> Author: Jared Dyreson

> Class: CPSC-240 09 @ 11:30 - 13:20 TR

ST-Micro STM32F103

This processor was created in the mid 2000's and is apart of a family of microcontroller ICs that are derived from the RISC architecture (Reduced Instruction Set Count). Chips on this architecture generally have small instruction sets and tend to have faster responsiveness compared to x86. This processor in particular is apart of a broader family denoted as "ARM Cortex M3" .Those aforementioned processors use a much larger ISA as they can afford the space. The maximum speed of this particular CPU was 72 MHz, which in today's standard is not very fast at all. It however has access to a full USB interface and can control external devices. These processors also are related to the CPUs found in modern smart phones such as the iPhone and other low power devices.

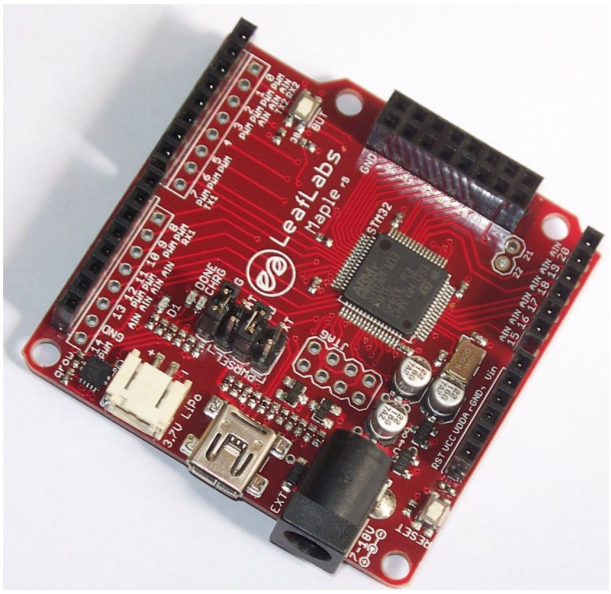

Figure 1: Example Board

Number of Registers

There appear to be 16 general purpose registers in this architecture. Registers R0 - R12 are used as containers for storing the operands of instruction ans to store the results of these operations. There is a distinction between high and low registers. R0 - R7 are low and R8 - R12 are high. There are some restrictions placed when using the high registers so it is common practice to just use the first 8 without messing with these impedances. The size of these registers is of 32 bit size, which is noted in the processor naming scheme.

Bus

An interesting feature of this architecture is the inclusion of a bus matrix. This allows for multiple buses to be linked up in one major highway. The different types include; ICode, DCode, System Bus, and the Private Peripheral Bus. All buses on this chip are of 32 bit size. The most intriguing one is the Private Peripheral Bus (PPB). It's main job is manage all of the different peripherals or extensions of the system at a given time. It uses the AHB protocol which is less efficient but overall can get the job done. This specific protocol was developed by ARM themselves.

Architecture Style

The lineage of ARM Cortex-M has changed it's instruction variance. M0 - M2 use the Von Neumann architecture but then it changes to implement Harvard until M7. This was likely due to the advancements in squeezing more buses on a given chip.

Floating Point Arithmetic

This processor family does indeed support floating point math however it is only single precision (floats only). We start to see some commonalities across this report, that the overall size of our registers and buses constrain our mathematical abilities. That's why there is single/double precision, some architectures can only, at a hardware level, support 32 bits of information.

Code Density

This is something I have never heard of before and this is how many instructions does it take to perform a specific task. I mentioned this before in the introduction stating that RISC architectures will perform better and this is why. If there are less instructions, then you get creative in using the limited amount you have available. Fundamentally, ARM Cortex has access to Thumb 1 and 2 instructions. This means that in T1, you have access to 16 bit registers and 32 bit registers in T2. These effectively scale your code density by a factor of two, gaining more fine grain control of speed and optimisation.

Stack Operations

Like all other architectures, ARM Cortex has a stack in which it executes code sequentially. Generally, we have the same sort of stack operations we saw in NASM such as POP, PUSH. There are some differences, such as point to memory regions uses ADR whereas in NASM it would use lea (load effective address). Both are the same in this regard but are in ARM it is not restricted to a hard coded registers (rbx in NASM), which opens up more versatility of registers. In the case of return and call, we use a combination of the `lr` register and the `bl` instruction which effectively store the processor context, allowing for program flow to resume and begin. Passing parameters would be done using registers r0-r3 and registers r4-r11 must be preserved during function calls, effectively limiting the amount of registers you can use. This helpful diagram shows a more in depth explanation:

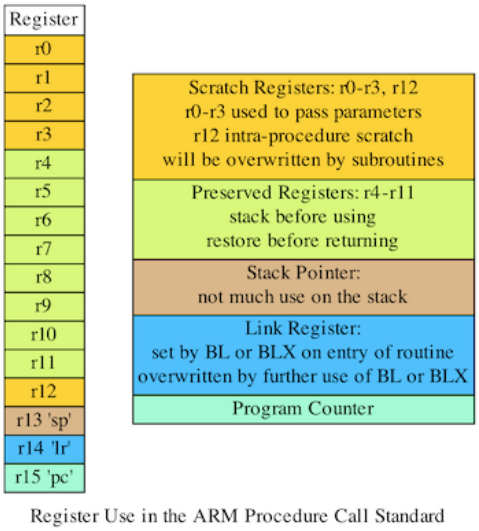

Figure 2: Function calls at a register level

Example Code

```
1 LDR R0, =0
2 LDR R1, =10
3 _start:
4 ADD R0, R0, #1
5 CMP R0, R1
6 BNE
```

This sample code uses a few key differences between how we have been coding in class. Firstly, our `mov` instruction is now replaced with the “LDR” (load register) operand. Also, the syntax is slightly off as we include an equals sign to denote the register’s value. Our entry point label is the same as if we were using GCC to link our NASM program. Also note that all operands are uppercase, not lowercase like in NASM. Adding any particular value to a registers you need to use a pound sign which is something that AT&T syntax uses which we do not see in Intel based syntax. Lastly, we use the BNE (branch not equal) operand as our conditional jump statement. Both of these are fundamentally the same and pose no performance improvement.

External Links

Example Code

LED Example Code

Wikipedia Page

Jumping and Branching

Code Density Explanation

Introduction to ARM Thumb

Function Calls in ARM