A project report for morse code on processors

INTRODUCTION

This project will look at code for translating English words and letters to standard morse code. The approach taken is to first code the project in C language on the ATEMGA328p on the grove board, we then build on top of this by adding an interrupt into the project, the final step was to add inline assembler code into particular places where math operations occur. These operations mainly have to do with translating a letter to a particular value in our ‘look up table’. We will attempt to compare these implementations. Another aspect is also to try to transpose some of this code to PIC32mx250f128b architecture and compare our codes to the ATEMGA with some limitations.

CONTEXT:

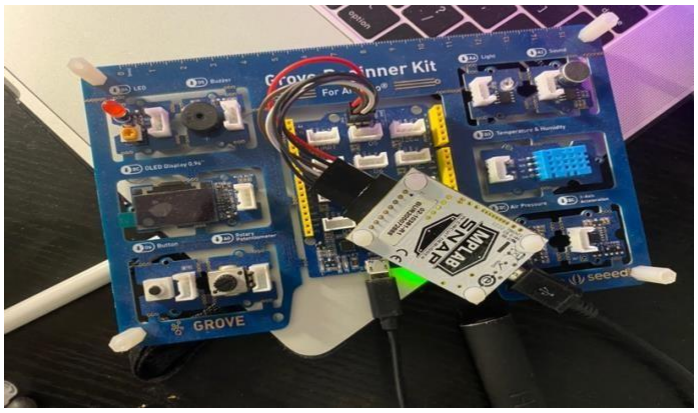
The project is a morse code translator. It takes an input (for our particular project the input is directly fed into MPLABX). The input then in converted with a set of functions to create instructions for the LED on our ATEMGA328p and also PIC32 (in simulation).

We have various versions of the code for both the ATEMGA and PIC32. The versions include some with just C, some with C and interrupt or C + interrupt + inline assembler. The functionality of all of these is the same, it will translate English letters to morse code signals on the LED.

The reason for doing this project are quite vast. First, being a Physics and Computer science student, I would want to tackle the major project in more of a software approach and incorporating the things learnt from this class in that context. This project allows me to that with minimal hardware due to how difficult it was to acquire some pieces of hardware.

Next, this project is scalable. It allows me to work on this project and keep developing more complicated versions of this with more inline assembler or more complicated interrupts in the software. It would be interesting for me to see how various coding styles and languages directly affects hardware while adding more functionality.

TECHNICAL REQUIREMENTS / SPECIFICATIONS



Figuire 1: A photo of our GROVE system which contains the LED, MPLAB snap programmer connected and an LED that we use to output the morse code on D4. The buzzer can also be seen which we later was phased out due to consideration of not being able to adjust the buzz time accurately

A project report for morse code on processors

What should this system do?

- It should be able to convert any words or letters of the English alphabet on the LED for the grove board (ATEMGA versions)
- Similarly, the PIC32 versions should be able to convert words to morse code seen in simulation of an LED
- There is standard time for delays between words and letters in morse code. However, this project is adjustable, for all versions the time between letters, words, and sentences can be adjusted using the various parameters of the program.
- Able to compare different implementations of the morse code method between two processors using mixturs of C, C+interrupt and C+asm

What was in this system?

- GROVE beginner kit which includes the ATMEGA328p
- MPLABX snap programmer
- USB cables x2
- LED on grove board
- Buzzer (limited functionality)
- MPLABX IDE for debugging and running program
- PIC32 simulation with logic analyzer tool
-

PROCEDURE:

- Describe the process that you used in creating your project.

  First, we write the program in C for both the pic32 (in simulation) and atemga328p. This allows us to do some comparison of this program performance on both archeitures. We find when comparing this that executing the main method of the function requires about 56 instructions on the ATMEGA and only 36 instructions in PIC32 according to our disassembly. It seems our PIC32 is performing the morse code translation more efficiently.

  Second, we write an ISR (interrupt service routine) for the ATEMGA code to be integrated with the C to see how this would affect the disassembly. The best thing to compare this to is the C code which uses the <util/delay.h> library. Essentially, we are replacing this with an ISR. In this case the main method required only 40 instructions when compared to the the fully C version with <util/delay.h> which needs 56 instructions.

  When comparing the asm inline version with the interrupt version there is no within the disassembly window. There are 40 instructions in disassembly for both C+Interrupt and C+Interrupt+inline assembler. However, we should look at another method here where the inline asm is actually used.

  Instead we look at the encode_ch function method where we use inline assembler with C+interrupt and compare it to the version where just C+ interrupt is used. Here we see 125 instructions for the former and 118 instructions for the later. We see that using inline assembler has reduced the number of instructions needed to perform the method.

TEST and COMPARE:

The testing was mainly done with the debugging tool on MPLABX. Various inputs were used which in our case are different words in "*const char *text[]"*. The main source of error is usually the mapping from input to the lookup table, this is important because this is the foundation of the program and thus our base level fail case. The other functions all use this input.

After this looking at the LED was the main source of checking and comparing our LED output to what we expect from morse code.

Once this is complete, I pay attention to the timing between the letters, words, and sentences (although sentences were not really tested as they take a long time to check. If words and letters timings work, it is almost certain that certain are going to work as they are simple repetitions of letters and word test cases.). We use a stopwatch to make sure our timing is correct. In standard morse code the time constant between parts of the same letter is one unit, time between the letters is three units and time between the words is 7 units. We need to see that roughly ratio of the times between all these parameters is consistent for the morse code to be read sensibly. The way I approached this was to mainly measure time. The adjustment of the time between all these factors by simply changing parameters is useful here so that we shorten everything and just make sure the ratios of the times makes sense.

In order to do testing of performance and comparison we use the disassembly window. Once we debug with a breakpoint on a particular part of the code like the main method, we can initiate this to translate into machine code. Here we assume that each line in disassembly is one instruction, therefore the more lines, the more instructions that particular processor requires to excute said function. We can judge efficiency of this program in this manner. Of course, we must assume the same clock speed in all our comparisons. Usually we would use lab tools, logic analyzers and oscilloscopes to do better comparisons but we are limited in our techniques at the moment.

Another form of testing was the simulation required for the pic32 simulation. For this because of hardware limitation we had to use the logic analyzer for the output. In the picture below we display the out for one of our tests where we translate "SOS" into morse code. The patter expected is dotdot-dot---dash-dash-dash---dot-dot-dot. We see below that this was successful, and our timer handled the timing between the parts of the same letter, parts of different letters correctly. This leads us to be certain that it will also be able to handle sentences correctly.

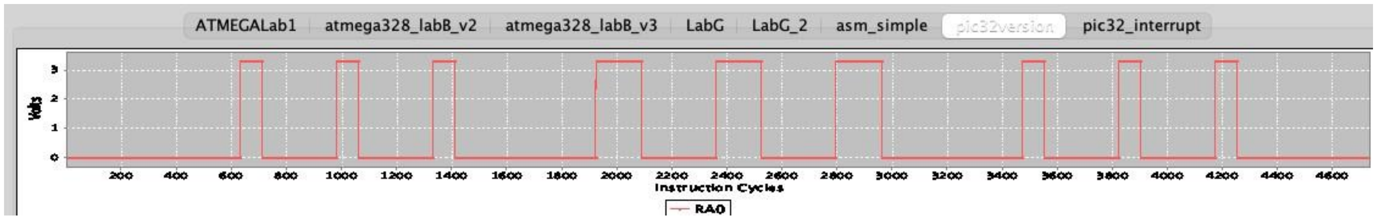


Figure 2: The output from MPLABX logic analzyer outputting the word "SOS" to morse code.

Our testing of the performance of the code allows us to conclude a few things. In our scenario it seems that the PIC32 is faster in handling C code than the ATEMGA328p, this was expected but it is interesting to see the output. The pic32 does use a modified Harvard architecture compared to the traditional Harvard architecture that the ATMEGA uses. There is no one size fits all micro controller but in our scenario the speed increase is likely due to the fact that the AVR uses 1 clock/instruction cycle while the PIC has a speed of 4 Clock/instruction per cycle.

The PIC32 also has a 32-bit bus width compared to the ATEMGA 8-bit bus width. This makes for example 32-bit calculations more tedious on the 8 bit ATMEGA as it has to use more registers and repeat processes giving it more instructions. The bus is a pathway for data and there are three different types data, address and control bus. In relation to the data bus, the bus width typically relates to data transfer. The width of the bus will usually directly correlate with the maximum amount of data that can be instantaneously processed and sent at one time. A 32-bit bus can deliver 32 bits of data at simultaneously for example and it is important to note that this data transfer is bidirectional. The address bus is also existent and carries addressing signals from the processor to memory and I/O.

Another aspect to compare was the consideration of I/O use in the both these microcontrollers. Both these processors allow for serial communication. The older version of this was the parallel communication method which is a bit outdated. Parallel usually allows transmitting large packets data over several parallel channels like sending 8 bits or 1byte at a time. This is expensive although it is quite fast, it is usually not worth it in our case unless we have a specific considering in the project. To further elaborate, the parallel form is also short distance though "N" channels although this was not very relevant to our project. Serial communication allows for bidirectional movement with higher resolution of data (bit by bit) where each bit would have a clock pulse rate. Since the resolution is higher in serial it is by far the one that produces the least noise and error in the data making them simpler and more reliable in the long run.

The good thing about the PIC is due to the architecture it has a smaller instruction set although for some implementations this can still be difficult compared to the atmega. The atmega has the caveat that many operations can only be done on certain registers making it "non-orthogonal". The interrupt latency is more or less constant in our testing for the PIC although this was simulation we can understand this from research as well, usually with the AVR family of micro controllers you often have a one cycle jitter when you're dealing with some larger inputs we see this in our program. The counters on the PIC can also be asynchronous, but the AVR family cannot wake up from edges except on certain parts that have pin change.

CONTINGENCY

Initially there was an idea to have multiple unit testing for the board (equivalent to j unit testing in java). This allows for the project to be more complete. It would also cover any cases that maybe we did not anticipate for in our manual testing. This is a very standard procedure of software development however due to time constraints and difficulty with MPLABX it was an idea that had to be scrapped and replaced with brute force

Secondly, the implementation of the buzzer. There were issues with the volume of the buzzer that was very hard to pick up on video. More importantly, I was having difficulty implementing longer 'buzzes' and shorter ones to output morse code effectively. In hindsight, it may have been useful to use pulse width modulation for this so we could affect how long and short buzzes are. Instead, for some of the ATMEGA main code I use the buzzer as a sound to just keep up with the morse of the LED as this is triggering two stimuli of sound and sight and its easier for a beginner in morse code to keep up with this

Some lessons I learnt here was simply to effectively explore the scope of the project outline in more detail before starting. If some things do not seem doable in a project time frame it may be best to cut it out unless it is a mandatory feature. In my case for this computer architecture course it usually was not. This caused a lot of time wastage so in the future it would be good to remember that this time could instead be transferred to the fundamentals of the project.

ADDITIONAL MATERIAL

Samuel morse was one of the first pioneers of the telegraph. When Samuel morse got his message that his wife had died when he was away for work it was already to late for him to return home, this was the reality of long-distance communication around 1791. Morse code is not really used by people nowadays besides the occasional hobbyist. It is usually seen as a toy nowadays to once it was once seen as. Even the US coast guard stopped using this communication around 1995 however it is still used for emergency signals in intelligence operations and radio operators. It is also used for distress signals like “SOS”.

More modern uses of morse code is as assistive technology for the disabled, allowing users to input morse code instead of a keyboard, this is found on android 5.0 and up. It makes it an effective tool for distress or general communication for most people with sever mobility issues and can be used in care homes.

There is a lot of misinformation and unclear procedures for interrupt service routines available I have found. Some books like AVR programming do a good job but it is unfortunate that they are only in English. The realm that people are limited by language to gain education is quite a frightening thing and so I have translated the interrupt article on Wikipedia (https://en.wikipedia.org/wiki/Interrupt) to Amharic, my native language in Ethiopia, I hope this will be of use to many people as hours were spent to make sure the correct translation was implemented for engineering terms. The article can be located here (https://am.wikipedia.org/wiki/Interrupt) . It was sad to see that there is not much actual educational content written in this language, and this project opened my eyes to this issue. I will definitely be pursuing this as a hobby from now on to translate educational content to Amharic in an effort to make education more accessible for all . The article was translated up to the “software interrupts” sub section.

CONCLUSION

In the end there is no one size fits all microprocessor unfortunately, or maybe that is a good thing because then specialization would not be a thing in science. In the research conducted, I realise that we do not have a perfect micro controller, we need to see the requirements of our project and see what is required in order for the task to be handled the best way possible.

In our testing we saw that when it comes to C the PIC32 is more efficient in running our programs, the hypothesized reasons were described along with how this may relate to the serial I/O input capabilities. Although both would be able to run this feature, the data transfer is therefore more relevant.

As expected, we saw that writing inline assembler commands is usually a good idea despite how tedious it maybe. It reduced our instruction set needed to encode our English words to morse code, this is typically what is done by embedded system engineers. I would say in modern day the reduction on time is average due to forcing the user to write shorter, more efficient code that directly talks to the machine since compliers are becoming very smart and are able to optimize code written at a higher level. You could say compliers have spoiled us a bit in writing ‘sloppy’ code. However, there are numerous scenarios as we have shown that inline assembler in particular can be very useful.

A few references:

1) *What is clock cycle, machine cycle, and instruction cycle* ... (n.d.). Retrieved December 16, 2021, from https://www.quora.com/What-is-clock-cycle-machine-cycle-and-instruction-cycle-in-a-microprocessor
2) Thornton, S., 2021. *The Internal Processor Bus: data, address, and control bus*. [online] Microcontrollertips.com. Available at: <https://www.microcontrollertips.com/internal-processor-bus-data-address-control-bus-faq/> [Accessed 16 December 2021].
3) Tech Differences. (2016). *Difference Between Serial and Parallel Transmission (with Comparison Chart)*. [online] Available at: https://techdifferences.com/difference-between-serial-and-parallel-transmission.html.
4) www.microchip.com. (n.d.). *Smart | Connected | Secure | Microchip Technology*. [online] Available at: https://ww1.microchip.com/downloads/en/DeviceDoc/MPLAB%20XC32%20C%20Compiler%20for%20PIC32M%20MCU%20UG%2050002799B.pdf [Accessed 16 Dec. 2021].
5) M I P S Reference Data. (n.d.). [online] Available at: https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf.

   microchipdeveloper.com. (n.d.). *Instruction Set Architecture Overview - Developer Help*. [online] Available at: https://microchipdeveloper.com/16bit:instruction-set-architecture [Accessed 16 Dec. 2021].
6) J3 (2019). *What is the major difference between PIC and AVR?* [online] Jungletronics. Available at: https://medium.com/jungletronics/what-is-the-major-difference-between-pic-and-avr-36c73e314042 [Accessed 16 Dec. 2021].
7) www.microchip.com. (n.d.). *Unit test | Microchip*. [online] Available at: https://www.microchip.com/forums/m551244p3.aspx [Accessed 16 Dec. 2021].
8) Ellliot Williamson (2014). *Make : AVR programming*. Sebastopol: Maker Media. C.
9) AVR Microcontrollers AVR Instruction Set Manual OTHER Instruction Set Nomenclature. (n.d.). [online] Available at: http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf.
10) Wikipedia Contributors (2019). *Interrupt*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Interrupt.
11) passportyork.yorku.ca. (n.d.). *Passport York Login*. [online] Available at: https://eclass.yorku.ca/eclass/pluginfile.php/3041570/mod_resource/content/1/LabH_EECS2021_PIC32_MixC_Student_VERSION_v3_2021.pdf [Accessed 16 Dec. 2021].
12) Tech Explorations. (n.d.). *Can you use delay() inside Interrupt Service Routine?* [online] Available at: https://techexplorations.com/guides/arduino/programming/delay-print-in-isr/ [Accessed 16 Dec. 2021].
13) Retrocomputing Stack Exchange. (n.d.). *When did compilers start generating optimized code that runs faster than an average programmer’s assembly code?* [online] Available at: https://retrocomputing.stackexchange.com/questions/16153/when-did-compilers-start-generating-optimized-code-thatruns-faster-than-an-aver [Accessed 16 Dec. 2021].
14) microchipsupport.force.com. (n.d.). *Microchip Lightning Support*. [online] Available at:

https://microchipsupport.force.com/s/article/KB100052 [Accessed 16 Dec. 2021].