Introduction to Microcontrollers - Further Beginnings

Mike Silva • September 1, 2013

Embedded Programming Basics

This tutorial entry will discuss some further embedded programming basics that you will need to understand before proceeding on to the LED blinky and other example programs. We will do this by looking at the general organization and types of instructions found in most microcontrollers, and how that organization and those instructions are reflected (or, in some cases, ignored) by the C programming language.

Quick Links

Basic CPU Organization

When you write a program for your microcontroller you are really writing a program that is executed by the µC CPU (central processing unit), executing various instructions designed into the CPU instruction set, so it's worth a bit of time to take a brief look at the innards of a typical µC CPU. If the program is written in ASM, the programmer writes the program using these CPU instructions directly. If the program is written in a high-level language, the compiler translates the HLL lines into corresponding sequences of CPU instructions. What follows is not in any sense meant to be a comprehensive discussion of CPU architecture design. Rather, it is just enough detail (I hope!) to make the sections that follow understandable. There's lots of additional detail on the net and in datasheets.

In the simplest sense, a CPU is that part of the microcontroller that executes instructions. It does this in a series of steps:

• Fetch an instruction from the "next instruction" memory location pointer

• Execute that instruction

• Advance the "next instruction" pointer accordingly

Every computer program is just a repetitive execution of this sequence.

CPU Registers

Any CPU will have a set of onboard registers, which can be viewed as very fast memory locations. These registers will either be addressed automatically (Instruction **I** always operates on register **R**), or they will be addressed by a few bits in the instruction operation code (op code). A typical CPU will have the following types of registers:

Data Registers

These registers act as sources and destinations for arithmetic, logic and shifting instructions. Sometimes these registers are called accumulators. If the data register can also be used for general addressing functions, it becomes a "general-purpose" register, as described below.

Addressing Registers

These registers are used to hold addresses for memory data accessing. Depending on the CPU design, these registers may be full-fledged general purpose registers (the same register can be used for data manipulation or data addressing, or these registers may be limited and designed only for holding addresses used for data accesses.

Stack Pointer

The stack pointer is an address register which points to a section of memory that is used for the CPU hardware stack. The hardware stack is the stack that is used by the hardware for subroutine calls and returns, and for interrupt calls and returns. It is also possible for the user program to use the same stack, pointed to by the stack pointer, for saving and restoring other data. Or, the user program may use another stack, pointed to by another address register. The difference between such software stacks and the hardware stack is that only the hardware stack is invoked by the CPU instructions for subroutine calls and returns, by the interrupt mechanism, and in many cases by instructions for pushing and popping stack data.

Not all CPU designs use a hardware stack pointer. Some just reserve a single address register, often called a link register, which serves as a one-deep stack on the CPU itself, thus resulting in very quick subroutine calls and returns. If a deeper stack is needed (almost always the case at some point in a program), user or OS code must copy memory addresses between the link register and a software-maintained stack.

Program Counter (also known as Instruction Pointer)

This is the address register that points to the current (or next) instruction to be executed. It may be accessed by special instructions, or it may be a standard address regiser or even a full general-purpose register. It will automatically advance to point to the next instruction in a program, and will automatically be adjusted based on program jump, call and return instructions.

CPU Instruction types

A microcontroller CPU will execute a range of instructions, some of which manipulate data, some of which affect program control or CPU behavior, and some of which perform other actions. Here is a brief overview of the main types of CPU instructions. There will of course be variations in the instruction details, how the source operands are accessed, and where the result ends up. The following list is pretty minimal - different CPU architectures may execute a much richer set of instructions, but will always execute variants of these basics.

Data Manipulation Instructions

Arithemetic Instructions

Arithmetic instructions allow the CPU to add and subtract numbers, and in many cases to multiply and divide numbers. These numbers may be constants, fixed for all time as the program runs, or they may be variable, modifiable values held in data memory or registers. Signed numbers are just about universally represented in 2s complement form.

• **ADD**

    A=B+C

• **SUBTRACT**

    A=B-C

• **COMPARE**

    B-C

The instruction to compare two values is actually a subtraction instruction, but one in which only the flags are set, with no values being altered.

• **NEGATE**

    A= -B

Takes a signed number and converts it to the negative of that number. Equivalent to A=0-B.

It is important to understand that, in the great majority of cases, your CPU will only have instructions for such arithmetic operations up to its natural word size. So an 8-bit AVR will have an instruction to add 17+43 (both values fit into 8 bits, as does the result), but not 170,000,000+43,000,000. The AVR is perfectly capable of performing the latter addition, but it just requires a sequence of instructions, each working on individual 8-bit pieces of the numbers. If writing in ASM the programmer will have to explicitly write this sequence of instructions. If writing in C or another language, the compiler will automatically generate the instruction sequence (which you can see by looking at the ASM listing output of the compiled program).

It is also common for CPUs to have increment and decrement instructions, which are shorthand instructions for adding or subtracting 1 (or, in some cases, a small number up to +/- 7 or +/- 15). Since these operations are so common in programs, it is worthwhile for a CPU to support efficient instructions for them.

• **INC** and **DEC**

    A = A+1

    A = A-1

Logic Instructions

Like the arithmetic instructions, these instructions will typically work on data that is the natural word size of the CPU, or smaller, e.g. 8 bits on an AVR, and 32, 16 or 8 bits on an ARM Cortex M3. To perform logic operations on larger data requires a sequence of instructions just as is required for arithmetic operations.

Note that all the following examples show data in binary format (a 1 or 0 in each bit position), not decimal format!

• **COMPLEMENT (NOT)**

X = ~A (sets every 1 bit in A to 0, and every 0 bit in A to 1)

A=00110110

X=11001001

• **AND**

X = A & B

produces the bitwise AND of A and B, e.g.

A=00110110

B=01101100

X=00100100 : 1 in every bit position where both A and B are 1, 0 elsewhere

• **OR**

X = A | B

produces the bitwise OR of A and B, e.g.

A=00110110

B=01101100

X=01111110 : 1 in every bit position where either A and B are 1, 0 elsewhere

• **EXCLUSIVE OR**

X = A ^ B

produces the bitwise EXCLUSIVE OR of A and B, e.g.

A=00110110

B=01101100

X=01011010: 0 in every bit position where A and B are the same, 1 where they are different

Shifting Instructions

• **SHIFT LEFT/RIGHT**

Shifts each bit one position to the right or left. For left shifts the "empty" bit position (D0, least significant bit, LSB) will be loaded with 0. For right shifts the empty bit position (most siginifcant bit, MSB) will usually keep the value it had (sign extension) but there may be versions of the instruction that shift a 0 into the MSB. For both directions, the bit that is shifted out is usually shifted into the Carry bit, which provides the linkage for multi-stage shifts.

• **ROTATE LEFT/RIGHT**

The same as shift, except that the bit that is shifted out is shifted back into the empty bit position. It is also shifted into the Carry bit at the same time.

• **SHIFT LEFT/RIGHT THROUGH CARRY BIT**

The same as shift, except that the Carry bit is shifted into the empty bit position.

• **ROTATE LEFT/RIGHT THROUGH CARRY BIT**

The same as rotate, except that the Carry bit is included in the rotate, so the Carry bit value is rotated into the empty bit, and the bit shifted out is shifted into the Carry bit (but not into the empty bit position).

Program Flow Instructions: Unconditional Jumps

• **JUMP**

PC (program counter) = jump destination address. Execution continues at this new address.

Program Flow Instructions: Conditional Branching

• **BRANCH EQUAL / NOT EQUAL (TO ZERO)**

• **BRANCH CARRY / NO CARRY**

• **BRANCH POSITIVE / NEGATIVE**

if the condition is true, PC = branch destination address, otherwise PC = next instruction address

Program Flow Instructions: Subroutine Calls and Returns

• **CALL**

Save return address

PC = call destination address

• **RETURN**

PC = saved return address

Subroutines are one of the most basic forms of code reuse. If you have a piece of code that needs to be run at different times and/or with different data, a subroutine lets you isolate that code and run it ("call it") from any other section of code, at any time. In fact, a subroutine can even call itself, but that's getting into recursion, which is not something we'll be discussing.

The essence of a subroutine call instruction is that it is a jump (jumping to the subroutine code), but a jump that remembers where it came from. By remembering the address that it came from, at the end of the subroutine it can jump back to where it came from so the calling code can continue. To be more precise, a subroutine call instruction is a jump instruction which saves the address of the instruction following the subroutine call instruction, not the address of the subroutine call instruction itself. It is the next instruction that needs to be executed after the subroutine returns. If the subroutine returned to the subroutine call instruction, the program would find itself in an endless loop, calling the subroutine without end.

The call instruction needs to have a known location to save the PC value so the subroutine can return. Normally this known location can be one of two places. The PC value can be saved in memory on a stack, which is pointed to by a CPU register, or it can be saved directly in a register (typically called the "link" register). The advantage of the stack is that subroutines can call other subroutines, and the stack will just continue to grow to hold all the return addresses. The advantage of the link register is that storing and retrieving data from registers is faster than storing and retrieving from memory. The disadvantage of the link register, if you want to call it that, is that if a subroutine wants to call another subroutine it needs to explicitly save the link register contents on a stack. A subroutine that calls other subroutines is often called a branch subroutine, which a subroutine that does not call any other subroutines is called a leaf subroutine. Calls to and returns from leaf subroutines can be faster with a link register, at the expense of requiring branch subroutines to manually save and restore the link register. In any case, either method performs the necessary function of saving the PC value in a known location so that it can be retrieved and loaded back into the PC at the end of the subroutine.

To return from a subroutine, as noted, the PC must be loaded with the address that was saved when the subroutine was called. And as mentioned, this address could be in memory on a stack, or it could be in alink register. Many CPUs have a special instruction called a "return" instruction that fetches the address and sticks it into the PC. Some CPUs just use a standard move instruction to move the saved address into the PC. But the result is the same in any case - the program continues to execute where it left off.

What Next?

After all this background it's finally time to think about writing our first embedded program, one that blinks an LED. In deference to tradition I am calling this first program “Embedded Hello World”. That will be the subject of the next tutorial in this series. In the meantime you might want to look over the datasheet or user manual for your microcontroller to get a better understanding of the particular instruction set it executes.

Previous post by Mike Silva:
Introduction to Microcontrollers - Beginnings
Next post by Mike Silva:
Introduction to Microcontrollers - Hello World