#EEDS
Report from Lab Assignment #1
TDT4258 Energy Efficient Computer Systems

Emil Taylor Bye, Sigve Sebastian Farstad, Odd M. Trondrud

February 13, 2013

SW1	Set to SPI0.
SW2	Set to PS2A/MMCI/USART1.
SW3	Set to SSC0/PWM[0,1]/GCLK.
SW4	Set to GPIO.
SW5	Set to LCDC.
SW6	Set to GPIO.
JP4	Set to "INT. DAC".
JP5	Set to "INT. DAC".

Abstract

This report presents our solution to assignment #1 of TDT4258 at NTNU, spring 2013. In the assigment, an Atmel STK1000 development board is programmed to display a moveable LED "paddle" using AVR32 assembly and the GNU tool chain.

Introduction

The objective of this lab assignment was to make a program for the STK1000 which allowed the user to control which of the LED diodes on a row of diodes should light up. This was to be accomplished by changing the selected diode to an adjacent diode, either to the right by pressing button 0 or to the left by pressing button 2.

In order to solve this task, it is essential to have a basic understanding of the STK1000 in order to correctly set up the hardware for input/outut. It is also important to possess basic knowledge of assembly for the AVR32 in order to be able to program, and more specifically how to handle interrupts in assembly in order to achieve high energy efficiency.

Description and Methodology

Configuration of the STK1000

Jumpers

The STK1000 has (number) jumpers that can be set to configure the board. For this assignment, the following settings were set for the jumpers:

GPIO connections

The STK1000 provides a general purpose input/output (GPIO) interface with 32 signal lines. 16 of the 32 available lines are in this assigment connected to on-board I/O devices on the STK1000. These devices are 8 on-board LEDs, which will be used as a rudimentary paddle display, and 8 on-board switches, which will be used as player controls.

The buttons are connected to GPIO0-GPIO7 (J1 on the STK1000) using a flat cable as in (diagram X). This maps the buttons to ports 0-7 of PIOB. The choice of low port numbers 0-7 is convenient for coding, and the choice of PIOB as opposed to PIOC is purely mnemonic ('B' for buttons).

The LEDs are connected to GPIO16-GPIO23 (J3 on the STK1000) using a flat cable as in (diagram X). This maps the LEDs to ports 0-7 of PIOC. Having the same port numbers for the buttons and the LEDs is a nice convenience for cleaner code and more efficient code, as translation from button ports to LED ports is not necessary.

(diagram X), note the orientation of the flat cables

Programming environment

JTAGICE

The AP7000 sisterboard on the STK1000 provides a JTAG interface, which is used for programming and debugging of the board. The development PC was connected to the JTAG interface of the STK1000 using an Atmel JTAGICE mk II (firmware 7.29). The JTAGICE does not require external power as long as it is connected to the PC over USB.

GNU Debugger

We followed the instructions presented in the compendium in an attempt to employ the GNU debugger, but were confused by the interface as we had not read the recommended further reading material supplied in the compendium. So we chose to not employ the GNU debugger in the development of our solution. TODO: run up to the lab and use gdb a bit, so we can write about it here afterwards.

Make, other tools etc

Development of the program

This section details the steps taken during the development of the program. Initially, a bare-bones program was developed with minimal functionality, to get familiar with the environment. Features were added iteratively, starting with simple LED and button integration, and moving on to more sophisticated interrupt-oriented logic/program flow.

Setting up the LEDs

The connection to the output LEDs must be set up before the LEDs can be used in a program. First, the I/O pins that the LEDs are connected to must be enabled. In this assignment, the LEDs are connected to the pins GPIO16-23, corresponding to PIO C lines 0-7. To enable the correct I/O pins, we must therefore set to 'high' bits 0-7 of the PIO C PIO Enable Register (PIOC PER), as in listing x. Here, r3 is the base offset of PIOC, AVR32_PIO_PER is the PIO Enable Register offset, and r6 contains the bitfield indicating which pins to enable.

Listing 1 : Enable the I/O pins

```
99      /* enable IO pins for the LEDs */
100     st.w r3[AVR32_PIO_PER], r6
```

Second, the I/O pins must be set to act as output pins, as opposed to input pins. This is done by setting to 'high' the corresponding bits (0-7) of the PIO C Output Enable Register (PIOC OER), as in listing x. Here, r3 is the base offset of PIOC, AVR32_PIO_OER is the Output Enable Register offset, and r6 contains the bitfield indicating which pins to set as outputs.

Listing 2 : Set the pins to act as output pins

```
102     /* set the IO pins to be outputs */
103     st.w r3[AVR32_PIO_OER], r6
```

Once this is done, LEDs can be turned on by writing the appropriate bits to PIO C Set Output Data Register (PIOC SODR), as in listing x. Here, r3 is the offset of PIOC, AVR32_PIO_SODR is the Set Output Data Register offset, and r4 contains the bitfield indicating which LEDs to switch on.

Listing 3 : Switch on LEDs

```
176     /* turn on the appropriate LED */
177     st.w r3[AVR32_PIO_SODR], r4
```

Analogously, LEDs can be turned of by writing the approtiate bits to PIO C Clear Output Data Register (PIOC CODR), as in listing x. Here, r3 is the offset of PIOC, AVR32_PIO_SODR is the Set Output Data Register offset, and r4 contains the bitfield indicating which LEDs to switch off.

Listing 4 : Switch on LEDs

```
173     /* turns all LEDs off*/
174     st.w r3[AVR32_PIO_CODR], r6
```

Setting up the buttons

The connection to the input buttons must be set up before the buttons can be used in a program. First, the I/O pins that the buttons are connected to must be enabled. In this assignment, the buttons are connected to the pins GPIO0-7, corresponding to PIO B lines 0-7. To enable the correct I/O pins, we must therefore set to 'high' bits 0-7 of the PIO B PIO Enable Register (PIOB PER), as in listing X. Here, r2 is the base offset of PIOB, AVR32_PIO_PER is the PIO Enable Register offset, and r6 contains the bitfield indicating which pins to enable.

Listing 5 : Enabling I/0 pins

```
105     /* enable IO pins for the buttons */
106     st.w r2[AVR32_PIO_PER], r6
```

Second, the pull-up resistors for the buttons must be enabled. This is because ¡reason¿. This is done by setting to 'high' the corresponding bits (0-7) of the PIO B Pull-Up Enable Register (PIOC PUER), as in listing x. Here, r2 is the base offset of PIOB, AVR32_PIO_PUER is the Pull-Up Enable Register offset, and r6 contains the bitfield indicating which pull-up resistors should be enabled.

Listing 6 : Enabling pull-up resistors

```
108     /* enable pull-up resistors */
109     st.w r2[AVR32_PIO_PUER], r6
```

Once this is done, the button state can be read by reading the appropriate bits from PIO B Pin-Data Status Register (PIOB PDSR), as in listing x. Here, r2 is the offset of PIOB, AVR32_PIO_PDSR is the Pin-Data Status Register offset.

Listing 7 : Reading the button state

```
225     /* read button status from piob */
226     ld.w r12, r2[AVR32_PIO_PDSR]
```

Interrupt Routine

Our interrupt routine first reads the state of the buttons before calling a debounce routine which runs for some amount of time. When the debounce routine is finished the button interrupt routine notifies that the interrupt has been handled before returning to the main loop.
The debounce routine's running time can be modified by altering the DEBOUNCE constant, which can be considered the routine's loop counter. The routine keeps the CPU busy by repeatedly subtracting one from this value until it reaches zero. This prevents further interrupts from being registered for the duration of the debouncing routine.

In order to implement an interrupt routine we have to enable the relevant interrupts and tell the interrupt controller where it can find the interrupt routine. Interrupts are enabled for specific I/O units by loading the appropriate values into the Interrupt Enable Register. We want to enable interrupts for SW0 and SW2. Their masks are `0x1` and `0x4`, respectively.

We enable interrupts by loading the appropriate values into the Interrupt Enable Register (PER). Specifically, the äppropriate valuesäre the masks for SW0 and SW2, which are `0x1` and `0x4`, respectively. Since

our buttons are connected to PIOB, we load the masks More specifically, we load these masks into a register, and then
We load PIO B's base address into r2 `/* */ lddpc r2, piob_offset /* The masks of SW_0 and SW_2 are loaded into r5 */ mov r5, SW_0 | SW_2 /* */ st.w r2[AVR32_PIO_IER], r5`

`st.w r2[AVR32_PIO_IER], r5`

This is done by storing the appropriate values in the PIO Enable Register (PER). We enabled button interrupts for `SW0` and `SW2` by loading their masks into the PIOB, with the appropriate offset to specify GPIO pins = PIO general purpose input/output pins

lddpc r2, piob_offset lddpc r7, intc_base mov r8, button_interrupt_routine mov r5, SW_0 — SW_2 st.w r2[AVR32_PIO_IER], r5 st.w r7[AVR32_INTC_IPR14], r8

`st.w r2[AVR32_PIO_IER], r5` Enables button interrupts for SW0 and SW2. First, interrupts must be disabled

lddpc r7, intc_base

mov r8, button_interrupt_routine

st.w r7[AVR32_INTC_IPR14], r8

The time is specified First the state of the buttons is read. The debounce routine simply waits for a time decided by the DEBOUNCE constant in CONFIGURATION.

st.w rd, rs

et spesielt minneomrde som skal inneholde adressen til en rutine som vil bli kjrt nr kortet mottar en relevant interrupt. I dette tilfelle er relevante interrupts knappe-interrupts.

rcall read_buttons rcall debounce

ld.w r0, r2[AVR32_PIO_ISR]

rd, rs

rete

Refactoring and Modularization

Program flow

This section presents a detailed overview of the program flow of the final interrupt-oriented paddle program.

yuml.me-source for diagrams

Main program flow: (start)-¿(Init)-¿(Main loop)-¿(Main loop)

Init: (start)-¿(Load pointers)-¿(Set up start values in registers)-¿(Enable I/O with interrupts)-¿(end)

Main loop: (start)-¿(Set leds)-¿(Sleep)[interrupt]-¿¡interrupt routine¿-¿[Was SW0 pressed][Yes]-¿(Move paddle right)-¿(end),[Was SW0 pressed][No]-¿(Move paddle left)-¿(end)

Set leds: (start)-¿(Turn off all LEDs)-¿(Turn on the paddle LED)-¿(end)

Button interrupt routine: (start)-¿(Read button states)-¿(Software debounce)-¿(Notify that the interrupt has been handled)-¿(end)

Debounce: (start)-¿(Set register to a high constant)-¿(Is value in register equal to 0)[Yes]-¿(end),(Is value in register equal to 0)[No]-¿(Decrease value in register by 1)-¿(Is value in register equal to 0)

Read buttons: (start)-¿(Read button status)-¿(Mask away buttons that were pressed in previous interrupt)-¿(end)

Move paddle right: (start)-¿(Is paddle at far-right end of the board)[Yes]-¿(Move paddle to far-left)-¿(end),(Is paddle at far-right end of the board)[No]-¿(Move paddle one step to the left)-¿(end)

Move paddle left (analogous to move paddle right, but included for completeness): (start)-¿(Is paddle at far-left end of the board)[Yes]-¿(Move paddle to far-right)-¿(end),(Is paddle at far-left end of the board)[No]-¿(Move paddle one step to the right)-¿(end)

Register Overview

The STK1000 has 13 general purpose registers named r0-r12. The programmer is free to use these registers for whatever they want. However, several conventions are commonplace to introduce a certain degree of structure.

Conventions:

r0	Scratch register used to hold intermediate values.
r1	Constant: 0
r2	Constant: base offset to PIOB
r3	Constant: base offset to PIOC
r4	Variable: holds the position of the paddle
r5	Constant: 5
r6	Constant: 0xff
r7	Variable: holds the previous button state
r8	Constant: pointer to button interrupt routine
r9	(not used)
r10	(not used)
r11	(not used)
r12	Holds return value from routines

r0 is a scratch register, used for intermediate calculations and such. r1 holds the constant 0.
...
r8, r9, r10, r11, r12:
used to hold parameters when calling a routine.
r12:
used to hold the return value when returning from a routine.
List of registers and what they are used for in the paddle move program:

Experimental Procedure

The first thing we did was locate the lab, on the fourth floor of the IT-Vest building, room ###. We removed one of the AVR32STK1000 boards from their box and set the jumpers to the appropriate positions (see page 37 of [**?**]). We booted up one of the computers and connected the AVR32STK1000 to it. Our first piece of code simply enabled all the LEDs and turned them on. Once we got the LEDs working we enabled the buttons. We followed this up by writing code to turn on just one of the LEDs, which designated the "paddle" per the assignment's description on page 37 [**?**]. Around this point we stopped using GDB because we couldn't figure out how it works.

one of the LEDs We wrote code to enable all the LEDs. Once that was working we enabled the buttons. Just one LED was enabled. Button 0 moves the paddle right (logical shift did not work, used arithmetic shift). Button 2 moves paddle left. When either button is pushed down the board registers it as a fuckton of presses, causing the paddle to fly off. We wrote code so that the paddle gets moved to the opposite side of the LEDs when it hits the edge. This was all done with busy waiting. We then implemented debouncing. At this point we had a problem where each button push was registered as two, causing the paddle to move to LED_n+2 rather than LED_n+1. We ignored that and implemented interrupts. After which we went through our code and inserted comments. Then we made it even nicer looking. And finally we fixed the issue with the paddle moving two LEDs per push by masking or something idk Sigve solved this one.

Results and Tests

This is the results and tests section.

Energy Efficiency

TODO: we should get an amperemeter and measure actual efficiency of our programme with busy loops, sleep etc

Testing

Evaluation of Assignment

Here comes the evaluation of the assignment.

The assignment was very nice (Y)

more information regarding what we should not do in the lab e.g. don't turn off the computers and don't use sleep 5.

Conclusion

This is the place for the conclusion.

Acknowledgement

Here, we acknowledge those who deserve to be acknowledged.

References

And finally, references.