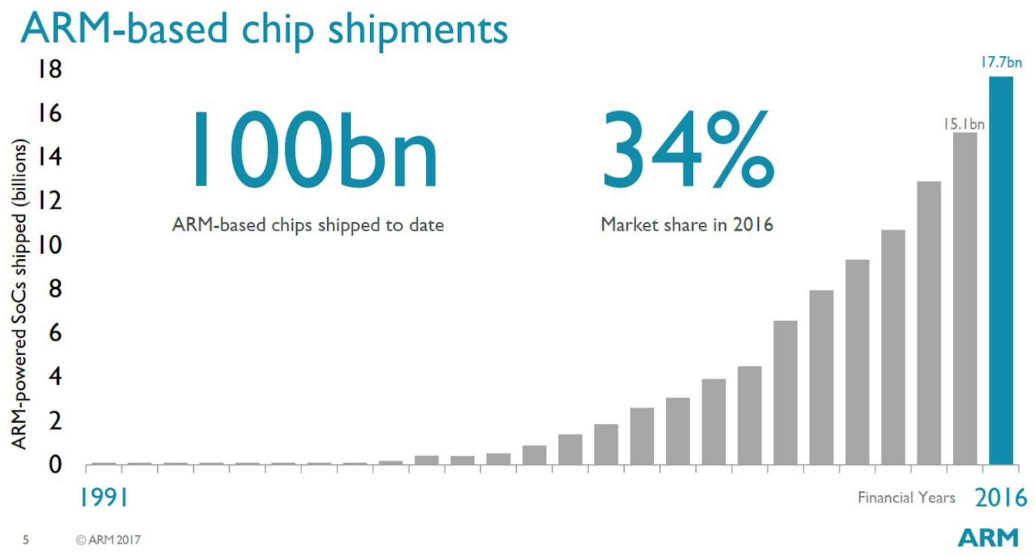
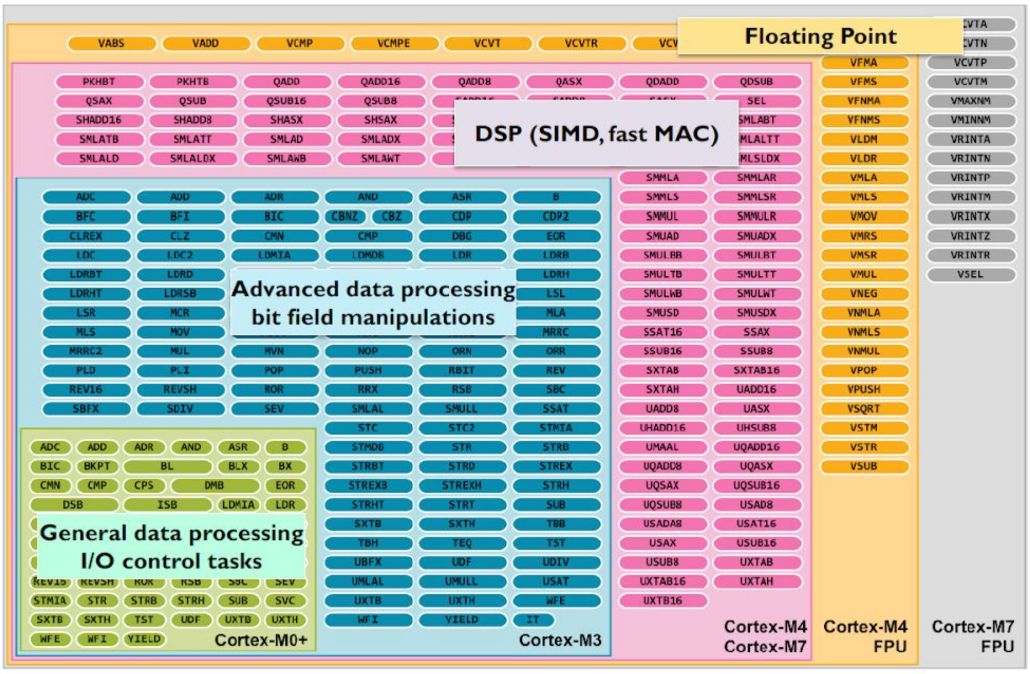
ARMed to the teeth #1 - Why go ARM?

The ARM family of processors are everywhere. Almost all modern mobile phones, TVs, home routers use at least one ARM processor. The growth of ARM powered devices has been explosive with ARM shipping nearly 18bn ARM powered chips in 2016.



ARM device growth

For many years the use of ARM devices was closed off to the hobbyist due to proprietary development tools which could only be licenced for thousands of pounds. Fortunately, the growth and widespread nature of ARM devices has lead to them being supported by popular open source tool chains like GCC, and with this came an increase in free IDEs offered by many manufacturers.
For the microcontroller and hobbyist markets, the 32-bit ARM Cortex-M is the most suitable device. The Cortex-M family covers a wide range of microcontroller applications and is rapidly eroding a market that was once dominated by 8-bit microcontrollers such as the 8051, ATmega and PIC.
But what's the attraction? Why should hobbyists be looking at using ARM microcontrollers for their next projects? Well there are several reasons, and this article will look at a few, but first let's have a more details look at the Cortex-M family.
Currently there are four main types of devices in the Cortex-M family, the M0+, M3, M4 and M7. The series is designed to cover as many microcontroller applications as possible, ranging from ultra low power in the M0 series, to extremely fast and complicated devices at the M7 end. Take a look at the graphic below, this shows the size of the instruction set for each of the Cortex-M devices.



Cortex-M family instruction set (©ARM)

Typically, M7 devices are more expensive and faster than M4, M4 more so than M3 and so on. It helps to have an understanding of what your wanting to achieve with your project to guide which Cortex-M processor you need. For example, if your doing a lot of floating point maths, it would be worth considering the M4F or M7 over the M3 or M0.
You may have already noticed that many popular IC manufacturers are offering ARM based microcontrollers, so why should a M4 from Atmel be any different from a M4 from ST? At this point it's worth pointing out that ARM don't actually make any physical devices, instead they licence their processor architecture for semiconductor manufacturers to integrate into their designs. This means that a Cortex M4 from Atmel will have a different set of peripherals/busses/clock systems etc in it that the M4 offering from ST. To try and make an offering that stands out in a fairly crowded market place, manufacturers may aim for specific markers, tailoring the on chip peripherals to match the target market. For example Silicon Labs target their Gecko series at low power devices, this may mean that the internal busses and flash storage may be simpler and lower power and maybe slower than other Cortex-M based devices.
So this is the first reason for choosing an ARM for your next project, there is a huge selection of devices to choose from. It's very likely you can find a device that suits all your requirements for flash/RAM/speed etc, whilst maintaining a suitable price point.
The next benefit to using ARM is free tools. Most of the big manufacturers will now provide a complete IDE and toolchain package for free, these will often be based on the GCC compiler set. This might seem a bit of a moot point since Microchip already give away free tools for developing on their 8-bit solutions, and the Arduino IDE can be used from a web page, but the addition of development boards allows these free IDEs to become hugely powerful development systems.

IC Manufacturer	Development platform
Atmel	Atmel Studio (free)
Silicon Labs	Simplicity Studio (free)
ST	Atollic (free)
Texas Instruments	Code Composer (free - limits code size)

Some free IDEs for Cortex-M devices

Which brings us to the next point in their favour; ARM devices are available on a range of cheap development boards. Maybe not as cheap as an Arduino clone, but there are significant advantages to paying the £10 or so extra for an Cortex-M development board. Typically, Cortex-M development boards will come with on-board programmers. These are tools that allow you to squirt your program onto the chip, but it gets better, these are also fully featured in-circuit debuggers. This means that you can debug your program at runtime, whilst the code is executing on the hardware. Additionally, you can look at values of variables, insert breakpoints and see the contents of the microcontrollers memory. All this is possible using the included debuggers. Some development boards even allow you to use the debugger as a serial interface to the PC, and some even provide external connections to allow it to program devices on your own PCBs.

Finally is performance, and this is where things start getting really interesting. The standard hobbyist micro is the good old ATmega328p, an 8-bit micro capable or running up to 20MHz. The cheap Cortex-M0+ based micros are usually capable of running up to 24 or even 48Mhz, have more flash and RAM, but more importantly are 32-bit.

Let's look at this in a bit more depth. An 8-bit microcontroller means that the registers that hold the data in the processing core are 8-bits wide, and typically each instruction takes 8 bit values as an input and saves 8-bit values as as output. But there's a limit to how useful 256 values can be, and often you need variables that go to higher number. Going to 16-bit values means that the 8-bit micro is already working overtime, maybe having to run two or more instructions, and using more CPU registers to process these values. Sometimes even 16-bits isn't enough and 32-bits needs to be used. This is now 4 times as much data as the CPU is designed to handle, so the compiler has to insert many extra steps to be able to deal with these values. All this extra work obviously has an impact on the performance of your program.

But what about the 32-bit ARMs? Well, the CPU registers are all 32-bits wide, and the instructions natively deal with 32-bit values, so it's easy to assume that both 8 and 16 bit data is automatically as fast, but this isn't quite the case. There's a downside, because everything is arrange in 4-byte blocks for quick 32-bit access, if you only want to address a single byte you need to make sure you don't overwrite the remaining 3 bytes in the same 32-bit slot. So, in the same way that 8-bit get slower on wider data types, 32-bit micros get slower on smaller data types.

But does this mean that the ARMs can't been the 8-bit ATmega at its own game? Let's take a look at some simple performance tests:

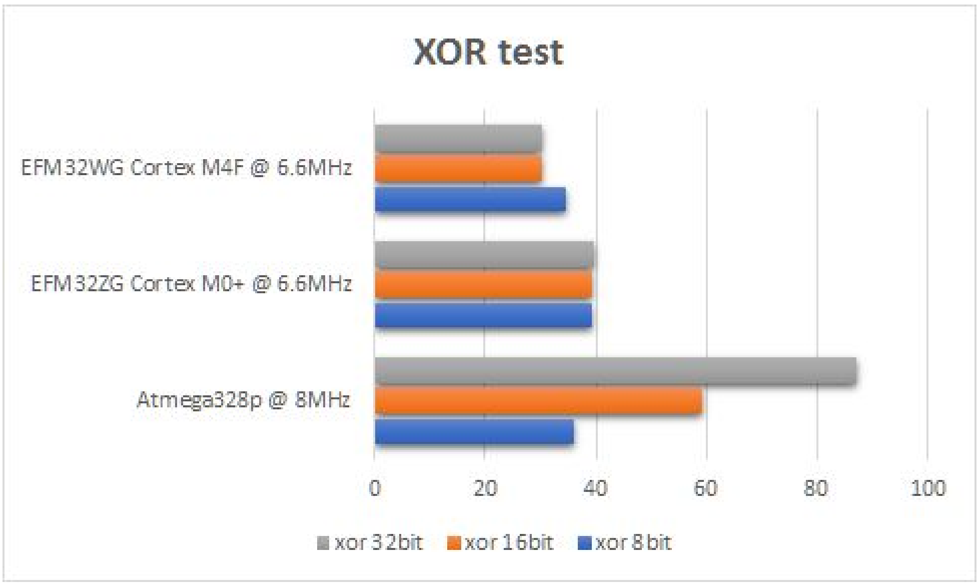
Test hardware:

1. ATmega328p @8MHz
2. EFM32ZG (Cortex-M0+) @6.6MHz
3. EFM32WG (Cortex-M4F) @ 6.6MHz

Test Setup:

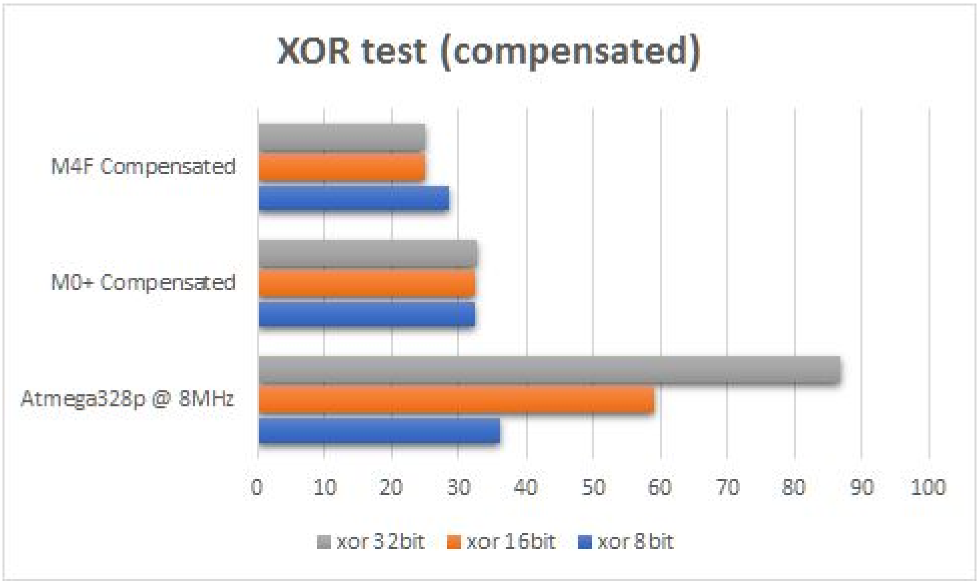
1. Atmel Studio 7.0 using AVR8-GNU-toolchain V3.6.1.1750
2. Simplicity Studio 4.0 using ARM-GNU-toolchain V7.2.1
3. -O1 optimisation level
4. Direct call to GPIO to set pin before execution of test function. Timing measured using Saleae Logic Pro 8 @250MSPS

Our first test is nice and simple, it takes 30 numbers and generates the XOR of all the values. The test was run three times, firstly using 8-bit data, then 16 and 32 bit data.



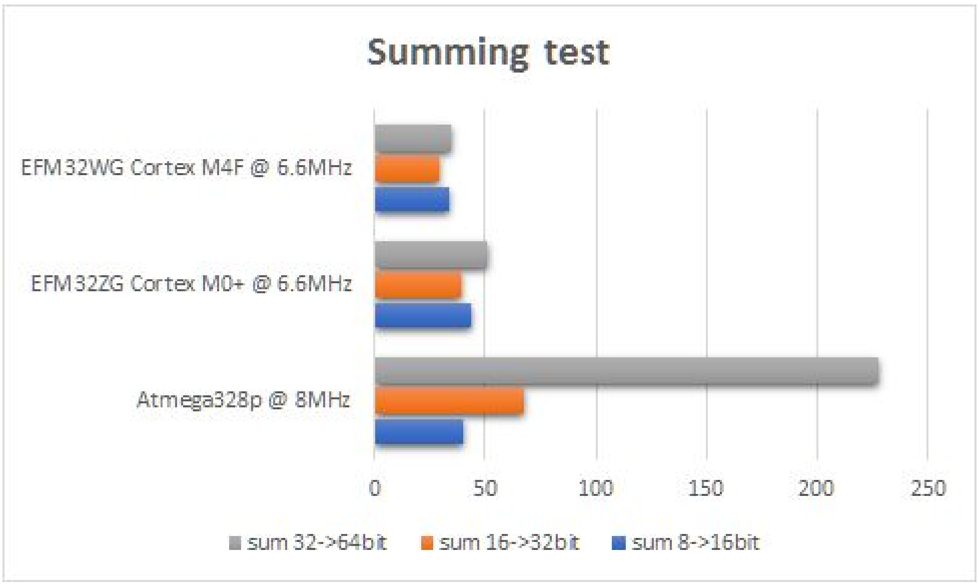
Execution time in microseconds

We can see that the ATmega wins with 8-bit values, but quickly falls behind with the 16 and 32 bit values. The M4 is faster than the M0 as it has more instructions for handling 16-bit values. Does this mean that the 8-bit matches the performance of the mighty M4 when dealing with 8-bit values? Not quite, look back at the test setup, the ARM micros are running at a lower clock than the ATmega, if we compensate the ARM values for the slow clock, we get the following:



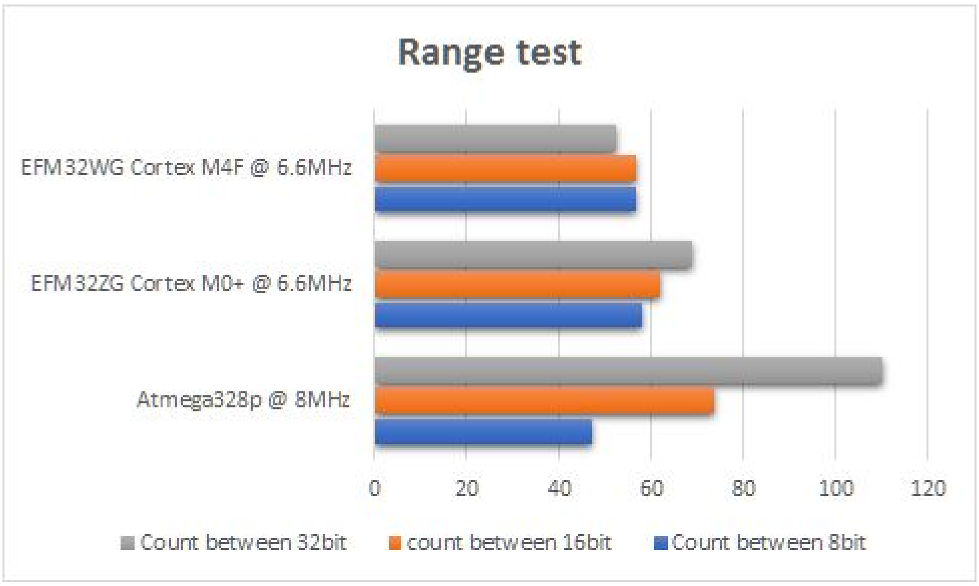
Execution time in microseconds

Compensating for the slow clocks, even the M0 is winning out against the ATmega.
Moving on to the next test, this once again takes a list of 30 values, but this time sums them into a total. What's important here is that the resulting value has to be of a greater width than the input values, so the 8-bit values are summed into a 16-bit result, 16-bit into a 32-bit etc.



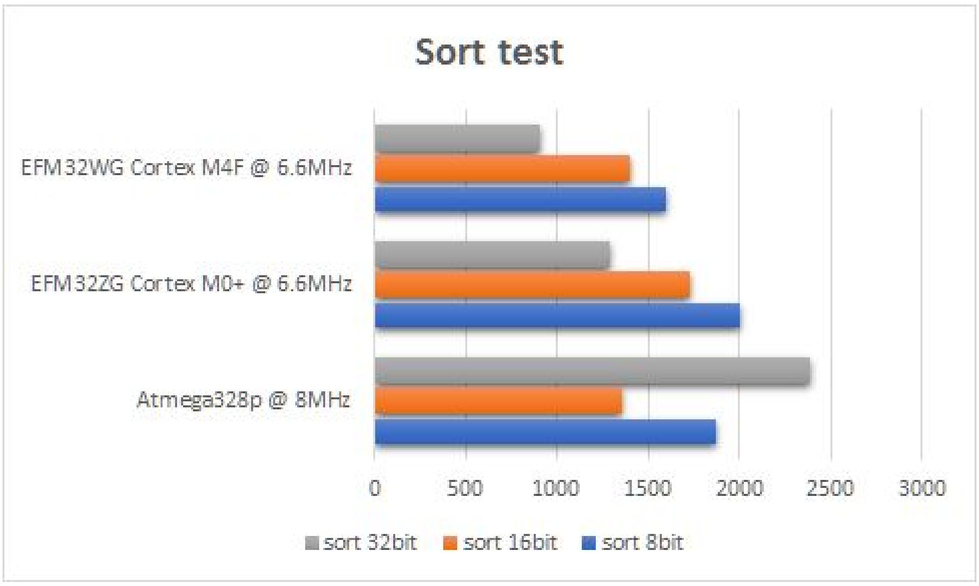
Execution time in microseconds

The ATmega really suffers here, especially on the 32-bit summing where the output is being accumulated into a 64-bit with value. The ARMs suffer a little by having to use 64-bit values but it's only a fraction of the total execution time.
Adding an extra level of complexity, the next test performs a conditional test on the 30 input values and counts the number of entries that fall within a specific range. Once again this was conducted for 8, 16 and 32 bit input values.



Execution time in microseconds

This test yields some interesting results with the M0 clearly getting slower as the data width is increased. This maybe due to the cache having to load more data when the branching condition is incorrect. Compensating for the slow clock speeds makes the 8-bit tests on the ARMs execute in exactly the same time as the ATmega.
The final test takes the list of input values and performs a simple bubble sort on the data. This is quite intensive as it involves multiple iterations of the data and conditional instructions.



Execution time in microseconds

The ATmega performs incredibly well on 16 bit values, and I've yet to come up with an explanation for this, it even beat the M0 when compensated. On 32 bit values though, the sort is taking nearly twice as long as the M0 and 2.6x as long as the M4.

So what are the downsides to using Cortex-M for your next project? Typically two fold, one is the cost with ARM devices usually being a little more expensive than their 8-bit counterparts. Secondly is learning curve, some Cortex-M devices can be several lines of code just to setup the clock distribution to get a simple 'blinky' working. And for the breadboard fans, good luck finding a Cortex-M in a DIP package!

Thanks for reading. In the next installment I'll be looking at device selection, installing the IDE and getting the first blinky running.