Microcomputer Interfacing Chapter 2 - Programming

Mark Lipski

September 13, 2016

1 The Hardware and Architecture

Before I get into how to program an MCU, the generalized MCU architecture and assembly instructions should first be covered.

General core computer architecture components found in almost all MCUs important to programming

- ALU(Arithmetic Logic Unit) - Should be familiar, used for arithmetic operations such as addition, subtraction, and logical operations such as comparisons, and's, or's, etc.
- Barrel Shifter - Used for logical shift operations, will be covered later
- Multiplier/Divider - Used for multiplication operations
- FPU - Used to handle any floating point operations
- PSR(Program Status Register) - A register which contains data based on the state of the program, primarily useful for debugging programs.

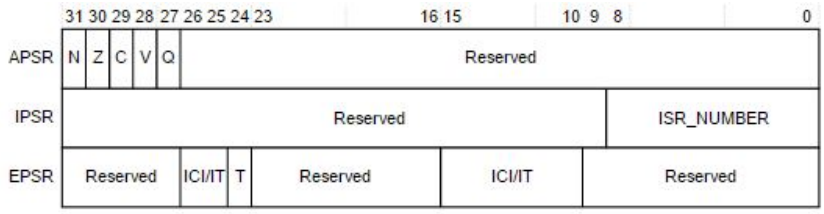

Figure 1: A diagram of the program status register for the Cortex M4 MCU, directly from ARMs website

2 Assembly Language

What is assembly? Assembly is essentially the lowest level programming language that exists. In assembly there aren't any variables, but rather you must manually control the flow of data between registers, the stack and memory.

Assembly is relatively important to understand, as it gives the programmer additional control and precision compared to C. However it should be noted that assembly has its place, and the vast majority of MCU programming will be performed in a higher level language like C. The reason for this is that most C compilers will be able to write cleaner, safer, more efficient assembly code than you can, while simultaneously being more comprehensive to both read and write.

Assembly can be used when you need explicit control over the content in registers or regions in memory, or if you need very explicit control over the timing of a block of code.

Programming in assembly involves writing instructions line by line, which are binary codes fed into the processor. Each instruction corresponds to some operation to be performed by the processor. These operations involve the hardware architecture components outlined earlier. For instance, an ADD assembly instruction might take the value from two registers, add them, and then store the result in another register.

Registers are an array of flip flops which have a set length, based upon the architecture of your processor. If your processor has 32 bit registers, a single register can store integers up to $2^{32} - 1$. Common register lengths are 8, 16, 24, 32 and 64 bit registers. The majority of modern MCUs require all data to flow through registers. What this means is that in order to utilize any of your processor's hardware components, you must pass data through the registers in your processor.

For instance, if you want to add two numbers in an arm processor, you must store the two numbers in two registers, then feed the value of both registers as inputs to the adder, and save the result in another register.

Some core assembly instructions for the ARM instruction set:

- MOV - Copies the data from one register into another register, or a literal value into a register
- ADD/SUB - Adds/subtracts the data in the two provided registers, then writes the result to the provided register
- CMP - Compares the values in the two provided status and updates the appropriate value in the PSR with the result

- AND/OR/XOR - Performs bitwise AND/OR/XOR on all the bits in both registers, comparing them, then writes the result to the specified register.
- LSR/LSL - Shits the bits in the register left and right, padding the added space with zeros
- ROR - Rotates the values in the register right by the provided binary value
- B - Adds the provided value to the Program Counter, causing a branch to the new address.
- LDR/STR - Loads or stores to the provided address.

For more details, the reference manual for your ARM processor should be consulted.

2.1 Example 1

MOV r0, #15 //Moves the integer value 15 into register 0
MOV r1, #3 //Moves the integer value 3 into register 1
ADD r0, r1, r2 //Adds the values stored in register 0 and 1 and stores the result in register 2
LDR r0, [**r2**] //Loads the value at the address of register 2, and stores it in register 0
LSR r0, #2 //Shifts the value loaded into r0 left by 2, effectively multiplies by 4
STR r0, [**r2**] //Stores the value stored in r0 at the location of r2

The equivalent C code might look something like
int num1 = 15;
int num2 = 3;
int *num3 = (int*)(num1 + num2);
***num3 = (*num3) ¡¡ 2;** //The logical shift operation in C

Obviously the C code itself isn't anything useful, and hence neither is the assembly, it's just meant to demonstrate how you might use registers to perform arithmetic and memory operations.

2.2 Register List

Most MCUs have a different number of general purpose registers, but a standard set of special purpose register.

- PC(Program Counter) - The program counter contains the value of the current line of code that is being run.

- LR(Link Register) - Used for storing the return address when branching and returning from functions.
- SP(Stack Pointer) - Stores the address of the current location of the top of the stack

2.3 Conditionals

Conditional and if statements can be implemented in assembly using codes which can get appended to the end of every instruction.

Table 3.4. Condition code suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned
CC or LO	C = 0	Lower, unsigned
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned
LS	C = 0 or Z = 1	Lower or same, unsigned
GE	N = V	Greater than or equal, signed
LT	N != V	Less than, signed
GT	Z = 0 and N = V	Greater than, signed
LE	Z = 1 and N != V	Less than or equal, signed
AL	Can have any value	Always. This is the default when no suffix is specified.

Figure 2: List of Condition Code Suffixes from the ARM website

Appending the condition codes to the end of an instruction ensures that the instruction will only be executed if the condition is met.

2.4 Example 2 - Branching and Conditionals

The C code might look something like:
char *string = 25031;

```
int i = 0;
int lim = 10;
while(i != lim) {
    printChar(string[i]);
    i++;
}
```

There are going to be a few pieces of assembly you might not understand but I'll explain after.

```
          MOV r4, #25031
          MOV r3, #0
          MOV r5, #10 //The end condition of the loop
loop      LDR r0, [r4]
          CMP r3, r5 //Compares r3 to r5
          BL printChar //Branches to printChar label
          ADD r3, #1 //Increments the loop by one
          ADD r4, #1 //Increments the address of the char array
          BNE loop //Branches if the NE flag is set in the PSR
```

There are two points which should stand out in the assembly code above.

The first of which is the loop label on the 4th line of the assembly code. The "loop" label signifies to the linker that any instances of "loop" in the code will reference the line of code where the label "loop" was declared. The program branches to that line, therefore creating a loop.

The second of which should be the **BL** instruction code. The only difference between this and a normal branch is that it writes the PC + 1 to the link register(LR). The incentive for this being that you can reference any function from any piece of assembly code, provided you provide a return address in the LR. Consider a scenario where you branch to a random C function. The C function has no record of where the Program Counter (PC) has been, and hence it has no way to return from the function call. However, since we used the BL function the address of the line we left in the assembly code was stored and the program will resume from there once

One thing worth mentioning is that I've assumed that the memory address format is that of a byte offset. So as the char data type is a byte long, I add 1 to increment the position in the array.

There are many textbooks dedicated to assembly code, so this is just designed to act as a very basic reference to get you started.

3 Assembly Execution Time

One of the benefits of using assembly code is that it gives you relatively explicit control over timings, excluding interrupts and certain nuances involved in branch prediction.

The Cycles per instruction and clock frequency of the computer can be used to determine how long a segment of assembly code will take to execute. The cycles per instruction for the Cortex M4 can be found here. You should be able to find the CPI by googling the name of your processor, followed by CPI.

```
loop    LDR r0, [r4] //LDR has a CPI of 1
        CMP r3, r5 //CMP has a CPI of 1
        LSR r0, 2 //LSR has a CPI of 1
        STR r0, [r4] //STR has a CPI of 1
        ADD r3, #1 //ADD has a CPI of 1
        ADD r4, #1 //ADD has a CPI of 1
        BNE loop //B has a CPI of 1, given that it will predict branch taken
```

In total the loop executes in 7 clock cycles, excluding the startup and end cases. Given a clock frequency of 300MHz, the loop will execute in 23.3nS.

4 Writing Low Level C

There are several differences involved in writing C effective C code for MCUs compared to standard C code. For instance, if the MCU being used doesn't have an operating system, concepts involving dynamic memory allocation simply don't exist. C for MCUs are going to require some important binary operators. This reference details all the different bitwise operators that will be required to control registers and peripherals in C.

It's also good to understand how to utilize different data sizes. Depending on your C compiler there are different data types that correspond to different register and data sizes.

If you need to perform an operation involving a byte, the char datatype can usually be used, as chars are 8 bits. If you need uint8_t can also be used, as well as uint16_t and uint32_t, provided your MCU goes up to 32 bit registers.