodology

The initial step in solving the task was to read up on the technical details on the AT32AP7000 [3] microcontroller, the STK1000 development-board and the the AVR32 instruction set [2, Section 9] in order to figure out how to actually solve the assignment. Then a plan of five subsequent steps was made.

1. Activate LEDs
2. Read button states
3. Make use of interrupts
4. Optimize for energy efficiency
5. 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.

2.1 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].

2.2 Using the debugger

Setting up and using the debugger was crucial to our success. We got the debugger up during our first visit to the lab. In order to use the gdb debugger tool on the microcontroller, we had to set up a proxy, as well as use avr32s own version of gdb. The command given in the compendium[1] gave us the following error: "Failed to look up the host localhost: Unknown server error". This was solved by a quick google, which linked us to a thread at avrfreaks[4] where someone had the same problems. It turns out that the command worked when run with no arguments, and we learned that the port 4711 was chosen by default. Therefore we ended up connecting the debugger to the microcontroller through JTAGICE mkII in the following steps:

```
$ avr32gdbproxy
$ avr32-gdb
(gdb) target remote:4711
```

Because the proxy had to keep running, the second and third command were run in a different terminal window. We could have launched the proxy in the background with an ampersand, but we would then miss out on error messages and the like.

The debugger was used by running the current code one instruction at a time with the “si” command and reading the register-values on each iteration with the “regs”, checking if the values were correct and then changing the code accordingly. Commands to set specific registers, or use breakpoints were not necessary for us during this assignment.

2.3 LEDs

As mentioned above, the first goal was to simply light up the LEDs. We used the value 0b11111111 to target all the LEDs, and started by writing the value to the Pin Enable Register (PER) and Output Enable Register (OER) of PIOC where the LEDs were connected. Then we wrote the value 0b10101010 to the Set Output Data Register (SODR) to turn every other LED on. The reason for using such a pattern was to make sure it was easy to recognize that is was our code that doing the job and not random noise. After this the program would just continue in an infinite loop.

We experienced a lot of random lights turning on at first, but solved this using the debugger, and found the problem to probably be loading of wrong pointers.

2.4 Buttons

With the LEDs up and running, the next step was to get the buttons under control. We used the value 0b11111111 to target all the buttons, and started by writing the value to the Pin Enable Register (PER) and Pull-Up Enable Register (PUER) of PIOB where the buttons were connected. Now in the infinite loop, we read the Pin Data Status Register (PDSR) and invert it to get the status of the buttons where a one means that the button is down. We also masked this value against 0b11111111 to remove any unwanted noise. Then all LEDs were cleared before writing the button statuses to to SODR to light the LEDs over the pressed buttons.

With the ability to read the buttons it was time to make them move the light. By masking the buttons statuses to find which buttons are pressed we branched to different subroutines that updates a common registry containing one and only one 1 by either shifting it left or right, and then updating the LEDs. Special checks was added so that the light would wrap around. Since each iteration of the main loop is so quick, the light appeared to move at random during normal operation, but we could see that it was working correctly with the debugger and thought implementing interrupts would fix it.

2.5 Interrupts

Now that the buttons and LEDs were operating correctly, the next step was to implement interrupt handling by following the procedure in the compendium [1, Section 2.5.2].

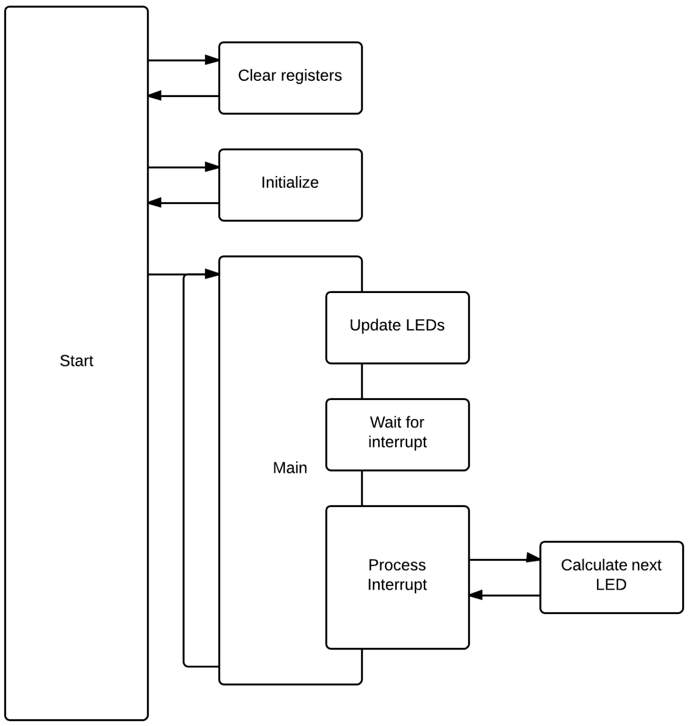

Figure 1: Structure of initialization and main loop

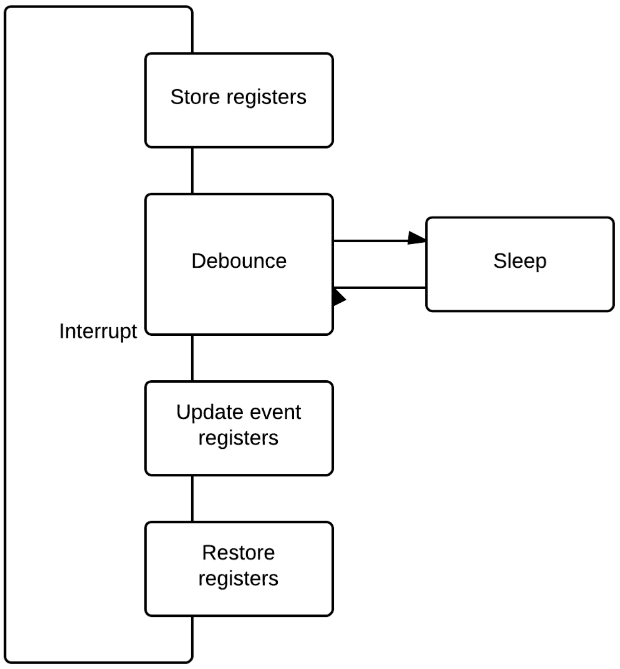

Figure 2: Structure of interrupt handler

We started by creating the interrupt routine which stores the button states and event states to some specified registers which the main loop would then read. 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 Pointer (SP) to the built-in assembler address _stack, and using load and store instructions which update the SP. Structure of the main routine is shown in figure 1, and structure of the interrupt routine is shown in figure 2.

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 triggered 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 more or less complete. However, each press generated several events due to bouncing, and the program did not distinguish between between presses and releases because we used the data from the Interrupt Status Register (ISR). Therefore, debouncing was added by waiting for a short while after an interrupt before processing the event. This was accomplished by decrementing a large number (we used 1000) down one by one to 0. Finally, a check for whether the button is now pressed or released was added by reading the PDSR as in the previous section.

2.6 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.

2.7 Additional functionality

A little extra time allowed us to implement some additional functionality. Specifically, switch 5 (SW5) will make the LEDs flash to the right for a few seconds, while switch 7 (SW7) will make the LEDs flash to the left for a few seconds. Additionally, switch

6 (SW6) will make the LEDs flash outwards in both directions from the currently lit LED.

2.8 Tools

- GitHub for handling version control
- Vim as main code-editor
- Google Docs for report collaboration
- LaTeXfor report markup
- JTAGICE mkII for connecting the computer to the board
- avr32gdbproxy for connecting to JTAGICE mkII
- avr32-gdb for connecting to the board, allowing us to debug the code

III Results and tests

We designed a set of tests to check our implementation against the specifications of the assignment, see appendix A. The final version passes all tests, but there have been some problems along the way. Test 3 failed in an earlier version, where LED 6 lit up instead of LED 7. This was because movcond can only take an immediate value of up to 255 and LED 7 has the value 256. However, a workaround was devised by first moving 255 to the register and then using subcond with -1 as there is no addcond that can take immediate values.

At first we did not find an instruction called invert or similar. Since we needed to invert the button states that was read we had to hack a solution by instead nanding the number by itself. However, after a thorough read of the AVR32 instruction set [2, Section 9] we found the complement instruction.

Eventually the group-members were satisfied with the resulting program's response time, behaviour and the fact that it passed all the tests. At that point the program was deemed finished.

IV Evaluation of assignment

When it comes to the compendium, we've found at least two errors: The front page says December 2013, and the avr32gdbproxy command doesn't work as described in section 2.2.

All in all, the assignment was a fun challenge, with a deadline which gave enough time to play around, while not being too loose.

V Conclusion

Despite facing occasional difficulties during the implementation, the final code worked according to the specifications of the assignment text. We have learned a great deal about programming on microchips, in particular the STK1000 with the AVR32. When it comes to energy efficiency, our result is a program that will sleep most of the time and only do work when necessary, which is optimal. We also had time to implement some fancy features that make simple animations with the LEDs.

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`.

[4] `http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&p=750034`

A Tests

Test 1:	**Move right**
Action:	Press button 0
Preconditions:	One, and only one, LED lit. The number of the lit LED should be between 1 and 7.
Wanted outcome:	One, and only one, LED should be lit all the time. After the button press, the LED right of the currently lit should turn on, and the previously lit should turn off.

Test 2:	**Move left**
Action:	Press button 2
Preconditions:	One, and only one, LED lit. The number of the lit LED should be between 0 and 6.
Wanted outcome:	One, and only one, LED should be lit all the time. After the button press, the LED left of the currently lit should turn on, and the previously lit should turn off.

Test 3:	**Wrap right**
Action:	Press button 0
Preconditions:	LED 0 lit and all others unlit.
Wanted outcome:	After the button press, LED 0 should turn off and LED 7 should turn on.

Test 4:	**Wrap left**
Action:	Press button 2
Preconditions:	LED 7 lit and all others unlit.
Wanted outcome:	After the button press, LED 7 should turn off and LED 0 should turn on.

Test 5:	**No action**
Action:	Do nothing
Preconditions:	Precisely one LED lit
Wanted outcome:	The same LED should remain lit for the entire period of inaction, and no other LEDs should light up.

Test 6:	**Quick click**
Action:	Press button 2 fast 5 times
Preconditions:	LED 0 lit and all others unlit.
Wanted outcome:	LED 5 lit and all others unlit.
Comments:	This test was made to make sure the debouncing code doesn't interfere with normal speed-clicking