CS 273 Course Notes: Assembly Language Programming with the Atmel AVR Microcontroller

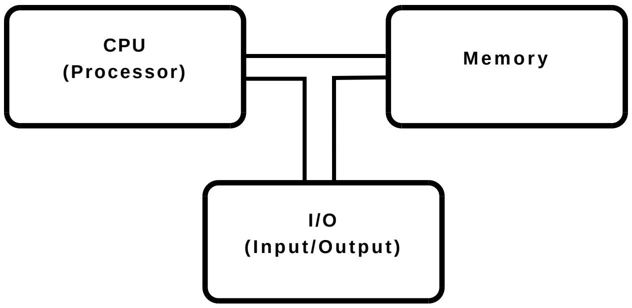
Contents

1 Introduction

1.1 High Level Programming Languages

So far your experience programming computers has been with high level programming languages like C, C++, and Java (and possibly other niche languages like Javascript, PHP, and Perl). But these languages are geared (mostly) towards how we think, not how the computer computes. They are independent of what type of computer you are using (e.g., PC, Mac, or even your phone) and so must be translated into how the particular computer you are using actually computes. This translation is done by a *compiler* or an *interpreter* (or even sometimes both, like with Java). In this course we are going to learn how computers really compute!

A computer has (essentially) three components:



- a **Processor**, also called a CPU (central processing unit); this performs computations that are specified by instructions (the program);
- some **Memory**, which stores both data and instructions (the program); and
- some **Input and Output** mechanisms, called I/O (such as keyboards, monitors, printers, disk drives, networking, etc.).

These components are connected by *busses*, which are collections of wires that transfer information (bits). Thus, a computer lets you get information in, perform some computation, and get results out. Getting results out is the whole point!

1.2 So How Does a CPU Work?

A CPU executes basic instructions, called **machine instructions**, or machine code. These are simple instructions like add, subtract, move (copy), and compare. Each machine instruction is executed directly by the hardware, which means that there is circuitry specifically for that operation. The instructions are, in essence, what the hardware understands. A compiler or interpreter must take the program you write in a high level language and translate it into the machine instructions of the CPU you are using. A compiler does this just once, producing an executable file that you can run over and over; an interpreter does this each time you run the program.

All of the machine instructions that a CPU understands are together called its **instruction set**. Thus, what a CPU does can be understood by looking at its whole instruction set; this understanding is captured in the term **ISA**, or *Instruction Set Architecture*. The ISA is everything you need to know about what the CPU does, without knowing *how* it does it (the internal design). The people who designed the CPU also design its ISA; so typically different CPUs from different companies (and even within a company) will have different ISA's. This means that compiled executable code for one ISA will not run on another ISA.

Wouldn't just one ISA for all CPUs make the world easier? Yes, it would, but when individuals in a company want to design a CPU, they can pick whatever instructions they want to (and they usually do!). Thus, they design their own ISA, different from everyone else's. This also means they are the only supplier of CPUs for their own ISA, so they can sell more if a lot of people start using their CPUs. But if the market is big enough, someone else might come along and design and build a CPU that copies the ISA of the successful CPU. In this case, the internals of the CPUs are different, but the ISA is the same! This is exactly what

AMD does with Intel's x86 ISA, and why you can buy a PC with a CPU from either company and run the same programs on it.

Another reason for different ISAs is that a CPU may be specialized for a certain task, like controlling a car engine, or doing graphics (yes, graphics cards have a CPU), and so will have different machine instructions.

1.3 The Fetch-Decode-Execute Loop

A CPU's life is pretty boring. Once powered up and initialized, it does the same thing over and over, as fast as it can, until it is powered down. Its actions are known as the **Fetch-Decode-Execute** loop (some people might just call it the Fetch-Execute loop). Over and over, the CPU does these steps:

1. Fetch: Read a memory location that contains a machine instruction.
2. Decode: Figure out which instruction it is that was just read.
3. Execute: Perform the computation specified by the instruction.

Essentially, programs are executed one instruction at a time, and so the CPU simply does this over and over as quickly as possible. Even though these sound like simple steps, things can get a bit tricky. Some instructions may need to access memory for data values, or perform I/O. Some instructions may cause the program to take a different execution path. These and other issues make the actual performing of the Fetch-Decode-Execute loop sometimes a bit tricky, but this is basically all a CPU does!

2 Numbers and Numbering Systems

Probably by now you have heard the terms **binary, bit**, and **byte** somewhere already, and maybe you already know exactly what they mean and why they are used. But read on anyways!

"Bi" means two, and binary is a two-valued (or base-2) number system, with only the digits 0 and 1. Why is binary important? Well, in short, because computers operate in binary. Computers are electrical machines; everything we do with them must end up as electrical signals. Devices such as toasters, even (older) TVs and radios, are **analog** devices, which means that their electrical signals vary continuously, but a computer is a **digital** electrical device, which means it operates on electrical values that are kept at discrete levels. Furthermore, all computers operate on just two discrete levels of electricity: these two levels are interpreted as the binary digits 0 and 1. Thus, in a computer, all values are in binary, and all operations (including addition, multiplication, etc.) are performed in binary. That's why binary is so important to us.

Each binary digit is called a **bit**; an eight-bit binary value is called a **byte**, and a group of bytes is called a **word**. Most CPU's these days call a 32-bit value a word, but not the CPU we will be programming in this course! It is mostly byte-oriented, but it occassionally uses two bytes together, and so it calls 16 bits a word.

2.1 Decimal and Positional Notation

In understanding and using the low-level operations of computers and their components, our normal usage of numbers can be quite cumbersome, so we need to introduce some alternatives. Our normal use of numbers is always in the **decimal**, or base-10, numbering system; the digits of 0 to 9 represent the values directly representable in base-10 (in any base, the digits are always from zero to one minus the value of the base). To represent values larger than 9, our numbering system uses **positional notation**. In this scheme, a digit can represent a value larger than its direct value. Thus

$$329 = 300 + 20 + 9$$

The digits 3 and 2 do not represent three and two, but 300 and 20. This is because the position of the digit indicates a **power of 10** by which the digit should be multiplied. This is represented as the formula

$$329 = (3 * 100) + (2 * 10) + (9 * 1) = (3 * 10^2) + (2 * 10^1) + (9 * 10^0)$$

The positional scheme is much better than, say, the Roman Numeral scheme, where each symbol represented a unique value no matter where in the number it existed (i.e., M=1000, D=500, C=100, L=50, X=10, V=5, I=1). For more information you can check out (Wikipedia:Positional_notation).

Positional numbering is great, and we will keep using it, but it is the decimal (base 10) part that gives us problems with computers. We need some other bases! The general scheme for a positional numbering system in a base B is:

- there are B unique symbols, representing the values 0 to $B - 1$;
- digits in a number are multiplied by a power of B, starting at 0 on the right and increasing by 1 for each digit to the left.

We can use this scheme for any base we can imagine, but there are only a few that are important to us, practically speaking. *Notice that* if we use the symbols '0' and '1' to represent zero and one in any base B, the value of B is always "10" in its own base-B numbering system!

2.2 Binary, Hex, and Octal, Oh My!

The number systems we will use in this class are:

- Base-10, or **decimal**: digits 0-9, and each place is a power of 10;
- Base-8, or **octal**: only digits 0-7, and each place is a power of 8;
- Base-16, or **hexadecimal**: digits 0-9,A-F, and each place is a power of 16; and
- Base-2, or **binary**: only digits 0 and 1, and each place is a power of 2.

We call binary digits "**bits**", and often say "hex" rather than the full "hexadecimal". For more information see (Wikipedia:Hexadecimal, Wikipedia:Octal, and Wikipedia:Binary). Visually, octal and binary are "easy" since the digits actually look like numbers to us; with hexadecimal, however, we need 16 digit symbols and our brains are only used to recognzing 10 (the symbols 0-9). In hex we also use the letters A-F (or a-f, case doesn't matter) as symbols for the digits representing the values ten through fifteen, and we just have to get used to thinking of them as numerical digits!

Because we use the same 0-9 digits in different numbering systems, the number "42" is a valid number in any of the octal, decimal, and hex number systems (and is a different number in each!). So we need some way of knowing which number system we are using. In a math course we would use a subscript to show the base, such as:

$$1011_2 = 11_{10}$$

Notice that both values above only use the digits 0 or 1, but they represent the same value (eleven) with different sequences of those digits, because of the different base. Subscripts are great, but in a computer science course where we write programs in simple text files that cannot represent subscripts, we will need to use special symbols to indicate which base we are working in. The AVR tools use the notation that is used in Unix, C, C++, Java, and many other places:

- A hexadecimal number has a leading "0x" attached to the number (e.g., 0x4f2a is the hexadecimal number for the decimal value 20266);
- An octal number has just a leading "0" attached to it, and then the number itself (e.g., 076 is decimal 62, and 089 is not a number!); and
- Sometimes (not often), a binary number has "0b" in front of it, and then the number (e.g., 0b10110 is decimal 22);

Usually in Unix/C/C++/Java you have to convert a binary number to something else (like hex), then use it in your source code. The Arduino environment allows you to use "B" as a prefix in your C/C++ code for binary numbers, and in your assembly files you can use "0b" as a prefix for binary numbers.

2.3 Converting Values between Number Systems

Computer Scientists use octal and hexadecimal a lot because these systems easily convert to binary, whereas decimal does not. The reason for this is that the bases 8 and 16 are exact powers of 2, whereas 10 is not. Because 8 is 2^3, each octal digit is exactly 3 binary digits; because 16 is 2^4, each hex digit is exactly 4 binary digits. For decimal there is no exact mapping, so you have to do some arithmetic to convert to binary.

For example, octal 046 is binary 100110 – the leftmost three bits (100) is the octal digit 4, and the rightmost three (110) is the 6; octal 036 is binary 011110; the leftmost three bits change, but the last three do not, since only the leftmost octal digit changed. This is *very nice* because when doing conversion we only need to think about each digit by itself, we do not need to be concerned about the overall value. The table below shows all of the binary-hex-octal digit mappings for all 4-bit values, zero to fifteen:

Binary	Hex/Octal/Decimal		Binary	Hex	Octal	Decimal
0000	0		1000	8	10	8
0001	1		1001	9	11	9
0010	2		1010	A	12	10
0011	3		1011	B	13	11
0100	4		1100	C	14	12
0101	5		1101	D	15	13
0110	6		1110	E	16	14
0111	7		1111	F	17	15

Notice that one octal digit exactly captures all unique 3-bit values, but when the fourth bit rolls over to one, so does the octal value; so one octal digit represents exactly 3 bits. The same is true for hexadecimal, but with 4 bits. However, decimal notation rolls over from 9 to 10 part way down the second column; this makes it much harder to convert between decimal and binary, which is why we like to use octal and hex!

To convert between hex/octal/binary do the following simple things:

- From hex to binary, write the four bits that are equivalent to the hex digit, in exactly the same position in the number as the hex digit. Always write four bits!
- From octal to binary, write the three bits that are equivalent to the octal digit, in exactly the same position in the number as the octal digit. Always write three bits!
- From binary to hex, group the bits into fours, starting at the right. Then translate each group into one hex digit, in the same position as the group of four bits.
- From binary to octal, group the bits into threes, starting at the right. Then translate each group into one octal digit, in the same position as the group of three bits.
- If on the leftmost side of the binary number there are not enough bits to make a complete group, it is OK to add 0's to the left.

Unfortunately, we think in decimal, so we often need to convert numbers to and from decimal. This takes a little more work. Converting **to** decimal is very easy: just multiply each digit by the power of the base for that position. Examples:

- Octal: $0473 = (4 * 8^2) + (7 * 8^1) + (3 * 8^0) = (4 * 64) + (7 * 8) + (3 * 1) = 315_{10}$
- Binary: $0b1011 = (1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) = (1 * 8) + (0 * 4) + (1 * 2) + (1 * 1) = 11_{10}$
- Hex: $0xC6A = (12 * 16^2) + (6 * 16^1) + (10 * 16^0) = (12 * 256) + (6 * 16) + (10 * 1) = 3178_{10}$

Notice that binary to decimal is *especially easy*, since anything multiplied by 1 is itself and anything multiplied by 0 is 0. This means we can drop all the terms with a 0 bit, and just use the power of 2 directly for each 1 bit. Thus the example above becomes

$$0b1011 = 2^3 + 2^1 + 2^0 = 8 + 2 + 1 = 11_{10}$$

Converting **from** decimal to another base is a bit harder: we must repeatedly divide by the base we are converting to. The remainders from these divisions will be our digits in the converted number (but they are backwards!) The algorithm is:

1. Divide number by base, get quotient and remainder. This remainder is the rightmost digit.
2. Now divide qotient by base, get new quotient and remainder. This remainder is second-rightmost digit.
3. Keep dividing each new quotient by the base to get the next rightmost digit. *Stop when new quotient is 0 and remainder is the old quotient; this is the final, leftmost digit of the number in the new base.* (shortcut: when you see a new quotient that is less than the base, this is your final leftmost digit)

Octal example: convert decimal 93 into octal. 93/8 is 11, remainder of 5. We then take the quotient and again divide by 8. So 11/8 is 1, remainder of 3. We do the same again, so 1/8 is 0, remainder of 1. We stop since we have a quotient of 0. Now we take those remainders (5,3,1) and reverse them for our octal number of 0135, which is the conversion of decimal 93 into octal. We can check that by converting back: $1 * 8^2 + 3 * 8 + 5 = 64 + 24 + 5 = 93_{10}$

Binary example: convert decimal 21 to binary. 21/2 is 10, remainder 1; 10/2 is 5 remainder 0; 5/2 is 2 remainder 1; 2/2 is 1 remainder 0; 1/2 is 0 remainder 1; done; Reverse remainders and get 0b10101. Check: $16 + 4 + 1 = 21_{10}$.

2.4 Negative Numbers: Two's complement representation

So far, we've just used positive numbers. If you have tried the last part of lab one yet, you have seen that the result is a negative number. Does the AVR CPU know this? How do we know this? In lab 1 it is easy because we can do the arithmetic by hand and see that a negative answer is correct, but if we don't know ahead of time, how do we know when we are programming?

In actual writing, we use a minus sign to indicate a negative number. But the CPU and memory just have 1's and 0's, that is all, so how do we determine a negative number? We have to somehow encode the fact that a value is negative into the value itself!

Firstly, let's recall that all of the values in a computer are of some fixed length of bits. In an AVR CPU, most are 8 bits, and some are 16 bits. But we always know this. This is important: *values are always a fixed number of bits.*

Given a fixed number of bits, an ingenious representation of signed numbers is the **2's complement** representation (abbreviated as 2C). The 2C representation is defined by the *operation to change the sign of a value*:

1. first, complement (flip to opposite) all the bits;
2. then add one

The first step is known as *1's complement*; the second step makes it 2's complement. For example, decimal -11 is found by:

1. converting positive decimal 11 to binary, which is 0b00001011 (must be 8 bits)
2. complementing it, which gives us 0b11110100 (each bit is opposite)
3. then adding 1, which gives us 0b11110101
4. so, decimal -11 is 8-bit binary 0b11110101 (hex 0xf5)

In 2C the upper bit acts as a sign indicator: a leftmost 1 means negative, and a leftmost 0 means positive. The **great thing** about 2C is that arithmetic can be done completely ignoring whether a number is signed or not! This means (**important**) that the CPU doesn't care if you are using signed numbers or unsigned numbers!

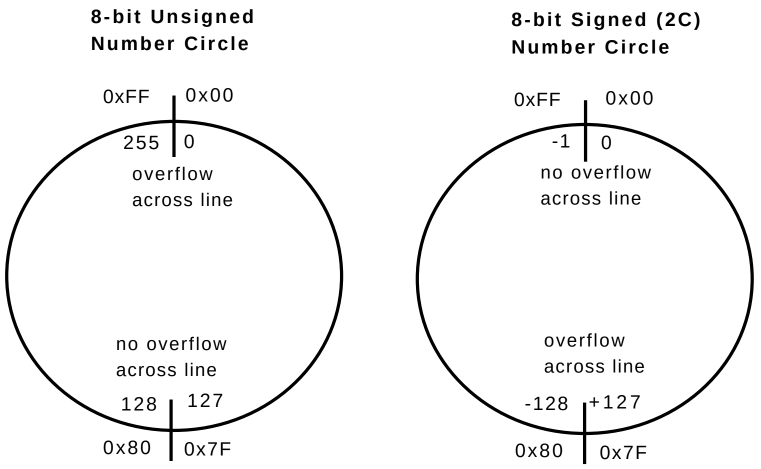
Some historical computers used 1's complement (just complementing all the bits) to represent negative numbers, but the wierd thing about 1C is that it has two values for zero – a positive zero (00000000) and a negative zero (11111111). Today all computers use 2C.

Important: We now have TWO number systems that we can use in a computer: an *unsigned* number system which uses values from 0 on up, and a *signed* number system that uses both positive and negative values (and 0). You can write programs that use either (or both if you are careful). In C/C++ programming you also have access to both: when you declare a variable as an "int", you are using the signed number system, and when you declare it as "unsigned int" you are using the unsigned number system (this is also true for other sizes of int, such as long, short, char, etc.). In Java there is no data type of "unsigned int", so all you can use are signed values.

Most arithmetic-type instructions do not care whether your values are signed or unsigned – the place where it makes a difference is in your compare-and-branch instructions, and in certain bit manipulations such as shifts.

2.5 Number Circles, Not Number Lines!

In a mathematics course we would talk about an *infinite* number line, but since computers are finite devices, we must think about our number systems as finite lines, not infinite lines. Even worse, due to the nature of the arithmetic operations as implemented in electronic circuitry, these finite lines form *closed circles*! The size of the circle is determined by the number of bits we are using: below are the number circles for 8-bit unsigned and signed number systems.



As with number lines, addition moves you "right" (clockwise) and subtraction moves you "left" (counter-clockwise). The top of the circle is where the bit patterns of all 0's and all 1's meet, along with the hexadecimal representation and their decimal meaning in the two number systems. The bottom of the circle is also marked with the hexadecimal representation of the bit patterns, and their decimal meanings in the two number systems.

For the unsigned number system, the top of the circle is where the ends of the finite number line meet, and for signed (two's complement), the bottom is where the ends meet. The important lesson is that *mathematical operations will freely cross those points*! We call this an **overflow** and it will play an important role later on, but you should realize that you can indeed perform an operation in your code and end up with an incorrect result if your operation crosses these lines.

3 Computer Memory, Load/Store Operations, and Assembly Language

In this section we delve a little deeper into how a CPU executes instructions, and then how you as a programmer writes those instructions.

3.1 Memory

You probably have at least one computer, and you probably know how much memory it has (maybe 4GB of RAM). But what does that mean, and what is it used for?

From the section above we learned that all values in a computer are in binary: just lots and lots of 1's and 0's. This means that the data looks like "0110001010100100000111110010111" And the machine instructions look like "0110001110101010111100" as well! Recall that the memory holds both instructions and data. So, the memory holds a bunch of bits (actually lots and lots of bits!) Your 4GB of RAM holds more than 32 *billion* bits!

Memory must be organized somehow, in order for us to find the particular data or instruction that we need. The best way to think about memory is to think of it as one huge 1-dimensional array, with each location storing a certain number of bits (usually 8, or one byte). In a program we would use an index to access an array element, but in memory we instead call it an address.

An **address** is a number that signifies a specific memory location, just like your apartment number signifies a specific apartment. Generally, each memory location holds one byte (8 bits) of data or instruction; this is true on your PC. On the AVR (the CPU we use in this course) the data memory has one byte per address, but the program (instruction memory) has two bytes (one word) per address.

Accessing memory is just like indexing an array in a program:

- an address (i.e., index) is presented to the memory;
- the memory looks up that address and returns the bits that are stored at that location.

This describes *reading* a value from memory; to *write* (or store) a value into memory, both the address and data are presented to the memory, along with a flag indicating a write operation, and the memory stores the new value in the location at that address.

3.2 General Concept of Computer Operation, with Memory and Registers

We saw earlier that a CPU's basic job was the Fetch-Decode-Execute loop: fetch an instruction from memory, decode it, and execute it, then do it all over again. For each instruction the computer executes, it must get that instruction from the memory. But instructions typically use data: for example, an add instruction uses two pieces of data (the two numbers to add) and produces one piece of data (the sum of the two numbers).

Where does the data come from (or go to)? It could come from memory, but accessing memory *four* times to perform an add (read the instruction, read two pieces of data, and write the result back) would be *extremely slow*, in computer speed terms.

So, all CPU's have some very special internal memory, called **registers**. These registers are fast, and these are typically what instructions like *add* use. For example, the AVR has 32 1-byte registers, named **r0** to **r31**. *Think of registers as predefined variables that you* ***have to use*** *in order to perform certain operations.* Most computational instructions only access registers, instead of allowing general memory access.

But if instructions like add can only use values that are in registers, we now need special instructions to get values from memory to the registers, and back, so that our values are in the registers when instructions like add need them. These special instructions are **load** and **store** instructions: load a register with a value from memory, store a value from a register back into memory.

Suppose we have the C or Java program statement "z = x+y;". What instructions does the CPU have to do to complete this single statement? Assuming that each of the three variables is stored in its own memory location, we need to:

- load the memory value of x into a register;
- load the memory value of y into a register;
- add the two registers together, with the result stored back into some register;
- store the result in the register back into memory at the variable z.

So, four machine instructions are needed to implement one simple C/Java statement. This style of *Load-Compute-Store* operation is very common, and ideally we want to make the "compute" part as long as possible, so that we avoid using memory too much. On the AVR we have a fairly large amount of registers to use, and so it is often possible to fit an entire iterative loop of operations inside the "compute" step, without loading from or storing to memory except before the loop begins and after it has completed its last iteration.

We now know that information in memory is retrieved by using addresses to indicate the location. The compiler typically picks where our variables are stored in memory, but the CPU must also be able to fetch the instructions from the program, and each time it finishes an instruction it must fetch another one at a different memory location. How does the CPU know what address to use when fetching an instruction? Well, the CPU has a very special register called the **Program Counter**, or just **PC**. The PC always contains the address of the *next* instruction to be executed; when the CPU goes to fetch the next instruction, it uses the value in the PC as the address to read the program memory at. As each instruction executes, the PC automatically gets changed so that the next instruction that is needed is fetched.

3.3 Assembly Language

Now we know about instructions, addresses, and registers. Let's get to specifics of programming. The machine instructions look like "01101010" – they are just a sequence of bits

- E.g., the two bytes "00001111-00000001" mean "add registers r16 and r17 and put the result back in r16" on the AVR CPU.

But we don't want to program using 1s and 0s, do we? That would be terrible! Actually, that is how computers were first programmed. The programmer entered the machine instruction using switches that were set to 1 or 0. *This was awfully tedious!* Instead we use words and abbreviations for each instruction, and then use a program to **translate** the words into the actual binary machine instructions.

- The word "**add**" signifies the AVR machine instruction 000011XX-XXXXXXXX, and is short for "Add Registers" (the X bits are filled in with the register numbers).

This "language" of words and abbreviations for the machine instructions is called **assembly language**, and the **assembler** is the program which reads a program written in assembly language, translates it and produces the binary machine code. This process is similar to compiling a program in a high level programming language, but we call it assembling to distinguish it from compiling because:

- each assembly instruction is exactly one machine instruction, whereas high level programming language statements are translated into many instructions each; and
- the assembly instructions are specific to a type of microprocessor (CPU) (e.g., an AVR assembly program cannot be assembled to run on a Pentium).

Even though assembling is not the same as compiling, but the **gcc** compiler, which we are using, also includes an assembler, so we can use it to assemble our programs. In the Arduino environment we will combine our assembly program with C/C++ code as well, so using gcc on all of the pieces makes sense.

An assembly language also has other nice features:

- It lets us use decimal, octal, and hex numbers (in addition to binary);
- It lets us assign names, or symbols, to certain values to make our program more readable;
- It lets us put comments in our code (This is important, really!).

(continued on next page)

3.4 An example AVR assembly language function w/ global data (not a complete program)

```
#
# Global data (val1 and val2) (new change)
#
    .data
    .comm val1,1
    .comm val2,1

#
# Program code (compute function)
#
    .text
    .global compute
compute:
    lds  r18, val2
    ldi  r19, 23
    add  r18, r19
    sts  val1, r18
    ret
```

An assembly language contains:

- **Mnemonic** names for the machine instructions (in the above program, these are *lds, ldi, add, sts, ret*);
- Register names: *r18, r19* in the program above
- Assembler directives for creating data and specifying addresses: *.data, .comm, .global, .text*
- Symbols that are names for values, such as data values or addresses: *val1, val2, compute* in the program above
- Syntax for specifying addressing modes (we'll get to this)
- Numerical values: *23*
- Comments: *lines beginning with '#'*
- NOTE: the assembly language does not contain anything that the microprocessor does not directly support. It is **not** a high-level programming language!

The assembler program (think of it as the compiler for the assembly language) then takes the textual program that you write (such as the code above), and does the following:

1. it use the instruction names and operands to select the correct machine instructions (**opcodes**);
2. it translates the symbols into numerical values;
3. it generates a binary file that contains the machine code that can execute on the actual CPU.

For the code above, since it is only one function and not a complete program, the assembler can only generate the machine code that represents that function, it cannot generate a complete, executable program. To do this some other code must be combined with the function; this process of combining separate pieces is called **linking**. When we assemble or compile a piece of a program into machine code, we call the result **object code**: in Linux/Unix the object code file has a ".o" extension, while in Windows an object code file has a ".obj" extension. When we link multiple object code files together to create an executable file, we call the result an **executable file**: in Linux/Unix, executable files typically have no extension at all, while under Windows they end with a ".exe" extension.

The Arduino environment requires a C/C++ file that it calls "PDE" code to contain at least the functions *setup()* and *loop()*; we use these as jumping off points to call our assembly language functions, and the Arduino environment compiles and links the object code pieces, along with object code from libraries, into a complete executable file. It also produces a version of the executable file that can be downloaded to the Arduino board, which is where we really need to execute it.

3.5 An assembler listing for the above program

The listing below is produced by running "**avr-as -a=compute.lst compute.s**", which assembles the file "compute.s" and puts the listing in "compute.lst".

```
GAS LISTING compute.s  page 1


   1
   2                #
   3                # Global data (val1 and val2) (new change)
   4                #
   5                    .data
   6                    .comm val1,1
   7                    .global val1
   8                    .comm val2,1
   9                    .global val2
  10
  11                #
  12                # Program code (compute function)
  13                #
  14                    .text
  15                    .global compute
  16                compute:
  17 0000 2091 0000      lds  r18, val2
  18 0004 37E1           ldi  r19, 23
  19 0006 230F           add  r18, r19
  20 0008 2093 0000      sts  val1, r18
  21 000c 0895           ret
  22

GAS LISTING compute.s  page 2


DEFINED SYMBOLS
                            *COM*:00000001 val1
                            *COM*:00000001 val2
          compute.s:16     .text:00000000 compute

NO UNDEFINED SYMBOLS
```

This listing shows our program and the resulting machine code and the memory that it takes up. The first column is the line number of the listing (and of the program). The second column is a *relative* memory address in hexadecimal. These relative addresses begin at 0, but that does *not* mean that when the program runs it will be at memory address 0! Following the relative address is the machine code that each instruction results in. Most are 16-bit machine instructions, but some are 32-bit. Finally, the program text line is displayed. The second page of the listing shows a table of defined symbols, and then a table of undefined symbols if there are any (in this case, none).

Important: each of these symbols represent *a memory address*, but we *don't know yet* what that address is! The two instructions that refer to "val1" and "val2" have a second 16-bit value of 0, but this is *temporary*: that value will be changed to be equal to the memory address represented by that symbol, when **the linker** create a final executable program. What's a linker? It is the final step that a compiler or assembler uses to create an executable program. It's job is to connect all the pieces together ("link"), and figure out the actual addresses used in the pieces, and then put those real addresses into the code whereever they are needed.

3.6 Creating and Using Functions with Proper Register Usage

In the Arduino environment, we will write most of our program content in assembly language as individual functions that will be called from the Arduino PDE code (which is C/C++ code), or from our assembly code itself. We will also need to interact with some of the Arduino library functions that are available.

Since all of the program must use and share the same 32 registers (*r0* to *r31*), there must be some agreed upon usage conventions for the registers. As an example, suppose in your assembly code you are using *r18* for some purpose and you need to call an Arduino library function. Should you expect that *r18* still has your value in it when the function returns, or should you expect that the code in the function might have used *r18* for some other purpose, and now it has garbage in it?

The good news is that the *gcc* compiler has already picked a usage convention for us, and all we need to do is obey it in order for our assembly code to "play nicely" with all the code it is linked together with.

The conventions are:

- Registers r18–r27 and r30–r31 are freely available to use in functions; the caller cannot expect their values to remain unchanged across the call; Note that r27:r26 is the X indirect addressing register, and r31:r30 is the Z indirect addressing register.
- Registers r2–r17 and r28–r29 (Y) can be used but *only if* they are saved first, and restored to their original value after use; the caller expects that their values remain unchanged across the call. The Y register is used as a frame pointer register by C functions (and by us).
- Registers r0 and r1 are never used for local data by the compiler, but may be used for temporary purposes; r0 is like a freely available register and can be used for temporary values (not preserved across function calls); r1 is assumed to always hold the value 0; if r1 is used for a temporary value (not across function calls) it must be cleared when the value is no longer needed.

In this course, ***ALL*** assembly code that you write must adhere to these conventions. You will be tempted to cut corners and skip these rules sometimes, but do not do it! It will save you countless hours of debugging time if you strictly adhere to these rules.

Typically, your assembly code should use registers r18–r25 for "regular" code that is performing computations. If you need to call some external function (such as a library delay routine), then you can either save your values into memory somehow (more on this in a bit), or you can first save the current value in a register in r2–r17, and then use it for your own purposes, knowing that the delay function cannot corrupt it.

Whether you are saving your caller's values in r2–r17 or r28–r29, or you are saving your own values in r18–r27 or r30–r31 in order to perform a function call, the easiest way to save register values is by using the **push** and **pop** instructions.

Later in the course we will learn about the detailed operation of the **stack**, but for now we can just say that there is a special portion of memory that the AVR treats as a *last-in-first-out* data structure called a stack, and we can place values on (or into) the stack using the *push* instruction, and retrieve them with a *pop* instruction. Think of a stack of dinner plates in one of those spring-loaded plate columns at restaurant salad bars; each dinner plate is a value, and you can only put a plate on the top (push) or take a plate off the top (pop). This means that *whatever you push on the stack must be popped off the stack in exactly the reverse order*; and because the stack is also used for other purposes, *you must* ***exactly*** *undo everything you do to the stack* (i.e., every push must have a corresponding pop).

3.7 Function Parameters (Arguments)

We will initially create functions with no parameters, but eventually we need to create (and also use other) functions that have parameters. In computer science we usually use the word *parameter* to refer to the named parameter variable that we use when writing our function's code, and the word *argument* to refer to something we use when we actually call the function. Worse yet, some textbooks will call parameters *formal* *parameters* and arguments *actual parameters*! I will try to use parameter and argument appropriately, but will probably slip up occasionally.

Later in the course we will learn the full set of rules for function parameters, but similarly to the register usage conventions in the last section, we will use the function parameter conventions that the C compiler uses, and for now the simplified rule set is:

- Arguments are passed in registers r25 down to r8, starting with r25.
- BUT, all arguments are aligned to start in an even-numbered register; this means that 1-byte arguments will only occupy even numbered registers and will skip odd-numbered registers; e.g., for a function with three 1-byte parameters, the first (leftmost) will be in r24, the second in r22, and the third in r20.
- Arguments occupy as many registers as needed, but are still little endian; e.g., if parameter 1 is 16 bits, the argument value is in r25:r24, and if it is 32 bits, the argument value is in r25:r24:r23:r22.
- A function return value is passed in r25 down to r18, depending on the size of the return value (a 64-bit return value would be in r25–r18), but as with parameters, a 1-byte return value is placed in r24, not r25; also, a 1-byte return value *must* be zero or sign-extended through r25, so r25 must be either all 0 bits (for an unsigned or positive value in r24) or all 1 bits (for a negative value in r24).

Most of our work in this course will deal with 1-byte arguments, and these are always in even-numbered registers starting at r24 and decreasing. We will sometimes use 2-byte arguments, and the Arduino library *delay()* function takes a 32-bit (4-byte) argument; we just need to use the registers appropriately.

4 The Atmel AVR Microcontroller

The title of this section calls the AVR a **microcontroller**. Why isn't it just called a CPU (or processor)? Well, it has the processor, the memory, and the I/O capability all embedded onto a single chip. This makes it a cheap solution for controlling fairly small systems, like microwaves or other such things. Despite the CPU, memory, and I/O being integrated, the abstract model of a processor communicating with memory and I/O still applies! Let's talk specifics:

- the AVR is an 8-bit processor, so most instructions perform their operation using 1-byte values, although a few instructions use 2-byte (16 bit) values for specific purposes;
- the AVR uses 16 bit addresses, so any time we need to think about, use, or construct an address to access memory with, we must deal with 16 bits;
- the AVR has a general-purpose register set of 32 8-bit registers, named r0 to r31, but not all instructions use all the registers the same, and there are usage conventions that we must obey;
- the upper 6 registers (r26–r31) can be used as 3 16-bit registers: X, Y, and Z; these hold changeable addresses for us, such as array element locations;
- the AVR has a 16-bit stack pointer, **SP**, which holds the address of where the stack is in memory; oddly enough, the SP sits outside of the typical CPU registers, and must be accessed using I/O operations! (at I/O addresses 0x3d and 0x3e (memory addresses 0x5d, 0x5e)) What's a stack? We'll learn in a bit;
- the AVR has a 16-bit program counter, **PC**, which as said before always contains the address of the next instruction to execute; thus the PC keeps track of where the program is executing;
- the AVR has an 8-bit status flags register, **SREG**, which contains 8 individual 1-bit flags (true/false conditions) that indicate any important conditions that resulted from the previous instruction(s); like the stack pointer, SREG is accessed in I/O space, at I/O address 0x3f (memory address 0x5f).

All of this information is part of the AVR's *ISA (Instruction Set Architecture)*, because it is necessary for understanding what its set of instructions actually does.

4.1 Time

We will need to understand how long our program, or a piece of our program, takes to execute. This is important for such things as, say, turning the robot for 1/2 of a circle. We would take the speed of the motor and figure out how long to keep the motor on, then write a program to handle that.

In a digital **synchronous** system, which is what a CPU is, time is strictly controlled by a **clock** signal. During each clock **cycle**, something happens. On the AVR, most instructions take 1 or 2 clock cycles, with a few taking 3 or 4 cycles. If you recall the fetch/decode/execute explanation of program execution, it is **hard to imagine** that those three steps can all happen in one cycle!

Well, they usually don't!

The AVR uses the idea of *pipelining*, which simply means the overlapping of the execution of sequential instructions. A good analogy is washing clothes: you don't wait until your first load is finished drying to start your second load in the washer, you overlap the washing of the second load with the drying of the first load. That's pipelining!

Modern complex processors like the ones in your personal computer use very complex pipelining schemes, but the AVR uses a simple one: the *fetch* of the next instruction is overlapped with the *execution* of the current instruction. This allows most of the simple instructions to take only one cycle of execution time, even though their total fetch+execute time is two cycles.

Branches cause problems with pipelining because the processor doesn't know what instruction to fetch next until the branch decides which way to go! The AVR does a small bit of **speculation**: it goes ahead and fetches the next instruction in sequence while the branch instruction is executing; if the branch falls through then the instruction fetched is ready to execute. If the branch needs to be taken, then the instruction that was fetched is thrown away and the instruction at the branch target is fetched. This is why in the instruction table branches are shown as taking either 1 or 2 cycles: they take 1 if they fall through, but take 2 if they actually branch.

4.2 Endianess

Although the AVR uses mostly 1-byte data values, it does sometimes operate on *multi-byte* values, with addresses being 2-byte values and all machine instructions being 2-byte or 4-byte values. With multi-byte values, bytes are stored in order, but the important question is: is the first byte the least significant byte or the most significant byte? All CPUs must choose a **byte ordering** when dealing with multi-byte values. The AVR chooses to store the lowermost byte first, and we call this choice **little endian**.

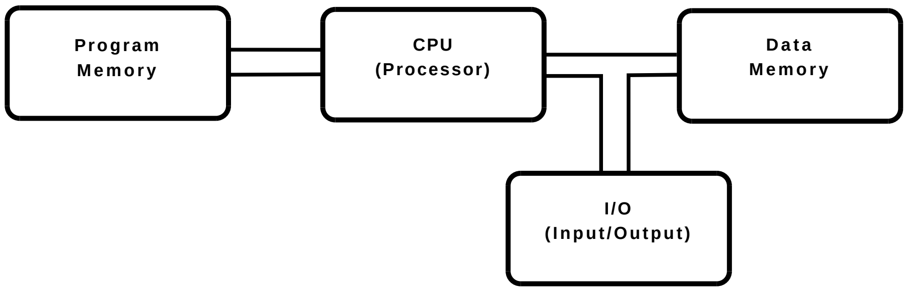
Other CPUs choose to store the highest byte first, and we call this **big endian**. Highest byte first seems natural in one way, because if we look at memory as increasing in address from left to right, the number reads in the same direction we would write it down. But what if you had a 16-bit result where you knew the upper byte was zero, and you just wanted to access the lower byte. This actually happens alot in C programming, because many of the C library functions return an integer, but actually the result is a single-byte character. In this case, then, you need to access the memory location at 1 plus the base address of the value on a big endian CPU, but on a little endian CPU the value is directly at the address.

In computer science, we call this choice *endianess.* If the most significant byte is first, then that CPU is "big-endian", and if the least significant byte is first, then the CPU is "little-endian". Our AVR processor is little endian.

Intel/AMD CPU's (and thus the resulting computers) are little endian, while most Motorola CPU's are big endian. If you sent a 16 or 32 bit number from a Intel-based computer to one with a Motorola CPU without modification, they would not agree on what that number is!!!

4.3 The AVR Memory: Separate Program and Data Memory

The AVR uses 16-bit addresses, and since $2^{16} = 65,536 = 64K$ ($K = 1024$), it can potentially address 64Kbytes of memory. The AVR is a **Harvard** architecture, shown below,



which simply means that the memory for data and the memory for the program are **separate**, and the address spaces are separate. Furthermore, each program memory address in the AVR is a *word address*, meaning each location holds 16 bits (two bytes), while the data memory is *byte addressed*, meaning each location holds just one byte. This means the AVR can have up to 128KB of program memory and 64KB of data memory. Our particular version (the avr328p) has 32KB of flash memory for storing programs, but only 2KB of data memory (SRAM). The program memory is not accessible by the memory load/store instructions, only the data memory is. The program memory is only accessed *implicitly* by fetching the instructions as we need them. A special program is used when your program is downloaded that takes your program instruction and flashes them into the program memory, but when your program actually runs, this memory is read-only.

Recall that the AVR is **little endian**, meaning that wherever multi-byte values exist, they are always store *least-significant-byte-first*. This is true in program memory, data memory, the 16-bit registers X, Y, and Z, and any other place where multi-byte values are used.

4.4 Input/Output

Microcontrollers like the AVR have built-in I/O capabilities that can do a variety of I/O tasks: digital input and output, analog input (built-in analog-to-digital conversion), timers (lots of built-in timer capability), and serial communications (sending bits to another device, and receiving bits back). The AVR has many neat I/O capabilities, but it also has some weirdness that we need to deal with.

In designing a processor, the designers must consider how I/O will be accessed. Two main approaches are: use special I/O instructions that use special *I/O addresses* to access the various I/O capabilities, or carve out and reserve a section of the memory addresses for use in I/O capabilities, and then use the generic memory load/store instructions for I/O purposes. This is called *memory-mapped I/O* and is possible because input is similar to reading a value from memory, and output is similar to storing a value to memory, except that rather than memory it is an I/O device that is interacted with.

Oddly enough, the AVR does both!

The AVR has a *legacy* I/O system where separate unique *I/O addresses* are used with special I/O instructions: **in, out, sbi, cbi, sbic, sbis**. This I/O capability is still supported, but the I/O capability has been extended and they ran out of I/O addresses; so now I/O operations are also **memory-mapped**, where generic memory instructions can perform I/O operations by accessing special memory addresses. The weird thing about having both modes is that where they overlap, *different addresses* are used in the I/O space and memory space!

I/O addresses are from 0x00 to 0x3F: the bit-oriented instructions (cbi, sbi, sbic, sbis) can only operate on I/O addresses from 0x00 to 0x1F. The instructions (in, out) can operate on any of them.

Memory-mapped I/O addresses are from 0x0020 to 0x00FF. These can only be used by memory instructions (e.g., load and store). The I/O addresses 0x00 to 0x3F correspond to memory addresses 0x0020 to 0x005F: just add 0x20 to the I/O address for the corresponding memory-mapped address. In the AVR documentation of the I/O capabilities, you can see I/O address given, and then the corresponding memory address in parentheses.

Interestingly enough, the register set r0–r31 can also be accessed by the memory addresses 0x0000 to 0x001F! E.g., the instruction "lds r3, 0x0010" copies the value in register r16 to r3.

So, the data memory map looks like:

Address Range	Purpose
0x0100 - 0x08FF	2K SRAM (general purpose RAM)
0x0020 - 0x00FF	Memory Mapped I/O
0x0000 - 0x001F	Registers R0 - R31

If our AVR had more memory installed, the memory would continue above address 0x08FF, potentially all the way up to address 0xFFFF.

The AVR manuals contain the full I/O map. Some of the most used I/O addresses are:

I/O Address	Memory Address	I/O Name
0x03	0x0023	PortB data, input only
0x04	0x0024	PortB data direction set
0x05	0x0025	PortB data, input and output
0x06	0x0026	PortC data, input only
0x07	0x0027	PortC data direction set
0x08	0x0028	PortC data, input and output
0x09	0x0029	PortD data, input only
0x0A	0x002A	PortD data direction set
0x0B	0x002B	PortD data, input and output
0x3D	0x005D	SP-lsb, stack pointer low byte
0x3E	0x005E	SP-msb, stack pointer high byte
0x3F	0x005F	SREG, status register (flags)

Other I/O capabilities that we'll use are above the I/O address space and only accessible using memory instructions and addresses. Especially notable are the A/D (analog-to-digital) capabilities.

5 Addressing Modes

A program must get data to and from memory; this is done with **load** and **store** instructions. We've already learned that accessing memory requires an *address* that specifies the location in memory to access. But how does the address get formed?

To make certain types of programming easier, such as accessing an array of data, the AVR supports several different **addressing modes**, or ways of accessing memory locations. These addressing modes are:

Immediate The actual data is located in memory immediately following the instruction's opcode, or is embedded in the opcode. This is only useful for a constant, it won't work for a variable since the value is embedded in read-only program memory. Thus, immediate addressing is *never* available on store instructions, since they write to memory. On the AVR, immediate addressing is available on load, and some others (add-to-word, compare, subtract, etc.).

Direct The 16-bit address of the data is located immediately following the opcode. This is used for accessing single global variables where the linker knows the address of the variable when it is constructing the executable program file. Since the address is embedded in program memory, it cannot change and so the instruction always accesses the same memory location, but the location is in data memory and so direct addressing can be used for both load and store operations. On the AVR the suffix 's' is used to indicate direct addressing (I have no idea why!), so *lds* and *sts* are the load and store instructions that use direct addressing.

Indirect The value in an index register (X, Y, or Z) is used as the address of the data, potentially in addition to a constant offset. This addressing mode has many uses, including accessing array elements, local variables, function arguments, and other uses. It is very versatile and is really what gives computers their power. In all of computer science, many many problems have been solved by providing a level of indirection and then manipulating the indirection! By storing the address that will be used by the instruction in a register, the address can be modified so that the instruction accesses different memory locations each time it executes (it may be in a loop, or the function it is in may be called multiple times).

On the AVR, X, Y, and Z are the names of 16-bit "registers", which are really the upper six 8-bit registers treated as register pairs: *r27:r26* is X, *r29:r28* is Y, and *r31:r30* is Z (recall that the AVR is little endian, so the lowest numbered register is the lower byte of the 16-bit value). The load and store instructions that use plain indirect addressing have no suffix and are just *ld* and *st*; a form that adds a constant displacement uses the mnemonics *ldd* and *std*. Furthermore, the AVR indirect addressing instructions have *auto-postincrement* and *auto-predecrement* modes where the index register is automatically adjusted by +/- 1 as it is used to address memory. Note that there is *ONLY* post-increment and pre-decrement, which means that for increment mode the register is adjusted by +1 *after* it is used as an address, and for decrement mode the register is adjusted by -1 *before* it is used as an address. There are no pre-increment or post-decrement modes!

Relative (or PC-Relative) This is used for branches and relative jumps/calls, not for accessing data. These instructions need a 16-bit address of the location in the program to branch or jump to, but this address is not stored in the machine code; only an offset is stored that is *relative* to the address of the current instruction (contained in the PC). The actual formula is

$$targetAddress = currentPC + 1 + offset$$

where "currentPC" means the address of the branch/jump/call instruction itself. The offset is a 2's complement offset stored in the instruction's opcode. This offset is only 7 bits for branch instructions (+63 / -64 locations), but is 12 bits for the relative jump/call instructions (+2047 / -2048 locations). Recall that program memory is word addressed, and most instructions are single 16-bit words. This means that typically, branches can branch forward or backward about 64 instructions, and relative jumps or calls about 2048 instructions. There are a few instructions that take two 16-bit words, so the offset count is not necessarily equal to the instruction count.

6 Generating Machine Code from Assembly, and Vice Versa

Assembly programming is the lowest level of *programming* that humans typically do, but the textual program that we write still is not the executable program that the CPU understands. That is the *machine code*, the binary representations of the instructions that we write in our text assembly program.

In the very earliest days of computers, people *did* actually create the machine code by hand (and then entered it byte by byte using toggle switches on the front of the computer!). Today we have assemblers to do this for us, and in reality most low-level programming is not even done in assembly, but in C code (sometimes with assembly code in-lined). But the purpose of this course is to make you better *overall* programmers by learning what is "under the hood", and so for part of this learning we must learn how to translate from an assembly program into machine code, and also learn the opposite: how to look at machine code and figure out what instructions are in there.

6.1 AVR Machine Code Instruction Formats

First, let's look at the format of the machine code that the AVR uses. Remember that the program memory (Flash memory) is addressed *per 16-bit word*, that is, two bytes of program code per address. This can be done because *all AVR machine instructions are either one or two 16-bit words.* In actuality, all AVR instructions are one 16-bit word, but some instructions use a full 16-bit address as one of their operands, and so for them the second 16-bit word is the address.

Every AVR instruction mnemonic has an associated 16-bit machine code value, but since the instruction has operands (registers, constants, and the like), the operand values also need embedded into the 16-bit machine instructions.

Take the basic ADD instruction, which has two register operands, *Rr* and *Rd.* In AVR assembly the destination is always the leftmost operand, so *Rd* is the left operand and *Rr* is the right operand.

The machine code format for ADD (which we find by looking in a machine instruction table) is

```
0000-11rd-dddd-rrrr
```

The 0's and 1's are fixed value bits that are those values for *every* ADD instruction (they are, in essence, the bits that define an ADD instruction). The r's and d's are the bits that hold the operand values. If our program had the instruction *"ADD r25, r17"* in it, then the *r* bits would hold the value 17, and the *d* bits would hold the value 25. The bits above are all in normal magnitude order, left decreasing to right; even though the *r* bits are not contiguous, they still represent one value (in this case, 17).

So, since decimal 25 is binary 11001, and decimal 17 is binary 10001, the instruction *"ADD r25, r17"* is exactly the machine code

```
0000-1111-1001-0001
```

All we did was write the *r* bits in and the *d* bits in, left to right.

Finally, recall that the AVR is a *little endian* CPU. This means that for multi-byte values, it stores the bytes by first starting at the least significant byte (i.e., right to left, but this applies to *bytes*, not bits; within a byte the 8 bits are still left-to-right).

So to finish up the machine code as it is *actually stored in program memory*, we should swap the two bytes, and so our example instruction has the machine code, in memory, as

```
1001-0001-0000-1111
```

which we could write in hex as 0x91 0x0F. Note that when referring to a single machine instruction we will usually just leave it in the normal 16-bit order, but when showing machine code listings or discussing machine code as it is stored in memory, we will view it in little endian (reverse) byte order.

6.2 By-Hand Assembly: from Assembly to Machine Code

The section above shows the basic conversion of a single instruction to machine code. In converting a whole program, this process is followed for each instruction, and the words of machine code are built up as we work down the assembly program. It *really* is that simple, except that various instructions have different formats and different operands, which cause the process to be a bit more complicated, but not much. Below are a list of issues to pay attention to.

Simple two-register instructions such as ADD are very straightforward, as shown above; just remember which register is *Rr* (the right operand) and which is *Rd* (the left operand). All 5 bits of the register number are placed into the instruction opcode.

Some one-register instructions (such as LDI, which has a destination register and an 1-byte immediate constant) have *only 4 bits* for the register number in the opcode; the way these work is that the uppermost bit of the register number is *assumed* to be 1, and is not stored in the opcode; this means that the instruction can only use registers R16 and above, since R15 and below have a 5th bit that is 0. For example, if R23 was used in an LDI instruction, 23 is binary 10111, and so 0111 would be placed in the opcode field.

Some instructions that have a constant operand (such as LDI) *may* have the constant field in the opcode split into multiple sections; for example, LDI's 1-byte constant is placed in the opcode in two separate 4-bit chunks. This should not be a problem, just write the 8 bits as a normal binary number, and fill in the opcode fields in the same order. A common mistake student's make is to *repeat* a small constant in each 4-bit chunk; for example, if we use decimal 6 as our constant in an LDI instruction, the byte value of 6 is 00000110, but students will see the two 4-bit chunks in the LDI opcode and put binary 0110 in each chunk, since decimal 6 fits into 4 bits. Don't do this! The upper 4 bits should be 0000 in this case.

Branch and relative jump/call instructions (rjmp/rcall) have a *signed, 2's complement* offset field; for branches it is 7 bits but for rjmp it is 12 bits. This is the offset *in program words* from the branch/rjmp instruction to the branch target location. A negative offset means the instruction branches/jumps backwards (forming a loop). The offset is actually from the instruction that follows the branch/jump instruction, so the formula for calculating the offset is ($targetAddress - branchAddress + 1$). You can actually use this formula, even doing the arithmetic in hexadecimal since we usually write addresses in hex, but most students prefer to do a simpler, counting method. If you are writing a sequence of machine code words in a table or even just in a list, then:

1. start at the machine code for the branch instruction you are working on (probably blank at the moment, since you haven't filled it in);
2. find the machine code for the target instruction (a label in a program always refers to the next instruction, either on the same line as the label or on the next line if the label is by itself); if the branch is a forward branch you should leave space for the branch instruction, work out the machine code forward through the target instruction, then go back to the branch and do these steps;
3. go forward from the branch instruction one program word (this is the "+1");
4. now count off program words until you reach the target instruction (this is forward if the target is forward, and backwards if the target is backwards); you must reach the target instruction itself, so in going backwards you will count the target instruction opcode itself, to reach the address where it begins, not where it ends;
5. convert your count to binary, putting 0 bits in front to make it as wide as your operand field (7 or 12 bits); if the branch was a forward branch, you are done and this is the offset to put in your opcode field; if the branch is backwards, negate the count by performing the 2's complement operation on it, and put this in the opcode field.

You can do the above on the assembly program listing as well, but you *must be careful to know how many program words each instruction results in*, because the offset is counting machine code program words, not assembly language instructions. *Most* instructions result in just one program word, but a few produce two (the most common for us are the LDS and STS instructions), and for those you must count them twice if you are counting your offset over the assembly program instructions. In the assembly code, start at the branch, go forward to the next instruction, then count off to the target instruction (counting two for any two-program-word instruction). You will get the same offset as the machine code process above.

The direct addressing instructions, such as LDS and STS, require a full 16-bit address as their second program word. This is the data memory address that the instruction will access (read or write). In assignments or exam questions you will be told the starting address of where global variables are allocated from, and then according to the data section of the program (which reserves memory locations for global variables) you can allocate and figure out the actual addresses for each global variable, and then translate from the symbol name to the actual address. This address is also stored little-endian, so the lower byte is stored first and the upper byte is stored last, just as with other machine code words.

6.3 From Machine Code to Assembly: Disassembling the Opcodes

We can also take raw machine code and reconstruct the assembly program that produced it. All we need to do is do everything backwards that we did in the hand assembly steps. Starting at the first program words, we basically do this:

1. write the program word in binary, flipping the bytes back so that the most significant byte is leftmost;
2. find the machine code pattern in the opcode table that matches the actual machine code; *all* of the constant 1's and 0's in the pattern must exactly match the given program word; there will be only one exact match;
3. the pattern shows two things: what instruction this program word is, and what the operand fields are; write the operand fields separate with the bits from the instruction opcode;

4. translate the operand fields from binary into something meaningful in the program, and write it in the reconstructed assembly program.

For example, suppose we had the program word *0x5B 0xE2*, shown as stored in program memory. Since the stored word is little endian, our actual progam word is *0xE2 0x5B*, or binary 11100010 01011011. Looking this up in our opcode table (the one ordered by opcode value), we find that this matches the LDI pattern of 1110-KKKK-dddd-KKKK. So we know this is a load immediate instruction. The constant K field is 8 bits, and pulling those bits out they are 00101011 (the hex digits 2 and B). In either unsigned or in 2C this is the decimal value 43. The *Rd* field is only 4 bits, so our rule about the 5th bit being assumed to be a 1 comes in to play; the field itself is 0101, so our register number is 10101, or decimal 21. So the final instruction that this program word represents is *LDI r21, 43*.

7 Branches

In high-level programming languages, we usually don't have to think about how our code branches – it is implicit in the control structures and the curly braces. For example, we know implicitly that if an **if** condition is false our program will branch to the else clause, and that at the closing brace of a loop body our program will branch back up to the loop condition at the top.

In assembly language, however, these branches are not implicit, but instead they are very explicit – you have to pick them and place them in the right spots.

7.1 Branches encode the relation operators

In high-level languages, we use relational operators in our conditional expressions, and it would be natural to think that these operators have some synonymous assembly language instruction that implements it, but this is not exactly true!

Rather, in assembly code it is the **conditional branch** instructions that encode the relational operator. There is only one kind of instruction that compares values – but there are many different branch instructions that encode the relational operator – such as BRGE for "branch if greater than or equal" and BRNE for "branch if not equal to", and more.

So for relational operators, in assembly language you need to do a two step process: compare and branch. The comparison instruction never changes (except the form in terms of what operands it uses), but the branch instruction changes based on what relational condition you are checking.

7.2 Signed versus unsigned branches

As we learned earlier, with two's complement (2C) representation for signed values, the arithmetic operations don't change – they just work properly! But the place in your program where it DOES matter if your program is using signed or unsigned data is in choosing the branch instructions.

The four relational operators (greater-than, less-than, greater-than-or-equal, and less-than-or-equal) need *eight* different instructions, one each for unsigned and signed data. The AVR CPU actually only implements *four* of these: it does not implement "greater than" and "less than or equal" since you can always just flip your operands around and change them into "less than" and "greater than or equal". The AVR conditional branches for these are:

Operator	Unsigned	Signed
Greater than or equal	BRSH	BRGE
Less than	BRLO	BRLT
Equal	BREQ	BREQ
Not equal	BRNE	BRNE

The two equality checks (equal to and not equal to) are the same for signed and unsigned: BREQ and BRNE. But for the others, it is **very** important for you to pick the right branch depending on your data. The wrong branch simply will not work!

7.3 Branch addressing

Branch instructions need to store their destination as an operand – the place they will branch to if the condition is true. they do not store this as an absolute address (e.g., like extended addressing does), but rather they use a **relative** addressing mode.

Specifically, the AVR branch instruction uses a 7-bit two's complement (i.e., signed) offset relative to the address of the branch opcode, plus 1. Put another way, the target address for the branch is the branch opcode address plus 1 plus the relative offset:

$$\textit{Target-address} = \textit{Branch-opcode-address} + \textit{1} + \textit{Offset}$$

Remember that AVR program memory is *word-addressed*, so a 7-bit 2C offset allows branches to branch backwards or forwards approximately 64 instructions, though a few instructions take 2 words to store and so could affect this rough estimate.

There is also a "relative jump" instruction, RJMP, which has a 12-bit offset and uses the same formula as the branch instructions. This can branch backwards or forwards approximately 2048 addresses.

7.4 How Branch Instructions Interact With Compare Instructions

There's no hidden magic in computers, especially at the assembly language level. So how does the branch instruction know the result of the comparision instruction? The answer is in the **Status Register (SREG)**.

In the next section we explain the SREG in detail. The lower five bits of the SREG (the C, V, Z, N, and S bits) are indicators of arithmetic overflow and condition flags. The thing to remember is that "a comparison is a subtraction". That is, the compare instruction subtracts one operand from the other, and in the process it sets the codes in the SREG. Even though it throws the value result of the subtraction away, the SREG values will tell the next branch instruction whether to branch or not.

In the instruction table, each branch instruction has in its description a mathematical *boolean* formula that looks at the SREG codes and determines whether to branch or not.

8 Condition Code or Status Register (SREG)

We have introduced the idea of a "compare" machine instruction and then a "conditional branch" instruction that depends on the result that was produced by the compare instruction. But where is that result saved? It is in the status register (SREG).

The SREG is an 8-bit register. But it is not treated as an 8-bit value. Each bit is a true/false flag for some condition that you (the programmer) might have to test for. These conditions are set by a wide variety of operations. The compare sets some of them, arithmetic operations set some, etc. The bit flags are 1 if the condition is true, or occurred, and are 0 if it is not true. The bits and flags are:

- Bit 0: the C flag (Carry): A carry bit to/from the most significant bit occurred. For 2's Complement numbers (2C), this is not meaningful. For unsigned numbers, it means the result was too big.
- Bit 1: the Z flag (Zero): The result was exactly zero. NOTE: the Z bit is 1 when "zero" is true!
- Bit 2: the N flag (Negative): For 2C numbers, the result was negative. For unsigned, this flag is meaningless.
- Bit 3: the V flag (Overflow): a 2C result was too big. Analagous to the C flag, but for signed numbers.
- Bit 4: the S flag (Signed): a precomputed result of (N XOR V). Used by signed conditional branches.

- Bit 5: the H flag (Half Carry): A carry occurred from bit 3 to bit 4 (i.e., half the byte). This is useful only when dealing with BCD numbers, something we do not discuss in this course.
- Bit 6: the T flag (Bit Store): A temporary storage place for one bit.
- Bit 7: the I flag (Interrupt): Set to 1 to enable (allow) interrupts to happen. We will talk about interrupts later.

We will mostly use the first five flags. They are the flags signalling results of normal arithmetic and comparison operations.

8.1 Calculating the Arithmetic Condition Flags

In this course you will need to be able to calculate the N, V, C, Z and S condition flags by hand from an arithmetic operation.

Z is the easiest. If the result is 0 (i.e., all 0 bits), then Z=1. Notice that Z is 1 when the result is zero, and Z is 0 when the result is not zero. This is because the Z flag is **true** when the result is zero, and truth is represented by 1.

N is also very easy. It is simply a copy of the uppermost bit of the result. In a byte this is bit number 7 (we count from 0).

C is pretty easy. It represents an unsigned arithmetic overflow or underflow. When the arithmetic operation is an addition, C is 1 if there was a non-zero carry out past the uppermost bits of the operands. Thus, in 8-bit addition, if there was a carry of 1 into the 9th bit, then C=1. For a subtract operation, C is 1 if you had to borrow in from a ficticious 9th bit. That is, if to complete the subtraction you needed to borrow in, then C=1, otherwise C=0.

V is analgous to C except for signed numbers. It represents a signed (2C) overflow or underflow. Unlike C, it is not directly tied to a carry out or borrow in – you can have a carry out in 2C arithmetic and still end up with a valid result! The rule I teach is this: **if the signs don't make sense, V=1, otherwise V=0**.

In other words, do the operation you need to, then look at the signs of the operands and the result, and ask yourself it they are theoretically possible. For example, if you add two positive numbers, it should be impossible to get a negative result. If you did, then you just had a 2C overflow, and V=1. Here's all the rules in a table, but remember, you can always re-create them by just asking "do the signs make sense?"

Sign OP Sign -> Result Sign	V bit
P + P -> P	0
P + P -> N	1
P + N -> any	0
N + N -> P	1
N + N -> N	0
P - P -> any	0
N - N -> any	0
P - N -> P	0
P - N -> N	1
N - P -> N	0
N - P -> P	1

The "official" calculation performed by the CPU to calculate the V bit is: if the last two carries are the same, V=0. If they are different, V=1. Recall that we number bits from the right starting at 0, so the most significant bit in a byte, the eighth bit, is termed bit 7. So if the carry from bit 6 to bit 7 and the carry from bit 7 out are different, then V=1; otherwise they are the same and V=0.

Finally the S bit is simply the XOR (Exclusive OR) of the N and V bits that you computed. XOR can be seen as a "difference" operator, so S=1 if N and V are different, otherwise S=0.

9 Manipulating and Testing Bits

In high level programming languages you have used the logical operators: AND, OR, and NOT. In C/C++/Java, these logical operators are represented as &&, ||, and !. Since each bit is essentially a boolean value, these operators can be applied as **bit operators**, too! In fact even the high level languages allow you to do this, with the operators &, |, ~, and ^ for XOR. The AVR, like most processors, has specific instructions to perform these bitwise operations:

Instruction	Description
AND	bitwise and of two registers
ANDI	bitwise and of register and immediate value
OR	bitwise or of two registers
ORI	bitwise or of register and immediate value
EOR	bitwise exclusive or of two registers
COM	bitwise not of one register (complement)
NEG	2's complement negation of a register

9.1 Shifts and Rotates

There are a variety of instructions that move bits through a register, called **shifts** and **rotates**. These instructions shift the bit values in a register one position either left or right – that is, bit 3 gets the value of bit 2 in a left shift, and it gets the value of bit 4 in a right shift. Well, the question comes up as to what to do with the last bit on either side – i.e., if we are shifting left, what value do we give bit 0? And what do we do with the value of bit 7? Just throw it away? Well, that is why there are different kinds of shifts and rotates.

The **rotate** instructions ROL, ROR treat the register like a circle, except that they use a 9th bit as well – the Carry bit of the SREG register. So, ROL shifts all the bits left (up one position), takes the value of the Carry bit, and copies it to bit 0, and copies the value of bit 7 into the Carry bit. ROR does exactly the opposite—it copies the C bit to bit 7, and copies bit 0 to the C bit.

A **logical** shift instruction pushes a 0 bit value into the end that needs a bit value, and it copies the bit being shifted out into the Carry bit. Thus, LSL (logical shift left) sets bit 0 to the value 0, and copies bit 7 to the Carry bit. An LSR instruction sets bit 7 to 0, and copies bit 0 to the C bit.

An **arithmetic** shift instruction preserves the sign of a 2C number. To do this when shifting right, bit 7 (the sign bit) is copied back to itself, while also being shifted to bit 6. For the ASR, bit 0 is copied to the Carry bit. Well, what about an arithmetic shift left? What should we do with bit 7, the sign bit? It turns out that an ASL instruction would be exactly the same as the LSL instruction. Some CPUs allow both mnemonics to be used for the same machine instruction, but the AVR does not have the ASL mnemonic, only the LSL.

So, what use are the shift instructions? One of the most common uses is a fast way to multiply and divide. A shift left will multiple a number by two – think about what happens to base ten numbers when you multiply by ten – all the digits move over one place, and a 0 is added (e.g., 123*10 == 1230). The same is true for base two numbers when multiplying by two – the bit values just shift one place left, and a 0 bit is added to the right.

If shift left multiplies, then shift right divides by two. And it does. But in this case we need to know if our number is unsigned or is a signed 2C number. If it is unsigned, we need to use LSR. If it is signed, we need to use ASR.

9.2 Setting and Testing Individual Bits

You can use the AND and OR instructions to manipulate and test individual bits, but these operations *modify* the value, and sometimes you just want to check bits without modifying the value.

The AVR has four special "bit test and branch" instructions:

- SBRC: Skip if Bit in Register is Clear: skips the *next* instruction if the specified bit in the specified register is 0.

- SBRS: Skip if Bit in Register is Set: skips the *next* instruction if the specified bit in the specified register is 1.
- SBIC: Skip if Bit in I/O Port is Clear: skips the *next* instruction if the specified bit in the specified I/O address is 0.
- SBIS: Skip if Bit in I/O Port is Set: skips the *next* instruction if the specified bit in the specified I/O address is 1.

These instructions cannot be made to branch *anywhere*, they can only be used to skip the instruction following the bit test instruction. Of course, *that* instruction can be an RJMP that can jump anywhere!

10 Stacks and the SP "Register"

A stack is a data structure that holds data items. It is a special type of data structure, because it is very limited in the way you can access the elements. You can **push** a data item onto the stack, or you can **pop** an item off of the stack. Think of stacking blocks one on top of each other. When you push (or place) a block onto the stack, it is at the top of the stack. When you pop (or take) a block off of the stack, you always take the one at the top. So, the last block on is the first one off. This property is known as **LIFO**, or Last In, First Out. It is the basic characteristic of a stack.

The AVR, like all processors, has explicit support for a stack. This support is in the form of an SP (stack pointer) register, which keeps the address of the top of the stack, and special machine instructions to manipulate the stack. On the AVR the SP "register" is actually two I/O address locations (kind of like the SREG condition flag "register"). The SP-low-byte is at I/O address 0x3d and the SP-high-byte is at 0x3e.

You can use the IN and OUT instructions to read and write the SP value. To push and pop arbitrary registers on and off the stack, the instructions PUSH and POP are available. But there are other instructions that use the stack too, that we'll get to below.

An important thing to know about the AVR is that the stack grows toward lower addresses in memory – that is, the address of the top item of the stack gets smaller as the stack gets fuller. For some, they might consider this downward! We normally think of stacks growing upward, and so we typically draw memory with the largest address on the bottom and the smallest on top.

The startup code in the Arduino environment initializes the SP to the *largest possible address.* For our Arduino board this is address 0x08FF, because we have 2KB of data memory (SRAM), and the memory starts at address 0x0100 since addresses 0x0000 to 0x00FF are used for memory-mapped I/O addresses.

The compiler allocates global variables and other static data from the *lowest possible address*, down at 0x0100. If the stack ever grows so full that it reaches the global data and overwrites it, that is a *serious program failure*! This can and does happen, and can even happen to your C/C++ programs. It is something that you have to be careful about, especially if your programs use recursion.

10.1 Procedures, or Subroutines

The AVR support procedures with the special instructions CALL/RCALL and RET, which respectively call a procedure and return from it. These operations perform a jump in that they jump to someplace else in your code other than continuing sequental execution, but they also use the stack to coordinate those jumps. In particular, a CALL or RCALL instruction jumps to the procedure being called *and pushes the **return address** on the stack.* The return address is the address of the instruction below the call instruction, which is where the procedure should return to when it is done.

The RET instruction pops the return address off of the stack and places it into the PC register, thus effecting a jump back to the caller, at the location where the program should continue.

If a procedure (subroutine) has parameters, on the AVR the parameters are placed in registers r25-r8 (see the register usage conventions), and then if there are more parameters, these are placed on the stack *before the call instruction is executed.* It is your job as the programmer to put the correct number of parameters onto the stack, and also to remove them from the stack after the procedure is done.

10.2 The Stack Frame Pointer

If some of the arguments are passed on the stack, or if some local variables are created, then the procedure must have some mechanism for accessing those. The typical way to do this on most processors, and on the AVR, is to copy the SP register into an index register, and then use indexed addressing to access the necessary items.

On the AVR, the convention is to use the Y index register (r29:r28) for this purpose. We call the register used to hold a copy of SP the *frame pointer*, since it points to the *stack frame (more formally called the **activation record**)* of the procedure. The normal way to copy the SP to the Y index register is simple:

```
in   r28, 0x3d
in   r29, 0x3e
```

Note that you could also use the "LDS" instruction, but in that case you would have to use the equivalent memory addresses of 0x5d and 0x5e.

Once the SP is copied to the Y, then the **key realization** when thinking about how a procedure accesses things in its activation record is that *it does not pop them off the stack but simply accesses them in place, where they are within the stack!*

We can do this with the special indirect addressing instruction *ldd*, which is "indirect with displacement". This allows a constant unsigned offset to be applied to the address in the index register. *We need to simply count out* how many memory locations away from Y are the arguments or local variables that we want to access, and then use "Y+q" where q is that constant offset.

For example, if I had a function with 11 1-byte arguments, the first nine would be passed in the even-numbered registers from r24 down to r8. The last two would be passed on the stack. If I am going to use the Y register then I have to save it first, so the beginning of my function would look like:

```
push  r28
push  r29
in    r28, 0x3d
in    r29, 0x3e
```

So for the rest of my function the stack would look like:

Y+Offset	Contents
Y+0	–empty–
Y+1	saved r29
Y+2	saved r28
Y+3	return addr-low
Y+4	return addr-high
Y+5	argument 10
Y+6	argument 11

So using the *ldd* instruction we can access argument 10 as "Y+5" and argument 11 as "Y+6".

10.3 Call-by-Value and Call-by-Reference

In high level programming languages, there are generally two mechanisms for passing parameters to procedures/functions. These are call-by-value and call-by-reference. Other mechanisms do exist (such as call-by-name), but we will not discuss them here.

With call-by-value, the **value** of the argument is passed to the procedure. On the AVR this means that the value is either loaded into the correct parameter register or is pushed onto the stack. The procedure can then access the value of the argument from the register directly or from the stack using indexed addressing as mentioned above. Call-by-value is good because the procedure cannot alter any of the variables that were used as arguments, because all the procedure gets is a copy of the current value, and doesn't know where it came from. The procedure *can* use its parameter as a "local variable", and can assign to it and change it. But this only changes the copy on the stack, and not anywhere else.

Call-by-reference, as its name implies, is not so safe. With this mechanism, the procedure actually gets a reference to the variable that is being used as an argument in the procedure call. Now the procedure can actually change the variable outside of itself. To do this, we need to pass the **address** of the variable to the procedure, **not** its value!. Of course, addresses on the AVR are 16 bits, so call by reference will always use 2 bytes – either 2 registers or 2 bytes on the stack.

To access the variable in the procedure itself, we need to copy the address into an index register (usually X or Z), and then use indexed addressing with that address to access the variable.

10.4 Saving Registers

An important decision to make when writing a procedure or routine is whether or not the caller of the routine can assume that the values they had in the registers are still there after the routine returns. Since the routine needs to use the same registers (it is the same CPU, after all), one can imagine just telling the caller that they need to reload the registers with values because the registers are corrupted, having been used by the routine to do whatever it needed.

This is generally a frowned-upon solution. What if the routine only uses one register? How does the caller know which registers are corrupted and which aren't? If you told the caller, then how do you keep all the differences between the routines straight while you are programming?

On the AVR we are following the register usage conventions of the *gcc* compiler, and it designates that some registers are freely available for use, while others are "protected" and are not allowed to be modified by the procedure. If we want to use a protected register we must first save its original value, and after we are done with it we must restore the original value.

Where do we save the registers? Well, on the stack, of course. The first thing to do in your routine is to push the protected registers you will use onto the stack. Then, just before your RET to return from the routine, you pop the values back into the registers. **Important**: the pop order is exactly reverse of the push order!

10.5 Local Variables

Procedures need local variables to really be of use, and they must have the same properties as local variables in high-level programming languages. That is, they are unique to the specific call to the procedure, and do not occupy space if the procedure isn't being executed. Each call needs its own copies of local variables, so that recursion works.

How to make this work? The answer is, of course, to put the local variables on the stack. We do this at the top of the procedure, right after we have saved some of the registers on the stack. To "create" local variables, we need to open space up on the stack; we don't really care what initial values are in that space. How to do this? Well, the answer is: *whatever way is easiest!*

Because the SP is out in the I/O address space, if we wanted to simply decrement the SP by a certain amount, say 4 for 4 bytes of local variable space, we could do:

```
    in    r24, 0x3d
    in    r25, 0x3e
    subiw r24, 4
    out   0x3d, r24
    out   0x3e, r25
```

This grabs the current SP, subtracts 4 from it, and stores it back. But that's pretty difficult! We could also do:

```
push  r0
push  r0
push  r0
push  r0
```

Here we don't really care what value is in r0, we're just using it to complete the push instruction; 4 pushes create 4 bytes of space. This is good for a few bytes of local variable space; obviously if we needed 50 bytes of local variables, the subtract method would be better. A *really wierd* mechanism that the compiler will generate to allocate even numbers of variable bytes is:

```
rcall  .+1
rcall  .+1
```

The above also allocates 4 bytes of local variable space – how? Well remember that an RCALL pushes the return address on the stack, which is 2 bytes. The "." in the operand means "current instruction address", and so each of the instructions is "calling" the next instruction, which has no effect! Well, none except pushing 2 bytes on the stack!

Note that the "indirect plus displacement" addressing modes on the Y register (in the manual as *ldd*) only allow an **unsigned** displacement. This means that transferring the SP to the Y register must be done **after** local variable space is created, so that the offsets are positive.

10.6 Return Values

In computer science, we generally use the term *procedure* to refer to a sub-program that does not return a value. We use *function* to refer to a sub-program that does return a value to its caller. In C, almost all sub-programs are functions, even if all they return is an integer status code that indicates whether the computation succeeded or not.

On most CPUs, functions that return an integer or something like it (i.e., anything that fits into an integer-sized register) implement the return value by just leaving it in a register. The AVR does this as well, designating *r24* as the 8-bit return register (*r25* must be 0), and *r25:r24* for a 16-bit return value.

If a function needs to return something else, like a large data structure or something bigger than an integer, several possibilities can be considered. If an address is the same size as an integer, it can leave the address of the actual data in the agreed-upon return value register, and then the caller can copy the data from that address. Or it can place the return data "on the stack", in the area just above the return address. Both of these methods can be combined as well.

11 Input and Output with the AVR

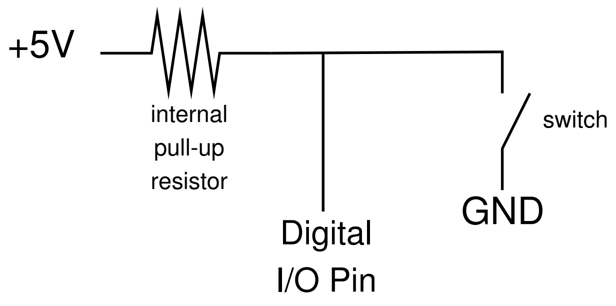
The AVR does I/O and other "extracurricular activities" through various **ports**. Each port is an 8-bit, or 1 byte, port. Each bit in the port typically corresponds to a **real** pin on the CPU, and thus the value of that bit (1 or 0) corresponds to a **real** voltage level on the circuit board. The three basic digital I/O ports are named **B, C,** and **D**. These were presented briefly in Section 4, and there are a variety of byte and bit instructions that can be used to manipulate these ports. Here we will describe some of the details of using these ports for digital I/O.

The first thing to know is that *each bit* in the ports can be treated differently than all the other bits. This means that you could use one bit in Port B for input while using another bit for output, and even another for both input and output. It is up to you how you choose to write your program!

Each of the three ports has three I/O addresses that control it: **PORTx**, **DDRx**, and **PINx**, where *x* is one of B, C, or D. PORTx is used for output, DDRx is used to set the data direction for each bit (pin) in the port (0=input, 1=output), and PINx is used for input. The three ports and their addresses are:

I/O Address	Memory Address	Designation
0x03	0x0023	PINB - input, port B
0x04	0x0024	DDRB - data direction, port B
0x05	0x0025	PORTB - ouput, port B
0x06	0x0026	PINC - input, port C
0x07	0x0027	DDRC - data direction, port C
0x08	0x0028	PORTC - ouput, port C
0x09	0x0029	PIND - input, port D
0x0A	0x002A	DDRD - data direction, port D
0x0B	0x002B	PORTD - ouput, port D

A further capability that these ports have is that each pin in each port has an indivually-selectable internal *pull-up resistor* that can be enabled in input mode. This is very useful when reading devices like a press-switch that is normally not pressed, but when it is pressed the program needs to detect it. An external circuit only needs to connect the switch to ground (0 volts), like such:



When the switch is not pressed, with no pull-up resistor the pin is essentially unconnected to any circuit and would thus have an undetermined value (the manual calls this a “tri-state” mode), but with an internal pull-up resistor enabled, the pin will read as a 1 reliably. Then when the switch is pressed and a connection to ground is made, the pin will read 0. To set a port pin with the pull-up resistor enabled, you need to write a 0 to the appropriate DDRx bit (this sets the pin in input mode) and then write a 1 to the corresponding PORTx bit (this enabled the pull-up resistor). As usual, you read the current input value using the corresponding PINx bit. An input value of 1 will mean “nothing happening”, while an input value of 0 will mean “my switch is pressed!”. Section 13.2.3 in the AVR Technical Reference, and Table 13-1 in that section, provide more detail.

11.1 Bit-Oriented Input and Output

In the above description of ports B, C, and D, we discuss them as 1-byte ports, and they really are 1-byte ports. But very often each individual bit (which ultimately is one individual pin on the microcontroller) is being used all by itself, perhaps as output to light up one LED, or as input to read one individual switch. So while you can use the I/O *in* and *out* instructions, or the memory load and store instructions to do whole-byte input and output, more often it is useful to be able to do bit-oriented input and output.

Thankfully, the AVR has instructions for just this purpose. For output the instructions are:

- *sbi ioport,bit* - this stands for “set bit” and it outputs a 1 value for the bit position given, on the I/O address given; bits are numbered 0-7.
- *cbi ioport,bit* - this stands for “clear bit” and it outputs a 0 value for the bit position given, on the I/O address given; bits are numbered 0-7.

We can use these to output 0 and 1 values on individual I/O pins – very handy!

For input, there is no “load bit” because if we are going to store a bit somewhere it has to be in a register, which is a whole byte, and so we might as well use byte-oriented input. But often we want to just test an input bit rather than saving it somewhere, and so the AVR *does* provide instructions for testing individual I/O bits. The instructions that do this are:

- *sbis ioport,bit* - this stands for "skip if bit set" and it reads the bit position at the I/O address given and if the bit is a 1 it skips the *next* instruction in your program.
- *sbic ioport,bit* - this stands for "skip if bit clear" and it reads the bit position at the I/O address given and if the bit is a 0 it skips the *next* instruction in your program.

Notice that these are basically "compare and branch" instructions, but they cannot branch to any arbitrary label like normal branch instructions can. All they can do is skip the next instruction or not. If the action you need to perform is only one instruction then you can put the instruction right there; but what if you need more? The answer is: put an "rjmp" instruction there and use it to branch. For example, if you wanted to do something if a switch at PORTC, pin 3, is pressed (which reads a 0 value), then you could do something like:

```
        ...
        sbic  PORTC,3
        rjmp  notpressed
        ;
        ; code here that handles the switch press
        ;
notpressed:
        ; continuation of program here
```

If the switch is pressed then the pin reads a 0 bit and the *sbic* instruction skips the jump and executes the code that handles the switch press; otherwise the jump instruction branches around the code and continues on in the program. The example above is only the if-pressed case; if you needed some alternative action for the not-pressed case, you would have to have an if-then-else structure to handle both cases.

11.2 Analog to Digital (A/D) Input

Most "signals" in real life are *analog*, which means that they vary *continuously*, like a continuous function in mathematics: a smooth curve with an infinite number of possible values. Sound, video, and temperature are all continuously varying "signals". In contrast, the *digital* values in a computer are *discrete*, meaning they take on exact values in a finite set. Some real-life signals are also discrete: whether a switch is on or off, or the number of quarters in your pocket. Discrete values are natural to represent in a computer and easy to input, but *analog* signals must somehow be converted into a *digital* representation.

This is done by *discretizing* the range of analog values and then "rounding off" a measured analog value to the closest discrete level. For example, if we knew we would only be measuring temperatures between 0 and 100 degrees Celcius, and we wanted to store the temperature in a 1-byte value, then we must have only 256 discrete levels (0-255 in our byte), and each +1 in our byte will represent about 0.39 degrees Celcius (100/256). So a byte value of 00101011 would represent 43*0.39 = 16.7 degrees Celcius.

Microcontrollers like the AVR all have built-in circuitry that will take an analog electrical signal and convert it into a digital value. On the AVR this defaults to a 10-bit value, thus offering 1,024 discrete values for the analog signal range (but we'll only use 8 bits, thus having a one byte value with only 256 discrete values).

11.3 Programming A/D on the AVR

Programming A/D input is quite a bit harder than just reading or writing a bit for digital input and output. Firstly, the A/D circuitry must be turned on, since by default it is off to save power. Secondly, it must be configured properly to select the right input and to run the A/D conversion. Thirdly, a conversion must be requested, and since it takes time, we must wait for it to finish. Finally, we must retrieve the digital value from the proper (memory mapped) I/O register addresses.

The AVR A/D system uses the following memory-mapped I/O addresses (these must be used with *load/store* instructions, not *in/out* since the addresses are memory addresses only and too big to be in I/O space):

- 0x0078: ADCL: this contains the low byte result;
- 0x0079: ADCH: this contains the high byte result;
- 0x007A: ADCSRA: this contains most of the configuration bits;
- 0x007B: ADCSRB: this contains some configuration bits;
- 0x007C: ADMUX: this contains input selection bits and some other configuration flags;
- 0x007E: DIDR0: this contains flags that turn off digital input;

Recall that the AVR produces a 10-bit result, so at least 2 bits must be stored in a second byte; but which two bits these are is configurable! The AVR can be configured so that either the upper two bits or the lower two bits of the result are in the second byte. We will configure it so that the lower two bits are in a second byte, and then we will just ignore them and use the upper 8 bits as our 1-byte value. Essentially we are rounding off the 10-bit value into an 8-bit value.

Below we treat each of these I/O registers in the most practical sequence, which is not the same as the address order above. The tables show the 8 bits in each I/O register, their names, and then below the tables are the bits' meanings and uses.

ADCSRA: A/D Control Status Register A, 0x007A

7	6	5	4	3	2	1	0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

- ADEN is A/D Enable: writing a 1 in this flag turns the A/D system on (0 turns it off);
- ADSC is A/D Start Conversion: writing a 1 starts an input reading and conversion; when conversion is complete, this bit will read back a 0;
- ADATE is A/D Auto Trigger Enable: this allows an event to automatically start a conversion; we will not use this;
- ADIF is A/D Interrupt Flag (read only): this is set to 1 when a conversion is complete;
- ADIE is A/D Interrupt Enable: setting this to 1 enables the A/D interrupt; we will not use this for now;
- ADPS2-ADPS0 is a 3-bit A/D clock PreScaler: Chapter 23 of the AVR tech manual says that A/D works best with an A/D clock of 50KHz to 200KHz; Table 23-5 shows how these three bits set a value to divide the system clock by; since our system clock is 16MHz, we should set these bits to binary 111 to get a divisor of 128.

ADMUX: A/D Multiplexer Input Source Selection, 0x007C

7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR	empty	MUX3	MUX2	MUX1	MUX0

- REFS1-0 is a 2-bit Reference voltage selector; Table 23-3 shows the options, but we need this to be binary 01 to select the +5V board voltage;
- ADLAR is A/D Left Adjust Result: setting this to 1 causes the high 8 bits of the result to be in ADCH and the lowest 2 in ADCL; setting it to 0 gives the opposite (lowest 8 in ADCL, highest 2 in ADCH); you should set this to 1 and then ignore the lowest 2 bits; *just use ADCH as your 8-bit value*;
- MUX3-MUX0 is a 4-bit input selector: binary 0000 to 0111 select the 8 different input pins of Port C; 1000 selects the internal CPU temperature sensor, 1110 selects the internal CPU voltage, and 1111 selects Ground.

DIDR0: Digital Input Disable Register, 0x007E

7	6	5	4	3	2	1	0
empty	empty	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D

- ADC[0-5]D: when set to 1, the digital input buffers are disabled, which saves power; this is for the pins 0-5 of Port C, which are also used for A/D input; pins 6 and 7 do not have digital input buffers; we should set these to 1 to turn the buffers off.

ADCSRB: A/D Control Status Register B, 0x007C

7	6	5	4	3	2	1	0
empty	ACME	empty	empty	empty	ADTS2	ADTS1	ADTS0

- ACME is A/D Comparator Multiplexer Enable: we will not use this;
- ADTS2-ADTS0 is a 3-bit A/D Auto-Trigger Source: we will not use this;

From the above, we can make A/D programming fairly easy. Firstly, we only need to use ADSRA, ADMUX, and DIDR0 for configuration and control. Secondly, if we set ADLAR to 1 then we only need ADCH for a 1-byte data value. Pretty easy! Programming A/D comes in two parts: one part is the initial setup that only needs done once, at the beginning of your program. The other part is the things you need to do each time you want to read a new input value.

Initialization :

- In DIDR0, initialize ADC[0-5]D all to 1 to turn off digital input buffers;
- In ADMUX, initialize REFS1-REFS0 to 01 (select +5V as reference), ADLAR to 1 (left adjust output bits), and MUX3-0 to 0000 (we'll change the MUX bits later on);
- In ADCSRA, initialize ADEN to 1 (turn on A/D), ADPS2-0 to 111 (divide system clock by 128), and the rest to 0 bits;

Reading a value :

- In ADMUX, set MUX2-0 to the desired input pin;
- In ADCSRA, set ADSC to 1 to start a conversion;
- In a loop, read ADSC until it reads 0;
- Fetch result byte from ADCH; ignore the lowest 2 bytes in ADCL

Once you have your byte value, it's up to you to make sense of it. All the AVR knows is that it read some A/D input channel, created a digital value, and gave it to you in a byte. What that "means" in your application program is up to you.

Also note that you must redo all of the non-initialization steps each time you want to read a new value. If you simply load from ADCH, you will just get whatever value was in there before. To get an actual new sensor reading you must do all four steps for reading a value each time.

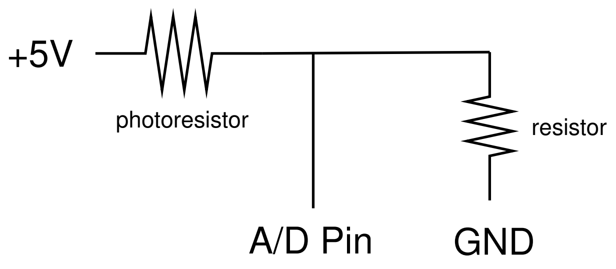
11.4 Wiring Light Sensors (photoresistors)

We will use A/D mainly for "light sensing". Our light sensors are actually just pretty cheap photoresistors. A regular resistor has a constant resistance, but a photoresistor's resistance varies depending on how much light is falling on it.

The most basic electronics equation is

$$V = IR$$

which says that the voltage drop V is equal to the product of the current flowing (I) and the resistance R it is flowing through. If we create a circuit that looks like:



We can sense the voltage drop on an A/D input line, and thus obtain a value representing the amount of light falling on the photoresistor. The A/D pin itself does not complete a circuit to Ground and thus does not draw much current, so we need a regular resistor in series with the photoresistor to complete a circuit from our +5V power source to Ground. The voltage drop across both resistors is always 5 (+5V to 0V), but the voltage drop to the A/D pin will vary depending on how much light is falling on the photoresistor (which will cause the current draw to vary since the voltage drop is constant). A good regular resistor size to choose is a 1K resistor, since this is on the order of the resistance range of the photoresistor.

12 The Arduino 7-Segment LED Display Shield

The Arduino board is designed so that daughterboards can be designed and created that plug into the Arduino and add some sort of functionality. In Arduino lingo these are shields. In this lab we are going to use the "7-Segment Display Shield", which has a 4-digit calculator-like LED display, along with a digital thermometer, a large multi-color LED, and some EEPROM. In this course we will use the 7-segment LED display, the thermometer, and maybe the multi-color LED. The display is called a "7 segment" display because each digit is created by turning on or off the 7 segments that fully make up an "8". As you might guess, turning each of these on or off is controlled by a single bit value!

12.1 The I2C Protocol

The way that the shield board communicates is a very simple serial protocol (serial means "one bit at a time") called I^2C that only uses two pins: one for a clock signal and one for the data that is being communicated. You can learn more about the I^2C protocol at Wikipedia:I2C. These two pins are controlled as output pins just like the pins you connected single LEDs to in previous labs. Your assembly code will make the data and clock signals go up (1) and down (0) according to the I^2C protocol.

The data signal is known as SDA and the clock signal is known as SCL. SDA is connected to PORTC, pin 4, and SCL is connected to PORTC, pin 5. I^2C has a maximum speed of 100KHz, which means that both the high and low periods of the clock signal must be at least 1/200,000 of a second (thus together making 1/100,000 of a second). Since we are already using a "delay()" function that takes arguments as milliseconds, we will run much slower, just invoking "delay(1)" whenever we need some time to pass. This means we will run the I^2C communication at 500Hz or so.

The I^2C protocol is pretty simple: For a master to send data to a slave it does the following:

- sends a start bit: this is a high-to-low transition of SDA while SCL is high; in assembly, SDA is set to 1 while SCL = 0, then SCL is set to 1 and we delay; then SDA is set to 0, creating the high-to-low transition; then we delay some more; then SCL is set to 0; we again delay some more to finish the clock period;
- sends an 7-bit address+r/w bit: Each bit is one clock period: for a 1 bit, SDA = 1 all the while the clock is high; for a 0 bit SDA is 0 during the high clock. The 7-segment display address plus a write bit is 0x70 (for reading it is 0x71, but we won't use that);
- sends multiple 8-bit bytes, each bit as with the address sending;
- sends a stop bit: like the start bit but a low-to-high transition.

Bytes are sent highest-bit-first; in other words, I2C is big-endian, the opposite of the AVR processor! Oh well!

After each byte (including the address), the slave will send an acknowledgement bit back indicating success, but the master must still generate the clock period for the ACK bit. So each byte actually needs nine clock periods generated. Our program will ignore the ACK (slightly unsafe!), but we still need the clock period.

12.2 Using the Display

We must use the I2C protocol to communicate with the devices on the shield. The most important device is the 7-segment display itself, which is driven by a particular chip on the shield, the SAA1064. To learn ALOT about the chip that drives the display, you can view the SAA1064 Datasheet. Page 5 shows the "write mode" protocol that we are using.

You can get pretty complicated if you want on how to control the display, but we will just use the mode where we update all 4 display sections together. In this simple mode, we will always send the following:

- A Start bit
- Address byte == 0x70
- Instruction byte == 0x00
- Subcommand byte == 0x47
- Data 1 == byte to drive digit 1 (rightmost)
- Data 2 == byte to drive digit 2
- Data 3 == byte to drive digit 3
- Data 4 == byte to drive digit 4 (leftmost)
- A Stop bit

A byte value determines what is displayed by the following mapping, where the number is the bit position (lowest bit is 0):

```
  0
5   1
  6
4   2
  3
    7
```

where bit 7 (the uppermost in the byte) is the decimal point "segment". A 1 turns the respective segment on, and a 0 turns it off.

12.3 Using the Digital Thermometer

The LED display shield actually has a thermometer on it – it is the tiny chip in one corner with "DigiTher" printed next to it. We can read it using the I2C protocol. Its base I2C address is 0x92, but as described above the lowest 8th bit is set to 0 to indicate write and 1 to indicate read (for the 7-segment display we only did writes). Another way to look at it is that the thermometer's write address is the byte 0x92 and its read address is the byte 0x93). You can learn a lot more about the thermometer at *http://www.ti.com/product/tmp175* if you want to (but you do not need to for this course).

To talk to the thermometer, we need to do some initialization, just once at the beginning of your program:

- Send a start bit;
- Send address byte 0x92 (7-bit address + 0 bit for write mode);

- Send control byte 0x01 (indicating writing to the config register);
- Send byte 0x60 (configure for full 12 bits of resolution); finally a stop bit. Remember that each of the three bytes needs an ACK clock period, too.
- Send a stop bit;
- Send a start (yes, this needs to be a second message);
- Send address byte 0x92 (7-bit address + 0 bit for write mode);
- Send control byte 0x00, this tells the thermometer to send the temperature data when we do our reads;
- Send the stop bit.

Those two messages configure the thermometer chip to be read; we only need to do this once, then we can read as many times as we want.

To read the temperature, we need to generate the clock periods, start and stop bits, and I2C address as with our writes, but *input* data rather than output it. This is how:

- Send a start bit;
- Send the address byte 0x93, NOTE: 7-bit address plus a *1 bit* to indicate a read; don't forget ACK;
- Read a byte: generate each bit's clock period and while clock is high, do a bit read; this is somewhat tricky; below this list is a function *readByte()* that you can call and it will leave the byte value read in r24.
- Read another byte; the thermometer responds with two data bytes; so call *readByte()* again (after saving r24 somewhere!);
- Send a stop bit.

The first byte that you get back is the signed integer number of degrees in Celcius. The thermometer has a range of -55 to +128. The second byte that you get is the unsigned fractional part to add to the integer part, in the count of 256ths. For example, if our first byte was binary 00010110, this is 16+4+2 = 22 degrees Celcius, and if the second byte is binary 11000000, this is 128+64 = 192 256ths more, or 192/256 = 0.75; this means you would add 0.75 to 22 and the temperature reading is 22.75 degrees Celcius. Note that if the first byte is a negative number you still would *add* the positive fractional part; if for example the first byte was -13 and the second byte was as above, the reading would be -12.25 degrees Celcius.

To read a data byte from the thermometer, use the following functions:

```
#
# Read a byte from the I2C bus (assumes all initialization has been done,
# and that the start bit and address have been sent; this only reads one
# byte (and does the ACK), caller must complete the communication, reading
# more bytes, sending a stop bit, etc.)
#
      .global readI2CByte
readI2CByte:
      push  r16               ; save r16 in case program is using it
      push  r17               ; save r17 in case program is using it
      cbi   PORTC, SDA        ; ensure output is low to switch to input
      cbi   DDIRC, SDA        ; change SDA pin to input rather than output
      ldi   r16, 8            ; we're going to read 8 bits
      clr   r17               ; r17 will hold data byte, so start it at 0
readLoop:
      lsl   r17               ; shift the bits we have so far one place to left
      sbi   PORTC, SCL        ; set clock high
      call  I2CDelay        ; keep high for a bit, gives time for therm to send bit
      sbic  PINC, SDA         ; skip next instruction if input bit is 0
      ori   r17, 0x01         ; input bit is a 1, so put it into data byte
      cbi   PORTC, SCL        ; set clock low
      call  I2CDelay        ; keep low for a bit
      dec   r16               ; decrement our loop counter
      brne  readLoop          ; if it is still not 0, go back to top of loop
readDone:
      sbi   DDIRC, SDA        ; change SDA pin back to output
      cbi   PORTC, SDA        ; set data line low for ACK
      sbi   PORTC, SCL        ; start ACK clock period
      call  I2CDelay        ; hold high
      cbi   PORTC, SCL        ; set clock low
      call  I2CDelay        ; hold low
      mov   r24, r17          ; move data over to return value register r24
      pop   r17               ; restore original r17
      pop   r16               ; restore original r16
      ret                     ; data byte is left in r24

      .extern delayMicroseconds     ; need to tell assembler we are using this library function
I2CDelay:
      ldi   r24, 50
      ldi   r25, 0
      call  delayMicroseconds
      ret
```

Note that in our lab experience, we could run the I2C protocol quite slow for communicating with the 7-segment display chip (we could use the Arduino millisecond delay function), but the digital thermometer chip was more picky; we had to speed up the protocol with the delay function in the above example for everything to work.

13 The Arduino Motor Shield

Your motor shield just plugs into the main Arduino board. Its purpose is to provide an interface between motors and the Arduino board. A schematic of the motor shield can be found on the course website. The motor shield has three main chips on it, a reset button, two five-pin output terminals, and several other components (capacitors, resistors, etc.). The shield supports the control of four motors, but we're only going to use two.

13.1 Motor Shield Overview and Pin Connections

Looking at the schematic, you'll see a chip labeled IC3 (part number 74HCT595N) in the lower left corner. This chip is an 8-bit serial in/out shift register with parallel output latches. This chip implements the interface to the Arduino board and thus controls how we need to program the motors. As the name describes, the chip takes 8-bits of data in serially (one bit at a time) into a shift register, then it latches the contents of the shift register so that these bits are seen on the output. These 8 bits of parallel output control the 4 motors, two bits each.

Notice from the schematic the output bits. M1A - M4B are the 8-bits that map to each of the four motors, two bits per motor. Motor 1 is mapped to bits 2 and 3; bits 0 and 6 for motor 2; bits 1 and 4 for motor 3; and bits 5 and 7 for motor 4. Setting these bits to 01 will turn the motor in one direction; setting them to 10 will rotate the motor in the opposite direction. Setting both bits for a motor to 00 or 11 will stop the motor, but you should avoid using 11.

Notice the two, five-pin output terminals on the motor shield. These are labeled M1-M4 for each of the four motors, which tells you where to connect your motors to the shield. The outer pins on each of these terminals interface to the individual motors; the middle pins are grounds (which we won't use).

In addition to the IC3 chip, there are two other main chips on the motor shield. These L293D chips provide a higher amperage current to the motors than the on-board signals (bits) that the Arduino generates directly (which can light LEDs well enough but would not be enough to even turn the motors!)

The L293D chips also have an "enable" pin for each pair of its outputs. This allows another channel of motor control and is used for Pulse Width Modulation, which is a fancy name for turning the motors on and off really fast, to slow down the motors from their full-on speed. These enable pins are seen on the schematic as the PWM signals (PWM0A, PWM0B, PWM2A, PWM2B) These signals are directly connected to Arduino pins in PORTD (J1) and PORTB (J3).

Although we are not going to control motor speed in this lab, you will need to set these signals to enable the latched motor byte to be seen on the motor output terminals (M1-M4). You can do this by outputting a 1 on the pin that maps to each PWM output signal. On PORTB, PWM2A is connected to pin 3; on PORTD, PWM0A and PWM0B are connected to pins 6 ans 5, respectively, and PWM2B is connected to pin 3. The template code has this in it.

From the motor perspective, in the motor control byte, motor 1 is mapped to bits 2 and 3; bits 0 and 6 for motor 2; bits 1 and 4 for motor 3; and bits 5 and 7 for motor 4. Setting these bits to 01 will turn the motor in one direction; setting them to 10 will rotate the motor in the opposite direction. Setting both bits for a motor to 00 or 11 will stop the motor, but you should avoid using 11.

Notice the two, five-screw output terminals on the motor shield. These are labeled M1-M4 for each of the four motors, which tells you where to connect your motors to the shield. The outer screws on each of these terminals interface to the individual motors; the middle screws are grounds (which we won't use). Looking at the motor shield from above with the 5-screw terminal with "M4" and "M3" written underneath it, the bits are mapped to the 5-screw terminals as:

bit0	bit6	Gnd	bit5	bit7
M4	M4		M3	M3
		Internal		
M1	M1		M2	M2
bit2	bit3	Gnd	bit1	bit4

Each motor also has an enable line that must be 1 in order for the motor to turn on. These are defined as M1ENABLE through M4ENABLE, and are connected to PORTB and PORTD. All of the pins we will use are:

Name	Port	Pin
MOTDATA	PORTB	0
MOTCLOCK	PORTD	4
MOTLATCH	PORTB	4
M1ENABLE	PORTB	3
M2ENABLE	PORTD	3
M3ENABLE	PORTD	5
M4ENABLE	PORTD	6
BOARDLED	PORTB	5

The last symbol is just the Arduino on-board LED.

13.2 Shift Register Protocol

To enable control of the motors, you have to follow a specific protocol just as you did when sending data to the 7-segment display. This protocol involves the bits comprising the motor byte (or the data; DIR_SER on the schematic, we call MOTDATA in code) and an associated clock (DIR_CLK on the schematic; MOTCLOCK in code), and a clock associated with the latch (DIR_LATCH on schematic; MOTLATCH in code). You can see on the schematic that DIR_SER is connected to PORTB, pin0; DIR_CLK to PORTD, pin4; DIR_LATCH to PORTB, pin4. This protocol is pretty simple. The basic protocol is as follows:

1. Send a single bit (MOTDATA) of the motor byte to the shift register.
2. Set the data clock (MOTCLOCK) to high for at least 1 ms; then reset and send the next bit.
3. Once all 8 bits have been sent to the shift register, latch the data to the output by setting the latch clock signal (MOTLATCH) to high.

4. Set the PWM outputs to high.
5. Reset the latch clock (MOTLATCH) by clearing the bit.

To send each bit of the motor byte to the shift register:

1. Clear or reset the latch (MOTLATCH).
2. Based on the bits in your motor byte (MOTDATA), send either a “0” or a “1”.
3. Set the motor clock signal (MOTCLOCK) high and keep it high for at least 1 ms.
4. Clear the motor clock signal (MOTCLOCK) to set the clock back to low.
5. Clear the MOTDATA bit so you’re ready for the next bit.
6. Delay for at least 1ms before you send the next bit.

Once all 8 bits of the motor byte have been sent to the shift register, you need to latch this byte to the motor output terminals. After the bits are latched, you should reset the latch by clearing the bit. To do this:

1. Set the latch clock signal (MOTLATCH) to high. Keep this signal high for at least 1ms.
2. Set the proper MOTENABLE signal(s) to high to enable the motor byte to be seen on the output terminals.
3. Reset the latch clock signal (MOTLATCH) by clearing the bit.

14 Interrupts and Timers

14.1 Interrupts

An **interrupt** is an event which stops the normal execution of your program, makes the CPU go off and do something else (like respond to the condition that caused the interrupt). When the CPU finishes what it needed to do, your program continues execution whereever it left off, as if the interrupt never happened (except, time has passed, and some memory locations could be changed).

All computers have interrupts. If you’ve ever installed a peripheral device onto a PC, you may have had to select “IRQ” addresses, or have seen the system select them automatically in the case of plug-n-play. “IRQ” just means “interrupt request”. Under Linux, you can look at the information in */proc/interrupts* to see the active interrupts.

Interrupts allow a system to be much more efficient, and thus faster. I/O devices are usually much much slower than the CPU. If the CPU had to wait for an I/O device to finish each time it used it, your PC would run much slower. By using interrupts to let the device tell the CPU that it is ready for more data, or it has more data for the CPU, or it is finished, the PC can continue doing something else while the device is completing its task.

How do interrupts work? Well, when an interrupt occurs, the CPU hardware recognizes and figures out which interrupt occurred. The CPU expects that there is some program code that must be executed in response to the interrupt (this is the code that will take care of whatever needs to be done). The CPU finds this code by using a table called an *interrupt vector*; this table is just an array of addresses, where each entry is the address of the code that should be executed in response to the interrupt that is associated with that array index. The AVR ATmega328P (the version on our Arduino boards) interrupt vector is:

Vector # Address	Entry Source	Name	Description
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete

The interrupt vector table is always at a fixed location in memory; for the AVR, as can be seen in the second column, the table begins at address 0x0000 in flash (program) memory, and ends at address 0x002B. Reading down the names of the interrupts, you can see that some are directly triggered by external circuitry connected to external pins, some are communication oriented, to handle the "slow" nature of data transfer, there is one for A/D conversion indication, and quite a few related to timers.

The first entry in the table is a *psuedo-interrupt* in the sense that it doesn't have a current program to get back to; it is the *power up* condition. Did you ever wonder how a CPU knows what to do when you first turn it on? Well the first entry is it! This *RESET* interrupt is triggered whenever the CPU is power-reset or software-reset; this entry contains the address of the code that needs executed first when the CPU starts up. In the Arduino environment this code eventually leads to the function *main()* being called, which is where all C programs start.

14.2 Interrupt Processing

The code that is executed in response to an interrupt is known as an *interrupt handler* or an *interrupt service routine (ISR)*. In assembly code we would write it much like a function, and indeed it is invoked much like a function, using the stack to store the *return address*, which is the address in the program that was executing where control must go back to when the interrupt handler is finished.

However, since the handler was NOT called as a normal function, we cannot use the RET instruction to return from it; rather there is a special RETI instruction to use, which means "return from interrupt".

When an interrupt handler returns back to the program that was executing, it is *very important that no registers have been changed*! The program that was executing must have all its data still in place. Some CPU's accomplish this by automatically pushing *all* registers onto the stack when an interrupt handler is invoked. The AVR processor does not do this; rather it leaves it up to the programmer to save whichever registers it needs to. For an interrupt handler, the normal register usage conventions DO NOT apply – ALL registers must be treated as "protect if used". Registers are saved just as in procedures, by pushing them on the stack.

THIS INCLUDES the SREG status register! An interrupt handler could actually be invoked in between any two instructions in a program, and this means that it could execute between a comparison instruction and the conditional branch instruction that will use the results of the comparison. So if the interrupt handler needs to execute any instructions that change any flags in the SREG register, it needs to save the current SREG value.

IMPORTANT: Interrupt handlers must always be kept SHORT and QUICK. Many times students will try to write their whole program as a response to some interrupt. The problem is that *further interrupts are disabled until the current handler finishes!* Thus, interrupt handlers should always be very short pieces of code which simply do something immediately doable and/or just set a flag for later code to check. Your goal always is to return from the interrupt handler as quickly as possible.

14.3 Timers

As seen in the above table, quite a few interrupts are related to timers. This is very typical of small processors meant to be used in embedded systems. Think of your microwave oven; almost everything the processor in that system needs to do is associated with keeping track of time.

Using a timer interrupt is essentially the same as setting your alarm clock; you configure the timer to run at a certain rate and then set a specific trigger point, and the interrupt happens when the timer reaches that trigger point. And just like your alarm clock that is automatically reset for the next day, the timer resets and will cause an interrupt the next time it hits the trigger point (and the next, and the next, etc., until it is turned off). In this manner the timer acts like a metronome, giving you a tick (an interrupt) at a very precise and regular interval.

Timers work as what is known as "free running counters"; that is, they are a self- contained circuit and register that can do a "timer++" action at the rate that you specify. Setting a trigger point means setting a value such that when the timer value reaches that value, an interrupt occurs. One natural point to trigger an interrupt is when the timer value reaches its maximum and "overflows" (it wraps around to 0). In the interrupt vector you can see three overflow interrupts.

The AVR has three timers: 0, 1, and 2. Timer 0 and 2 are 8-bit timers, while timer 1 is a 16 bit timer. There are also quite complex mechanisms available to set up timer interrupts, with various comparator setups. We are going to keep it simple and just use the overflow interrupts.

The memory-mapped I/O addresses associated with the timers are somewhat scattered:

- Timer 0 controls are down in the "pure" I/O space, with memory addresses 0x44 to 0x48 (I/O addresses 0x24 to 0x28); all the rest are only in memory space;
- Timer 1 controls are 0x80 to 0x8B;
- Timer 2 controls are 0xB0 to 0xB4;
- All three timer interrupt controls are at 0x6E to 0x70;

14.4 Selecting the Timer Rate

The timer speed is selected by three bits for each timer; these bits are called CSn0, CSn1, CSn2 and are the lowest three bits in their respective TCCRnB control registers. The meaning of these bits are:

CSn2	CSn1	CSn0	Description
0	0	0	No clock source (timer stopped)
0	0	1	Clk-I/O (no prescaling)
0	1	0	Clk-I/O / 8
0	1	1	Clk-I/O / 64
1	0	0	Clk-I/O / 256
1	0	1	Clk-I/O / 1024
1	1	0	External on T0 pin, falling edge
1	1	1	External on T0 pin, rising edge

My only problem right now is that *I do not know how Clk-I/O differs from the main clock Clk-CPU.* The manual clearly refers to them separately but it seems like a table is missing somewhere. I **suspect** that clock-I/O is the main CPU clock divided by 256, but I'm not sure.

In any case, the most logical setting for us is 101, selecting a divisor of 1024; because the system clock is fairly fast (16MHz), we want to slow it (even through Clk-I/O) down as much as possible.

The only other thing we need to do is enable the interrupt for the timer overflow. This is done in bit 0 of the TIMSKn control registers. Each names bit 0 as TOIE, meaning "timer overflow interrupt enable".

14.5 Installing Our Interrupt Handler

Presently we are just doing this in the C/C++ code environment that Arduino supports. There is a nice macro that is pre-defined which allows you to write an interrupt handler very similar to a function. The code looks like:

```
ISR(interrupt_name)
{
   // body of your interrupt handler here
}
```

The *interupt_name* must be a valid pre-defined interrupt name, the table below gives the names for the Atmega328P:

INT0_vect	INT1_vect
PCINT0_vect	PCINT1_vect
PCINT2_vect	WDT_vect
TIMER2_COMPA_vect	TIMER2_COMPB_vect
TIMER2_OVF_vect	TIMER1_CAPT_vect
TIMER1_COMPA_vect	TIMER1_COMPB_vect
TIMER1_OVF_vect	TIMER0_COMPA_vect
TIMER0_COMPB_vect	TIMER0_OVF_vect
SPI_STC_vect	USART_RX_vect
USART_UDRE_vect	USART_TX_vect
ADC_vect	EE_READY_vect
ANALOG_COMP_vect	TWI_vect
SPM_READY_vect	

15 Making C Pointers on the AVR

Pointers in the C programming language, or references in the Java language, are variables that "point" to a data variable – they are not data themselves (though they are variables!). How would you create a pointer variable on the AVR? Well, the only addressing mode that allows an instruction to access changing memory locations is the **indirect** addressing mode. We've used it up till now to access arrays (the changing address just stepped through the different elements in an array), and to access function arguments on the stack.

We can use indirect addressing mode to create pointers. Consider the following C code:

```
int i;
int *pi;

void func()
{
   i = 4;
   pi = &i;
   *pi = i + 3;
}
```

Let's translate this into AVR assembly language. First, the data declarations:

```
  .data
      .comm i,1
      .comm pi,2
```

The variable *i* is just one byte because the AVR uses 1-byte integers (mostly). But why is *pi* two bytes? Well, pi must "point" to some data location, and thus it must hold an address. On the AVR, addresses are 16 bits, or 2 bytes. Now let's add some code, for all but the last C statement:

```
  .data
      .comm i,1
      .comm pi,2

      .text
      .global func
func:
      ldi   r17, 4
      sts   i, r17
      ldi   r30, lo8(i)
      ldi   r31, hi8(i)
      adiw  r30, 3
      sts   pi, r30
      sts   pi+1, r31
```

So, what is going on here? Well, the first two instructions are easy. They load the register r17 with a constant 4, and then store it into the variable i using direct addressing. The next instructions looks odd, but are correct. They load the value of the **label** i – not the value of the variable i – into the Z register (which is r31:r30 treated as one 16-bit register). Think of these instructions as the "&" operator – in fact, when C programmers speak, they say "address of" to denote the "&" operator. Remember, all labels are address values, so the Z register now has the address of *i* in it.

The next instruction adds 3 to the *word* r31:r30, and then we store the 16-bit word into the variable *pi* using direct addressing (the second byte must go in the next memory location, so we use +1 on the label). The variable *pi*, which holds an 16-bit address, is now "pointing to" the variable i! This is exactly how a pointer variable operates. When it comes down to the machine instruction level, a pointer (or reference) variable is really just a variable that holds the address of another variable.

16 Serial Communications

We said earlier that the AVR has the ability to do serial communication, which is what the Arduino environment uses when you open the "Serial Comm" window and enter text to send and see printed text that the program outputs.

16.1 Basic Concepts of Serial Communication

Parallel communication is when all the bits of a value are sent to the receiver at the same time, i.e., in **parallel**. To do this you need a wire for each bit. This is the way a CPU sends and receives data from memory, and disks, and things like that. Parallel communication needs a lot of wires, but it can be very fast.

Serial communication sends each bit one at a time. You only need one (data) wire in this case, so this matches well when you need to communicate over long wires, like phone or network lines. All networks use serial communication, and some slow devices, like printers, often can use serial communication.

A **simplex** serial connection sends data in only one direction. One data wire is needed.

A **half-duplex** serial connection sends data in both directions, but not at the same time. One data wire is needed, and its direction is switched back and forth.

A **full duplex** serial connection sends data in both directions simultaneously. Two data wires are needed, one in each direction. This is the most common type of serial connection, with each side having a receive (often abbreviated Rx) and a transmit (often abbreviated Tx) pin, with a wire connecting the receive of one side to the transmit of the other side.

Our USB connections from the Arduino to a PC are full duplex connections. In addition to two data wires, a USB connection has one wire for a common Ground (0 volts) reference, and one wire for a +5 volts connection, which can be used for powering a small USB device (like your iPod), or recharging it.

Both sides must, of course, agree on a signalling protocol to be able to transmit data. USB defines its own signalling protocol, but a peripheral chip on the Arduino board manages the USB protocol, so a program that uses serial communication does not need to know about it. Instead the AVR processor has a USART (Universal Synchronous-Asynchronous Receiver Transmitter) programming mechanism, and a program simply needs to abide by its rules and use its I/O registers in order to program serial I/O on the Arduino.

16.2 Setting up Serial I/O on the AVR

Describe USART programming here. TODO.

16.3 ASCII Character Representation

When you use the Serial Comm window in the Arduino IDE to send strings back and forth, like "Hello World 123", each character that you type is a byte that is sent to the Miniboard. The question is, what byte value is each character (like 'H'), and how did we know that?

Well, an agreed-upon standard for byte values representing characters is the **ASCII** standard (American Standard Code for Information Interchange), and it assigns a number value to each character you can type, including punctuation (and some that you cannot type). So, for example, the character 'A' is decimal value 65, or hex $41, and 'a' is decimal 97, or hex $61. The letters are numerically in order for upper and lower case, so 'B' is $42, 'b' is $62, and on all the way to 'Z' and 'z'. (question: how do you capitalize an ASCII letter?) In between 'Z' ($5A) and 'a' ($61) is a few punctuation characters ('[' is $5B, '\' is $5C, ']' is $5D,...).

We started by asking what 'H' is, but we also have to ask about the '1', '2', and '3' also. These are actually characters, and are not numerically 1, 2, or 3. Rather the character '1' that we type has an ASCII value of decimal 49, hex $31, and the characters for each of the digits are in order from '0', which is $30: '1' is $31, '2' is $32',..., '9' is $39.

On a Unix workstation, you can type in the command "man ascii", and you will see a chart of all of the ASCII characters.

ASCII originally used 7-bit characters, and so only had 128 possible characters. **ISO 8859-1** extended that to 8-bit characters (different manufacturers had already used 8-bit characters, with the upper 128 values being proprietary characters (like line-drawing on MS-DOS)).

But even 256 values is not nearly enough to represent, for example, Chinese characters. ISO 8859-1 does well with basic European languages with the various accented characters, but that's about it.

A new standard called **Unicode** (ISO 10646 – Universal Character Set (UCS)) has been devised that uses 16-bit characters. With 16 bits – i.e., 65,534 different values – every character of every known language (including all the many thousands of Asian lanuage characters) can have its own unique value. One nice thing about Unicode is that values with the upper byte of $00 map directly to ISO 8859-1, and thus to ASCII. So 8-bit ASCII characters will be around for a while. An encoding of Unicode called UTF is popular as a method to preserve backwards compatibility with ASCII strings.

On our Linux workstations, "man unicode" and "man utf-8" will give you more information.

17 Combinational Circuits, Boolean Algebra, and Flip Flops

In this class we have seen basic bit, or **boolean**, operations – AND, OR, NOT, and XOR. We said that these operations are not only useful for logical conditions (in high level programmig languages), but are also useful at the bit level – for testing and changing individual bits. Since a bit is a 1 or a 0, and a logic value is True or False, these operations are equivalent whether you are talking about bits or logical values.

Actually, these basic operations are also the basic components that the physical circuits on a chip are built out of – and in digital electronic terms, they are called **gates**. Well, actually the lowest level on a chip are transistors – essentially electronic switches, but then these are used to build gates. So, all of the CPU operations that we can program at assembly level are basically designed into the hardware in terms of gates. And it takes alot of gates – CPU's like the Pentium have millions of gates.

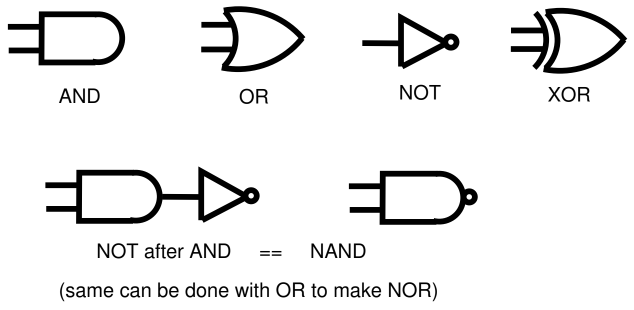
These basic boolean operations form a discrete mathematical system called boolean algebra. It obeys many of the properties that we are used to seeing in math, and more:

- OR is represented by a plus sign, AND by a dot (or nothing, like multiplication). NOT is a bar over the term (but sometimes a prime mark), and XOR is most often a plus sign inside of a circle
- like integer math's plus operator, 0 is the boolean identity value for OR – that is, anything OR'd with zero is itself.
- like integer math's multiply operator, 1 is the boolean identity value for AND – that is, anything AND'd with 1 is itself.
- AND and OR are associative, commutative, and distributive.
- NOT defines an inverse value
- DeMorgan's Laws
    1. NOT(AND(A, B)) is the same as OR(NOT(A), NOT(B))
    2. NOT(OR(A, B)) is the same as AND(NOT(A), NOT(B))

Just like in regular math, we can write down formulas in boolean math.

(in class examples)

Since circuit designers like to draw their circuits rather than just write formulas, there are a set of diagrams for each operator in Boolean logic:



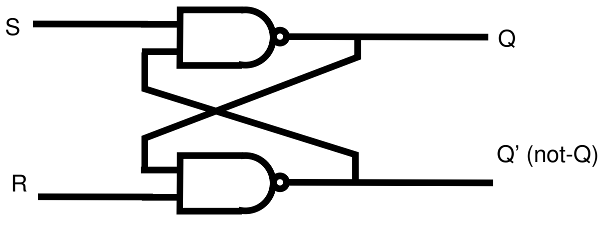
As an example for how these gates can be constructed out of switches (or transistors), the following diagram shows an AND gate and an OR gate made out of switches.

TODO: Diagram: gates from transistors

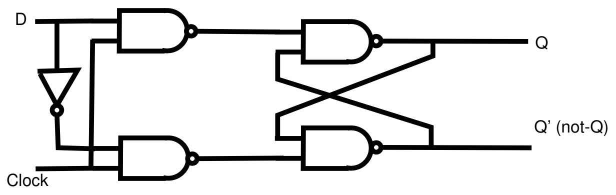
17.1 Flip Flops

Ok, the basic boolean operators let you compute values, but with a computer we also like to store values – i.e., have some memory that we can work with. So how do we do that?

Well, what happens if we connect some logic gates in a special circular fashion?



This circuit essentially stores one bit of data! But we don't have a "data" line into it, we just have two controls (set and reset). To make a data-in line and a data-out line takes a little more doing:



This circuit is essentially a one-bit storage circuit. If you put eight of them in parallel, you have one 8-bit register on the AVR!

18 Programming with Real Numbers and their Floating Point Representation

Thus far in class we have dealt only with integer values. We have seen a simple, direct unsigned binary representation for positive integers, and the Two's Complement (2C) representation for signed integers (both positive and negative).

Higher-level programming languages offer us the ability to use **real** numbers, simply because many problems need to calculate fractional values, not just integers. It is **extremely** useful for a programmer to understand how these real numbers are stored, so that they understand what might go wrong when they use real values.

This tutorial and exercise will take you through several steps of how real values might be supported, and will end up explaining the true method that languages such as C/C++/Java use.

18.1 Idea 1: Fixed point representation

One possible method for storing and using fractional values at the binary level is to have a **fixed point** representation. That is, with the bits you have in your data value, just pick a point at which the bits to the right are right of the "decimal" point, and the bits to the left are left of the "decimal" point. Actually, since we are using the binary number system, we should call the point a *binary point*, and we will do so from now on (when talking about binary values).

Thus, if we have 8 bits in our values, we could pick a fixed point representation of:

```
iiiii.fff
```

That is, 5 bits to the left, and 3 bits to the right of the binary point. Conversion to a decimal value is done exactly like we did for integers, with the positions to the right of the binary point being negative exponents. For example, the value 01011.101 would be

$$2^3 + 2^1 + 2^0 + 2^{-1} + 2^{-3} = 11.625$$

in decimal. With our representation, we can have values from 0.0 (our format is unsigned) to 31.875, in steps of 0.125 (0.001 binary)

[Recall that negative exponents are reciprocals of positive exponents, so $2^{-2} = 1/4$, $2^{-3} = 1/8$, etc.]

A fixed point format is useful, and it has been used in computer design, but the problem is that it doesn't scale to different problems. What if all the values we had were smaller than 1? This representation would waste all of the five bits in front of the binary point, and we wouldn't have very much accuracy with only three bits.

18.2 Idea 2: Floating point representation

A better solution would be to allow the decimal point to float – in other words have a **floating point** representation. Actually, this is already familiar to us in the form of scientific notation. Think of a scientific notation where you are only allowed to write down, let's say, 8 digits (not including the 10 that is the base of the exponent) – and that 3 of those digits are in the exponent, and five in the value.

With such a format, we could write down a number like $4.3275 * 10^{182}$, or a number like $8.4832 * 10^{003}$, or even (if we can have a negative exponent) $6.3491 * 10^{-821}$.

The important notion here is that we always have five digits of accuracy, even though our numbers range from very big to extremely small. We are always able to use all our digits of accuracy. That is the advantage of floating point numbers.

Ok, we can do the same thing in binary. But we need some names. The value that is the exponent is called the **exponent**, and the value of our number without the exponent is called the **mantissa**. That is, in the first example in the above paragraph, 4.3275 is the mantissa, and 182 is the exponent. 10 is the **base** of the exponent, and when we do this in binary, the base will be 2.

So, let's take our 8 bits and pick the same dividing point for the mantissa and exponent – that is:

```
mmmmm eee
```

So we have 5 mantissa bits and 3 exponent bits. For now, let's assume nothing is signed, and that we put a binary point to the right of the first mantissa bit. So, for example, the 8-bit value 10101011 is interpreted as $1.0101 * 2^{011}$. All of the conversions we have learned still apply, so this number is just

$$(2^0 + 2^{-2} + 2^{-4}) * 2^3$$

which is (1.3125 * 8) == 10.5. Notice that the mantissa 1.0101 just gets shifted over 3 places, since the exponent is 3. This is just like decimal scientific notation, there is nothing new going on here – it just happens to all be in binary!

With our representation, the biggest number we can represent is a bit value 11111111, or 1.1111 * 2^7, which is decimal 248. Notice that the next smallest number we can represent is 11110111, or 1.1110 * 2^7, which is 240. So we skipped some integer values!

Well, this should be no surprise – it is the same as with decimal scientific notation. If we limit ourselves to 2 mantissa digits in base-10 scientific notation, we can write $1.3 * 10^3$ and $1.2 * 10^3$, which are the numbers 1,300 and 1,200 respectively. But without more mantissa digits, we cannot write the numbers between 1,300 and 1,200. So it is the same with binary floating point.

This is an important lesson, because it underscores the need for the programmer to understand the accuracy of the floating point representation. It is not infinitely accurate.

Lastly, the only thing we haven't dealt with is signs. We will talk about this in the next section, but just notice that there are **two** signs needed – one on the mantissa, signalling a negative number, and one on the exponent, signalling a very small number (the binary point is moving far left).

18.3 IEEE Floating Point Representation

In the olden days (oh, 15 years ago and more), computer manufacturers (actually, the CPU chip designers) basically picked their own floating point representation. They would pick different sized mantissas and exponents, and handle signs in different ways, and the end result of this would be that a program that ran on different machines might get different results!

Well, a standard representation was devised by the IEEE, and now essentially all computers use the same formats for floating point representation. This is called the IEEE floating point standard (big surprise!).

The standard is actually a class of several representations of different sizes: **single precision** is 32 bits, **double precision** is 64 bits, and **quad precision** is 128 bits. In the C programming language, these correspond to the **float**, **double**, and **long double** data types.

In single precision the IEEE format has 1 sign bit, 8 exponent bits, and 23 mantissa bits, in that order from left to right (the 32 bits are seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm). The sign bit is for the sign of the mantissa (i.e., the sign of the overall number). Exponents of all 1's and all 0's are reserved, and the exponent stored in the bits is the actual exponent + 127. That way, the exponent stored as a positive unsigned number, but it represents exponent values from -126 to +127. The exponent is a power of 2, of course.

In addition to the mantissa, there is a hidden bit that is a 1 bit tacked onto the front of the mantissa. If you think about it, a binary mantissa always begins with 1 since we don't write leading 0's on numbers. So the IEEE format just assumes that a 1 is there, and doesn't store it. It is a free extra bit of accuracy.

So, the number represented by a single precision IEEE number is

```
Value = s * 1.mmmmmmmmmmmmmmmmmmmmmmm * 2 ^ (eeeeeeee - 127)
```

In decimal terms, this gives a number with about 7 digits of accuracy, and magnitudes from about 10^{-38} to 10^{38}.

In double precision (64 bit) IEEE format, the mantissa is 52 bits, and the exponent is 11 bits (with an offset of 1023). It gives us almost 16 decimal digits of precision, and magnitudes from 10^{-308} to 10^{308}. This is a much larger range than single precision. Quad precision is 112 bits of mantissa, 15 of exponent.

We said earlier that exponents of all 1's and all 0's are reserved. This is for special error conditions, like trying to divide by 0, or taking the square root of a negative number.

An exponent of all 1's is considered to be infinity – positive infinity if the sign bit is 0, negative infinity if the sign bit is 1. Dividing a non- zero number by zero results in infinity.

An exponent of all 0's is considered to be not-a-number, or NaN for short. Dividing 0 by 0, or taking the square root of a negative number, will result in a NaN value.

18.4 Floating Point Accuracy

When using floating point numbers, you must always remember that *the value that is stored is probably an approximation of the real value.* Because of this approximation, errors can accumulate until your end result might not be correct anymore – in terms of the actual physics or underlying theory of whatever you are calculating.

For an easy example, imagine I have a two digit decimal floating point format. So I can write numbers like 1.0, 3.2, 0.7, and such. In my computation I want to assign the value one-third to a variable X, but in my format, my variable X will only hold the rounded-down value of 0.3 instead of one-third. Now if I do the assignment Y=X+X+X, Y will end up holding the value 0.9, *even though the true value of three times one-third is 1.0, which is representable in my format!.*

You can actually take entire semesters of math courses that deal with problems like this. There are many issues in taking problems specified in ideal, infinite mathematics and translating them into the finite and discrete-valued realm of actual computation on computers.

Scientists used to have to be very careful, since different CPUs used different FP formats, to write their programs so that they would get the same results. Now that all computers use IEEE format, programmers and scientists are being less careful – the result is that even though all different computers will get the same result for their program, this result might be wrong!

19 Assembly Language Syntax Guide

This section provides a brief explanation of the syntax that is used in the assembly language we use for Arduino programming. Most of the syntax for the instructions is designed by the designers of the AVR microcontroller, but some of the organizational syntax is dependent on the particular assembler we are using, which is the Gnu assembler, part of the entire Gnu compiler toolset. The Gnu compiler tools are a very large toolset that is used on many different computer platforms, and so the entire syntax of the Gnu assembler is far beyond the purposes of this course. You can find an exhaustive description at *http://sourceware.org/binutils/docs-2.22/as/index.html* .

19.1 Assembly Language Overview

The typical parts of an assembly language program are:

- **instructions** are the actual instructions we want the processor to execute;

- **directives** are commands for the assembler itself, not instructions for the processor; these tell the assembler the extra information it needs to be able to properly assemble your program, and offer conveniences to make your program more readable; in Gnu assembly all directives begin with a dot ('.').
- **labels** are names we give for particular positions in our data and in our program; labels end up being translated into addresses, either data addresses or program addresses;
- **comments** are always important in any programming endeavor, and we need to have the ability to create comments in assembly programming as well; although the Gnu assembler allows C-style comments (/*...*/), the better style is: a line beginning with a #-sign is a whole-line comment, and any occurrence of a semicolon (;) indicates that the rest of the line is a comment.

An assembly file is typically organized with an indented column of instructions; only comments and labels ever begin a line without any whitespace in front of them (whitespace is tabs or spaces). Instructions and directives are indented at a constant indentation (6 or 8 spaces is a good indentation), and then instruction and directive operands are spaced over to form a second column.

Below is an example assembly program file with a standard organization, which is explained below the program.

```
#
# Symbolic constants
#
      .set  PORTC,0x08        ; symbolic constant for PORT C I/O address
      .set  DDIRC,0x07        ; symbolic constant for PORT C data dir I/O address
      .set  LEDB,5            ; symbolic constant for pin position of on-board LED

#
# Global data
#
      .data                   ; directive that starts the data section
      .comm val1,1            ; creates 1-byte global variable named val1
      .comm val2,1            ; creates 1-byte global variable named val2

#
# Program code
#
      .text                   ; directive that starts the program section
      .global compute         ; tells assembler to make label compute externally available
compute:
      lds   r18, val2         ; finally this is the first actual AVR instruction
      ldi   r19, 23           ; the first four instructions do:
      add   r18, r19          ;    val1 = val2 + 23
      sts   val1, r18
      ldi   r18, 0x20         ; set bit 5 to be 1 so that we can turn on the LED
      out   DDIRC, r18        ; tell the AVR we want pin 5 to be in output mode
      out   PORTC, r18        ; now output our 1 bit to pin 5, thus turning on the LED
      ret
```

The program above displays most of the basic syntax we will use, except for declaring arrays. The three sections in the program are important.

The first section is where we define *symbolic constants* using the ".set" directive. These allow us to use the symbol name in our program anywhere we would have needed to use the actual value. This makes your programs *MUCH* more readable, especially when creating programs that use a lot of I/O ports. You should use symbolic constants where ever appropriate. These are *not* variables, and do not exist when your program is running; the assembler substitutes the symbol name for its value as it assembles your program.

The second section is where we declare any global data (variables) that we need. This section always must begin with the ".data" directive. We can never put any actual machine instructions in this section, it is only for declaring data, and so the only thing here will be other directives. The code above uses the ".comm" directive (which means 'common', or shared) twice to create two variables. The operands for ".comm" are the variable name and then its size in bytes; the assembler then reserves that much space for it and equates the name of the variable with the address of where it is stored; whenever the variable name is used in the program, the address is substituted for the name just like a symbolic constant's value is substituted.

The third section is where we actually create our program, and this must begin with the ".text" directive. Calling the program code "text" goes back many decades to the very early days of computers. Don't ask me where it came from, I have no idea! In our code section we create the functions that will make up the program. In the above code there is only one function, but there could be many (with no need to repeat the ".text"). *The function's name is simply a label in front of the first instruction of the function.* To make the function accessible outside of the assembly file to other pieces of the program, we must use the ".global" directive to tell the assembler to share this label with other parts of the program; otherwise the label would "disappear" once this file was assembled into machine code. All functions end with a return instruction, which causes the execution to return to whomever called the function.

A *label* is a named location in the program; in this program "compute" is the only label in the program, but the variable names "val1" and "val2" can also be considered labels since they refer to data locations. Labels in the program (and sometimes in the data section) are always a name that begins at the beginning of the line (no leading spaces are allowed) and ends with a colon (which is not part of the label name). Function names are labels at the beginning of the function, and we use labels inside functions to indicate places we need to branch to, such as the top or exit of a loop, and if-then-else blocks.

Notice also that the assembler understands both decimal (plain) and hexadecimal (leading 0x) numeric constants; it also understands octal (with a leading 0), and binary (with a leading 0b).

19.2 Other Useful Directives and Syntax

Other assembler directives and syntax we will use include:

- *.extern symbol* tells the assembler that "symbol" is defined in some other part of the program, but we want to use it in our assembly code; one place we will use this is for library functions such as *delay()*.
- *.byte* allows us to declare initialized byte values in data memory; for example, the line "vals: .byte 23, 44, 99, 31" would allocate four bytes of memory and initialize them in order with the values given, and would equate the label "vals" to the address of the first memory byte, thus creating a 4-element array named "vals". We could also create an uninitialized array by doing ".comm vals,4".
- *.ascii "string"* creates the byte constants in memory for the characters inside the double quotes; as with *.byte*, a label is needed to refer to the string. *Warning:* C-style strings must always end with an extra byte value of 0 after the last character in the strings, and this directive DOES NOT create that byte. Usually you should always use *.asciz* instead of the plain version. It will create the 0 byte.
- *lo8(symbol)* and *hi8(symbol)* are built-in assembler functions for accessing the lowest byte (8 bits) of a 16-bit value and the highest byte of a value; these are almost always used on a symbol that represents an address, usually something like an array name, and is used to load the individual bytes of an address into an indirect addressing register, one byte at a time.