5: The ISA -- The Computer's Native Language

Summary: In this chapter, we'll start to move toward how to build a computer. This involves a substantial change in tack because building a physical machine that understands Python code turns out to be very difficult. In this chapter, we will describe what the components of an actual computer are--the computer's architecture--and then we will describe a language for manipulating these components--the computer's instruction set. This instruction set will be the actual interface for the computer, as opposed to the friendly interface exposed by the compiler.

5.1: Where are we now?

We now take another step down the tower of abstractions:

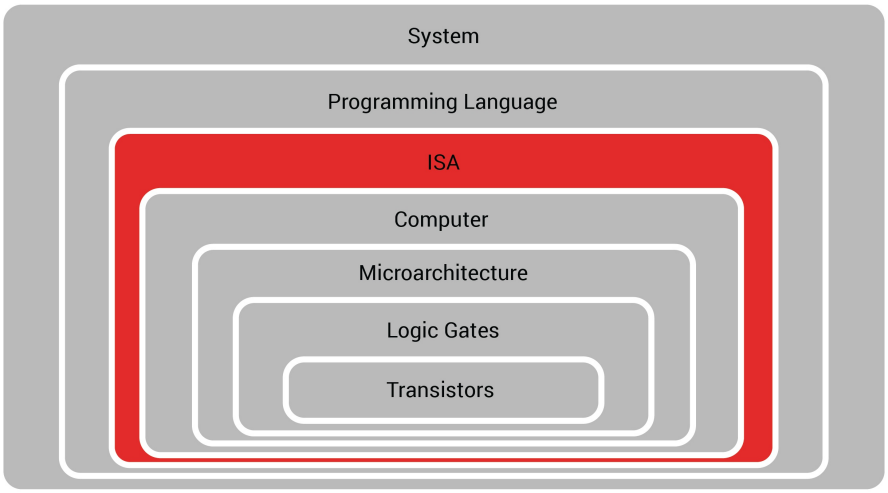

Figure 5.1.1:

We have learned to program a computer using Python. But the computer is a machine, and for reasons we will get into in this chapter, it is very hard to build a machine that understands Python. The language that the computer itself actually understands is called the ISA.

As such, the ISA will finally truly answer the question "How does one program a computer?". We can program a computer in Python, but in a sense we're not programming the computer directly--we're letting a compiler program the computer by turning our program into an ISA program and handing that off to the computer.

The ISA consists of the actual commands that the computer can execute, and as such is the computer's interface. In this chapter, we will explore it in detail and understand how to program in the ISA directly.

This will be critically important for our approach to the second question about how to build a computer. Since if we're going to build a machine, we have to know what kinds of inputs the machine should accept and what it should do with them--the machine's interface. So knowing the ISA will tell us exactly how the machine we want to build should behave.

5.2: What is an ISA?

As we come to the study of the computer itself, we intend to describe two things: The computer's interface and its implementation. We have already described one method for interacting with a computer--the Python programming language. Up to now, this has been our interface to the computer. However, as we said in chapter 1 and as the diagram suggests, Python is not the interface to the computer itself, but only to the compiler, which in turn interacts with the computer via its actual interface.

Indeed, Python has a number of features that make it inappropriate as an interface for an actual computer: A computer is a physical machine, with various actual physical components, like finite storage space and limited computation circuits. In contrast, Python allowed us to assume things like:

- We have essentially infinite memory that we can access at will: Python doesn't specify a limit on how many variables we can have or how large an array can be.
- We can access memory using our chosen variable names without worrying about precisely which slots in memory we are using: In Python we can blithely go 'x = 5' and not have to worry about where we're storing the number 5--we can trust that something (the Python compiler, in this case) figures out which memory slot to use.
- We can store arbitrarily large numbers and operate on them: Python will very happily compute and print the very very large numbers we computed in our factorial program. Now that we know

numbers in a computer are actually made up of bits, and that any memory slot in a computer tends to have a fixed number of bits, we cannot expect that storing small numbers requiring 8 bits to be the same as storing large numbers that require 100 bits. Python, however, allowed us to ignore this subtlety.

So Python is not a suitable language to be understood directly by a physical machine. In this chapter, we will introduce the native language of a particular sort of computer, known as that computer's **instruction set architecture**, or **ISA**. Any ISA consists of three things:

- The computer's **architecture**--that is, the basic physical components of the computer that we have at our disposal. For instance, we won't have an infinite memory whose slots can be given names and can store huge numbers or even text, as Python might have us believe, but we'll have some finite memory whose slots can be accessed by a numerical address and which can store numbers up to some bound.
- The **instruction set**--that is, the list of instructions the computer understands. These will be specified in terms of the architecture. For example, the computer will have several different memories, and there will be an instruction that takes the value from one memory and adds it to the value stored in another memory.
- The **encoding**. In this chapter, we'll learn about the instructions using a human-readable notation. For instance, the add instruction applied to certain memory slots could be written

  ```
  add r13,r18
  ```

  Such a representation is called the **mnemonic** for the add instruction. However, when a computer reads its instructions, it cannot read them as mnemonics--it can't literally look at these letters and figure out what to do. Remember that computers deal only in numbers, and so our machine requires that the instructions be given to it as numbers. The ISA also includes a scheme for corresponding instructions to numbers suitable for storage in a computer. This scheme is called the encoding of the instructions. For example, we will see later that the above instruction corresponds to the number 3794, and what the computer actually stores in place of this add instruction as part of a program is this number.

As one might expect, there are many different designs for computers, all with very different ISAs.

If you're reading this using a laptop or desktop computer in the 2010's, say, your computer's architecture is likely one known as x86. This is a very complicated architecture with many different components, and has a corresponding panoply of instructions that control all these components.

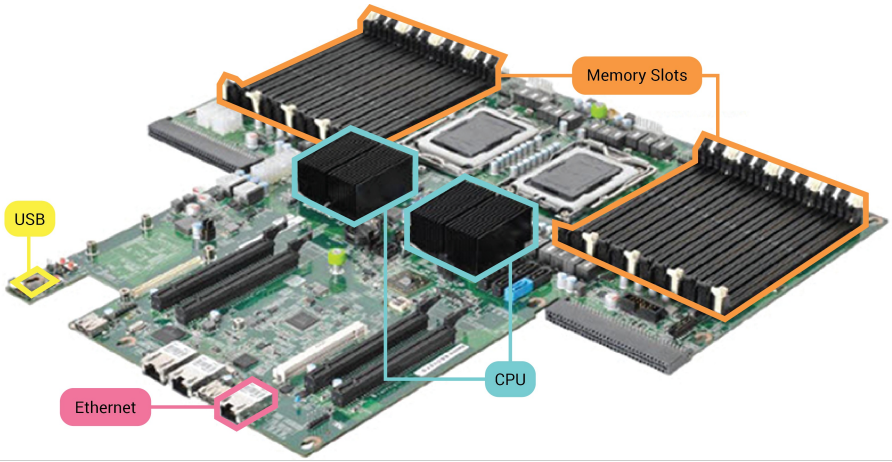

Figure 5.2.1: A circuit board that would go inside a desktop computer with an x86 processor (two of them, in fact!), slots for adding in more memory, and connectors for input/output (for mouse, keyboard, internet, etc.)

If you're on a tablet or phone or particularly low-power laptop, your computer's architecture is more likely to be the ARM architecture, which is simpler than x86 but is still a modern processor with many features.

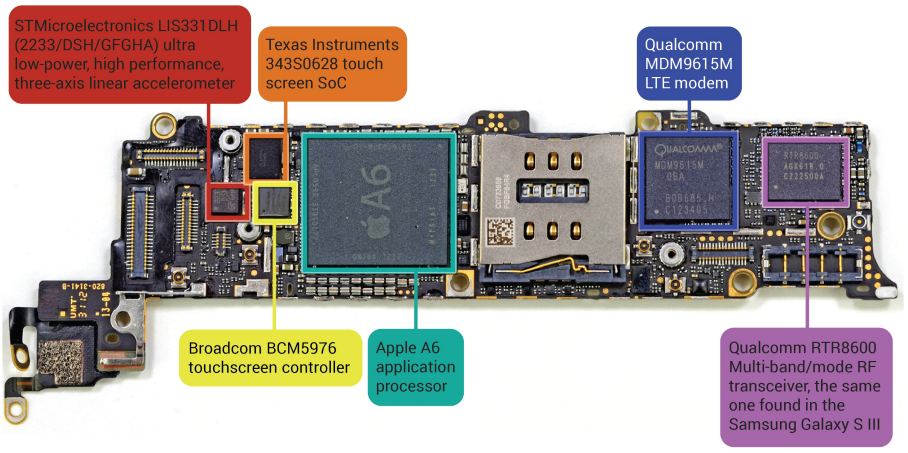

Figure 5.2.2: Photo of circuit board from inside an iPhone, which contains an ARM processor, as well as an accelerometer chip (as we've been describing) and several other modules for controlling the touch screen, accessing the internet, and connecting to the cellular network..

It turns out that while modern processors have way too many features for us to describe in this text, most of the advanced features of a complicated modern processor are expansions on top of a basic set of features that are common even among simpler computers.

In this book, then, we'll focus on the architecture of one such sort of simpler computer--the AVR family of microprocessors--that exemplifies the basic components but which lacks much of the added complication that is used to make desktop computers and phones as fast as they are. Actual AVR computers in the wild tend to be used for simpler things like the processor in charge of your thermostat or blender or car. They are also the computer used in the Arduino boards, which have become popular among hobbyists.

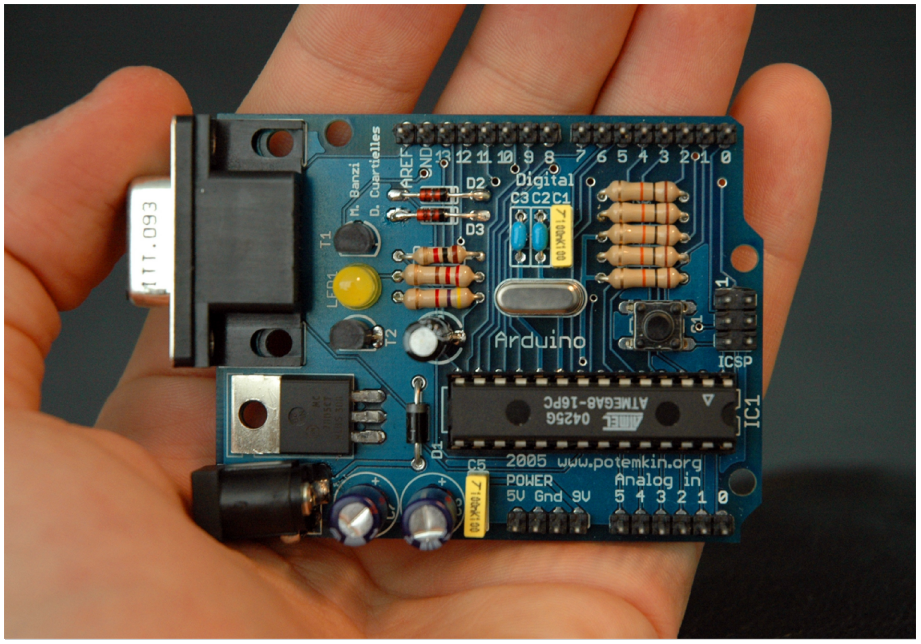

Figure 5.2.3:

Once we understand the AVR ISA, we will want to actually program an AVR computer using it.

We are able to simply write mnemonics for instructions from the ISA and have a program translate these to the encoded form--a sequence of numbers--that the computer will understand. This program is called an **assembler**. Most assemblers are a little more sophisticated than just encoding instructions from mnemonics and have extra features that make writing the instructions a bit easier. These extra features take the form of assembler **directives**. For instance, if you want an instruction repeated 32 times, you don't have to write the instruction 32 times, but can write a directive preceding the instruction that will tell the assembler to make 32 copies of the instruction for you. The language that the assembler understands is a programming language called **assembly language**. The ISA

instructions are a huge part of assembly language, but as we mentioned, there are is a small part beyond that consisting of the assembler directives which make it slightly more concise than just using the ISA instructions themselves.

In this chapter, we will start by discussing the AVR architecture (5.1) and instruction set (5.2 for basic instructions, 5.3 for more complicated ones), electing to leave all discussion of the encoding to chapter 6. We will then introduce the assembler and explain some features of assembly language that help with common tasks when writing instructions (5.4). We then launch into a sequence of examples, first seeing how Python programs, complicated though they appear, can be realized as assembly language programs (5.5 and 5.6), and then seeing (5.7) how our applications from the preceding chapters look at the level of the machine's native language.

5.3: Computer architecture

Summary: The AVR architecture consists of a collection of memories, all with different roles

As we described, a computer's architecture describes the actual pieces of the physical machine that the instructions will manipulate. In the AVR architecture, almost all of these pieces are memories: It has a large memory for storing all the data it needs for its operation, a second large memory for storing the instructions, a small, fast memory that stores the values we are currently operating on, and some additional small memories to store things like status and which instruction we should execute next, and that's it. All the instructions will do is store values in the memories, move values between the memories, and add/subtract/etc. numbers stored in the memories.

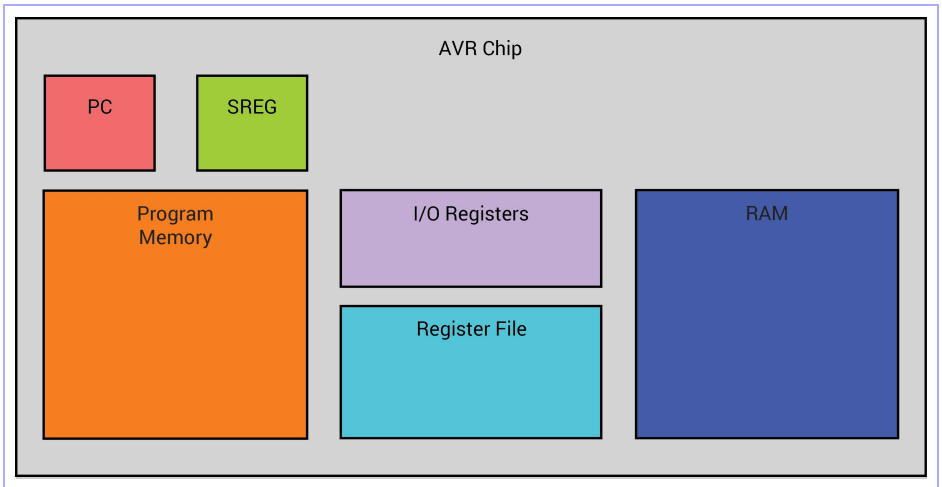

Figure 5.3.1: Overview of the AVR architecture

Before we get to all this, though, it behooves us to describe what we mean by "memory" more precisely.

5.3.1: What is a memory?

A note about what we mean when we talk about a memory is in order:

A **memory** is a component that can store integer numbers. It consists of a sequence of slots, each of which stores a single number consisting of a given number of bits. The number of slots that the memory has is called the memory's **size**. The number of bits each slot stores is called that slot's **width**.

The AVR architecture has several types of memory for different purposes. For example, the RAM, as we will see, has a size of 65536 and a width of 8. This means that it consists of 65536 slots to store numbers, each of which is 8 bits, as in:

Slot 0:	0	1	1	1	0	0	0	1
Slot 1:	1	0	1	0	0	0	1	1
...	...							
Slot 65535:	0	0	0	1	1	0	0	0

This gives an idea of what RAM might look like at any given time--65536 slots, each of which consists of 8 bits, populated with some random values depending on what the computer is doing.

In this picture, we have numbered each of the 65536 slots as 0-65535. In general, we will use such a numbering to refer to which slot in a memory we are talking about. The number corresponding to a slot is called the slot's **address**. In this example, address 1 in RAM contains the value 10100011,

representing the number 163.

Of course, like all parts to the computer, RAM is actually a physical thing, so slot 1 actually consists of 8 physical devices each of which is capable of storing electrons. In this example the first, third, seventh, and eighth of these actually contain electrons (represented by 1s) and the rest do not (represented by 0s).

The computer's architecture consists of several different memories, which we describe in terms of width and size, as well as how long storage operations on the memories take, whether the memories can be edited while the computer is running, and what, generally speaking, the memories are used for:

Memory:	RAM	Program Memory (PM)	Register file (RF)	PC	SREG	I/O registers
Width:	8	16	8	16	8	8
Allowed values:	0-255	0-65535	0-255	0-65535	0-255	0-255
Size:	65536	65536	32	1	1	various
Speed:	Slow	Slow	Very fast	Very fast	Very fast	various
Editable by instructions:	Yes	No	Yes	Sort of	Sort of	Sometimes
Use:	Storing large amounts of data that is not all needed immediately	Storing the actual instructions that are being executed	Storing any numbers that we are manipulating immediately	Storing the address in the program memory of the instruction we are currently executing	Storing information about the most recent arithmetic operation the computer performed	These are used for communicating between our computer and the external world.
Examples:	If you need to store some names, they would generally be stored in RAM.	Every instruction you want to run has to be stored in program memory	If you want to add two numbers or compare them or whatever, the numbers must be stored in registers	If this is 5, then the instruction that will be executed is the instruction in program memory at address 5.	If the last computation was an addition and the result was too big to store in a register (i.e. the result was bigger than 255), then the status register will be modified in a way to indicate this.	If our computer has some electrical connections coming out of it, we can send electrons flowing on these or not by writing values to the I/O registers.

In the next few sections, we'll give further details on each of these.

5.3.2: RAM

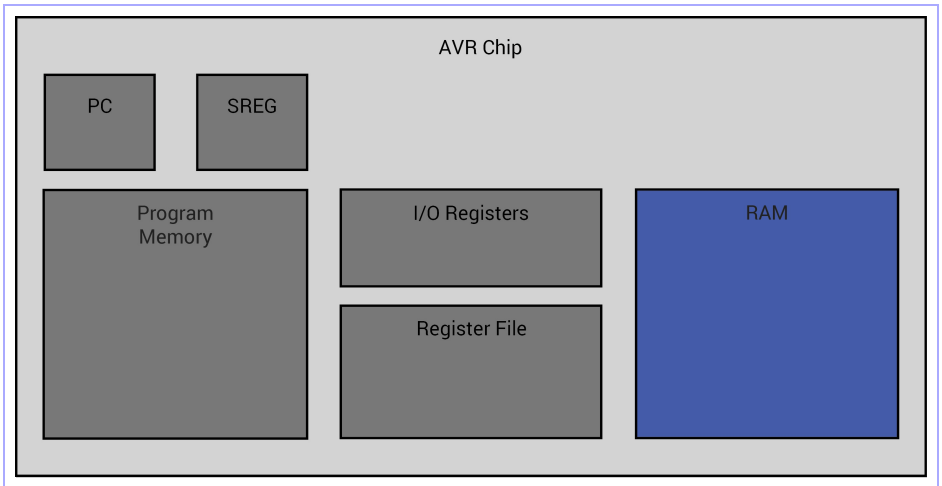

Figure 5.3.2.1: The many memories of an AVR computer.

When playing a game, say, with all many thousands of triangles on the screen at once, the computer must be keeping track of where all these triangles are so that it can constantly test, e.g. when two objects are colliding. This requires it to store the coordinates of every triangle, so

our computer has a memory of relatively large size in the form of is so-called **random access memory**, or RAM.

The RAM typically lives outside the core processing circuitry, and so whenever the computer wants to interact with RAM it needs to send some electrons on a long journey away and wait for them to return. This means that RAM is relatively slow (emphasis on the 'relatively'--modern RAM can transfer on the order of 10 billion bytes per second).

In an AVR computer, RAM will be a memory of width 8 (that is, it can store numbers 0-255) and have size 65536--that is, it can store 65536 of these numbers, with addresses 0-65535 (in practice, cheaper models of AVR chips will have less RAM available, but we'll proceed assuming we have the full-size RAM of size 65536). There is a small caveat that in many AVR computers, the first 96 slots in RAM are given special meaning and reserved, so you should not use any of these slots for storing values.

RAM is a **non-persistent** memory, meaning that when you turn off power to the computer, anything stored in RAM will be lost. For example, if you're playing a game, the game data(locations of enemies, how much energy you have left, etc.) are stored in RAM, and so will not re-appear if, as you're playing, someone pulls out the power supply to your computer (electrical cord, battery, or both, as applicable).

5.3.3: Program memory

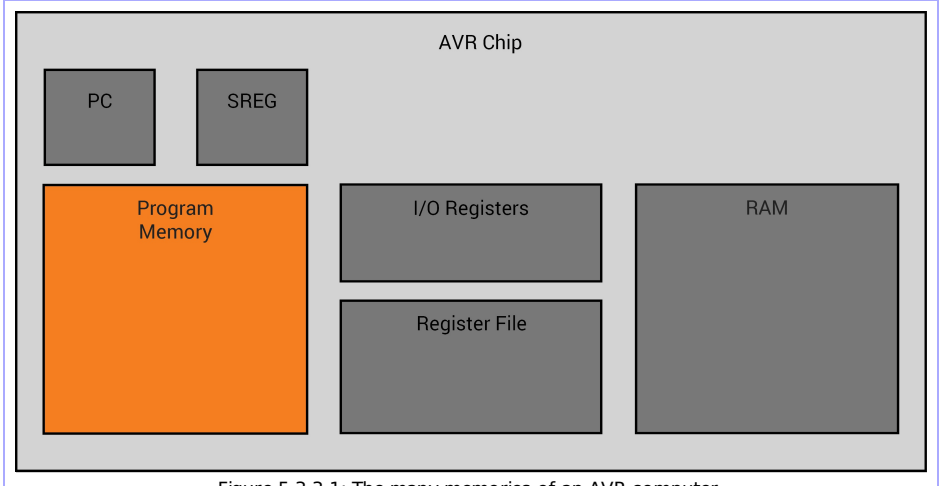

Figure 5.3.3.1: The many memories of an AVR computer.

In addition to RAM, we have another memory that stores the instructions the computer will execute.

This memory is called the **program memory**, abbreviated **PM**. This distinguishes it from the RAM, which stores data, and indeed is sometimes called the computer's 'data memory'. Unlike RAM, no program being executed can edit the PM (as this is where the program is stored--i.e. a program cannot edit itself on an AVR computer).

This memory will have width 16--that is, it can store numbers 0-65535--and size also 65536. Remember that instructions, while we'll name them with mnemonics like "add r0,r1", are actually just numbers which will be stored in the PM. Because the width of the PM is 16, this means all the numbers that represent our instructions will have to be numbers in the range 0-65535.

Program memory is a **persistent** memory, meaning that even if you turn off the power to the computer, what was stored in program memory will still be there next time you turn it on. (Persistence is also a property of the memory in which your files are stored on a normal desktop computer--you can expect that your files will still be there after you restart your computer because they are stored in a persistent memory.)

5.3.4: Register file

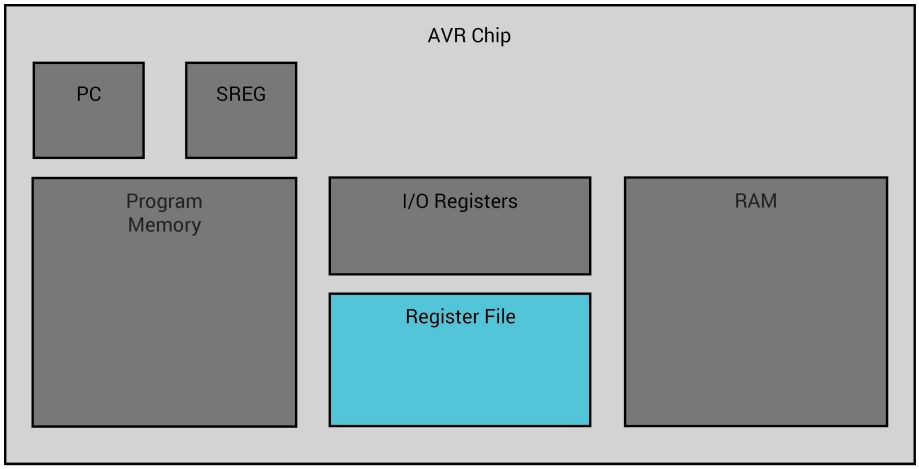

Figure 5.3.4.1: The many memories of an AVR computer.

Suppose we want to use our computer to add two numbers. If RAM is the only editable data memory we have, then, well, those two numbers have to come from somewhere, so we make two trips to RAM, one for each of the numbers we want to fetch. Then the processor adds them, but now it must put the result somewhere! If the only option is RAM, then it will have to make a third round-trip to RAM to store the sum.

Since RAM (especially on an AVR computer) can be slow, and we want add operations to be fast, the processor has another memory built into it, which is small but very fast, and which is what is actually used for arithmetic operations like adding. This memory has width 8 and size 32, and is called the **register file**.

Because we will be referring to the register file a lot, we have a special notation for it: The first slot, i.e. the slot at address 0, is called r0, then address 1 is called r1, proceeding through address 31 which is called r31. This naming scheme allows us to think of these slots as separate one-slot memories, or "registers". These particular registers are called the **general-purpose registers**.

5.3.5: Program counter

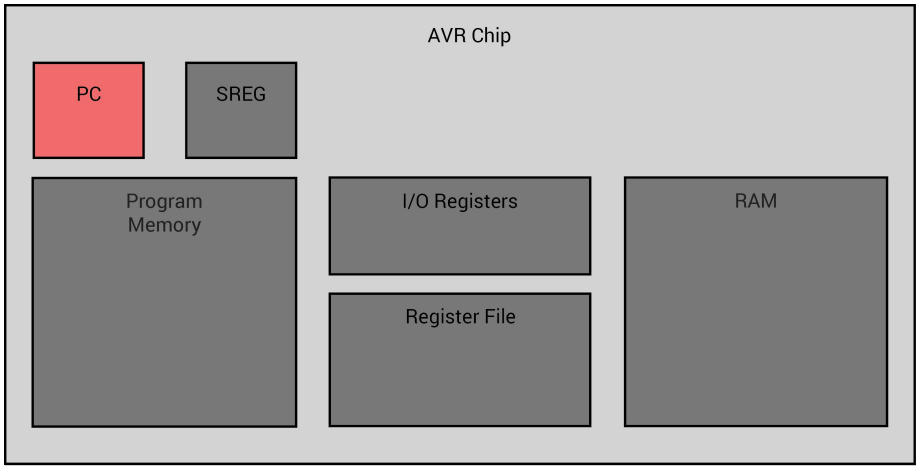

Figure 5.3.5.1: The many memories of an AVR computer.

We have seen that instructions are stored as numbers in the program memory.

The way the stored instructions actually get executed is that the computer has an additional register called the **program counter**--or 'pc'--that stores the address in program memory of the current instruction. Since it has to store the address of something in program memory, it has to store numbers 0-65535, i.e. it has width 16.

The program counter cannot be written to directly like a general-purpose register, but there are instructions that affect its value. In fact, most instructions will at least increment its value by 1, so once the instruction is finished, the program counter will have advanced to the next stored instruction. So the program counter allows us to have the usual sequential execution of instructions. But it also allows more complicated branching behavior: If we had the ability to, instead of incrementing the pc, give it a whole new value--effectively jumping to a different point in the code--subject to some conditions, then we would be able to do the same sort of branching we did with e.g. the if statement in Python.

5.3.6: Status register

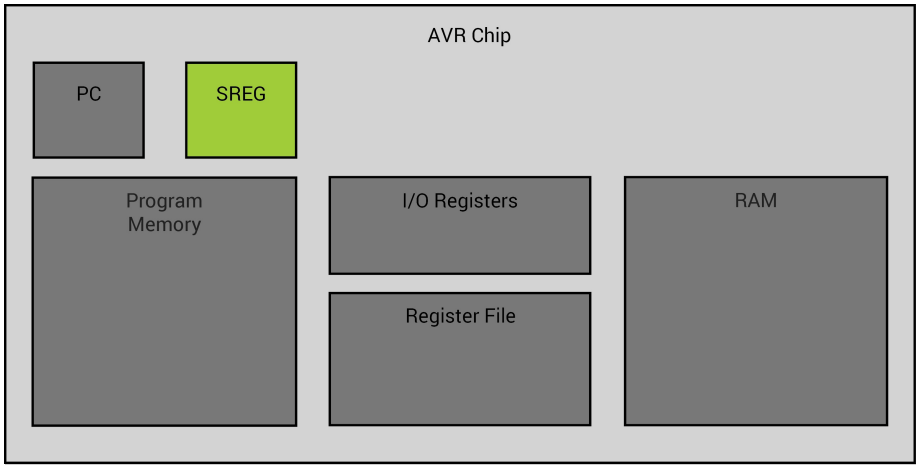

Figure 5.3.6.1: The many memories of an AVR computer.

Another memory in the architecture is the **status register**--or **sreg**. This is a register of width 8--that is, it is a number 0-255, but unlike the other registers, we think of it as comprised of 8 different numbers, each of which can be only 0 or 1--called **flags** (we will see in the next chapter how a number 0-255 can be thought of as 8 flags--for now we take it for granted). Storing a value of 1 in a flag is called **setting** the flag, and storing a value of 0 is called **clearing** it.

The basic idea is that while we can add/subtract/etc. numbers in the general-purpose registers and store the results in registers, there is further information about the result that cannot be communicated just by the result. For example, if we have the values 200 and 100 stored in general-purpose registers and we add them together, the result should be 300. But this result cannot be stored in a register! So what actually gets stored is the result modulo 256--in this case, 44. But the fact that we wrapped around like this is not lost--it is stored by setting one of the flags in the sreg. This situation, where the value we need to store is larger than the maximum value we are allowed to store, is called an **overflow**.

The 8 flags of the sreg are, in order, called I, T, H, S, V, N, Z, C, of which only three really come up in everyday use as important, and the other 5 of which we will not discuss here. These three relevant flags are:

- The C flag, which generally gets set if there was an addition or subtraction operation involved with the instruction and that instruction and this operation overflowed. In our above example, the addition would produce a result of 44 and would set the C flag.
- The Z flag, which generally gets set if the result of the previous operation was 0, and cleared otherwise. Since the above addition did not result in an answer of 0, the Z flag would be cleared.
- The N flag, which generally gets set if there was an addition or subtraction operation involved with the instruction and the result of the addition/subtraction was a negative number. The above addition would clear the N flag.

We use the word 'generally' in all of these descriptions, because there are exceptions, and when introducing each instruction, we'll explain precisely how it affects these three flags.

In the event that an instruction doesn't perform an arithmetic operation, say, the flags to not get changed, but rather remain what they were before the instruction was executed.

5.3.7: I/O registers

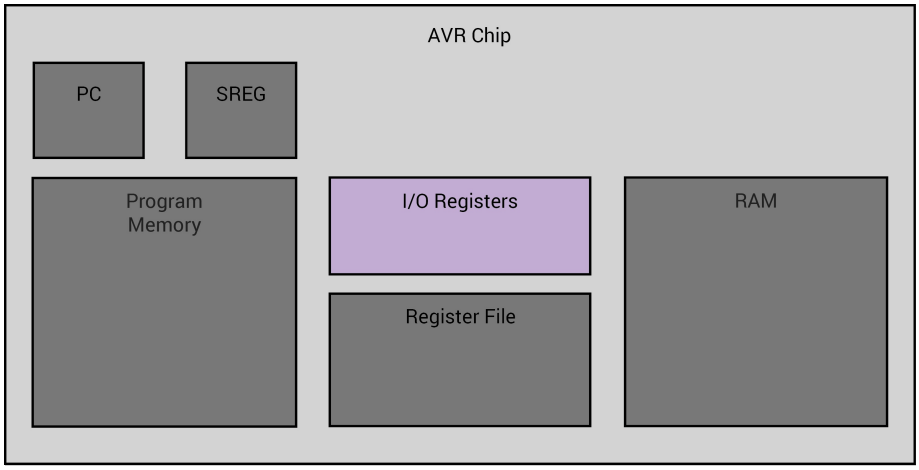

Figure 5.3.7.1: The many memories of an AVR computer.

There is one final piece to the architecture, which will not play a huge role in what follows, but which is very important for practical use: The **input/output registers**, or **I/O registers**. These are further special registers with miscellaneous purposes. There are 64 of them, and each has width 8. In fact, though they are called 'registers', their behavior is often different from that of normal registers: When you write a number to a normal register, that number gets stored somewhere, and will be returned when you subsequently read from that register.

One example of an I/O register is one that corresponds to certain pins on the computer chip. When you read from this register, the value you get will be determined by what current is being sent to certain pins. This allows you to e.g. see if a button has been pressed, since that button can be electrically connected to one of the pins on the computer, and reading from the I/O register will reveal whether there is current going to that pin, i.e. whether or not that button is down.

Likewise, writing to this register will allow you to cause current to flow from chosen pins on the chip, meaning you can turn on a light that is connected to one of the pins. Or perhaps, through more complicated use of the output pins, control a screen and display actual images.

We will not describe the behavior of all 64 I/O registers, but will give a more precise description of how to use three of them, namely I/O registers 16, 17, and 18, which are also known respectively as PIND, DDRD, and PORTD.

An AVR chip has 8 pins that, collectively, are called 'Port D'.

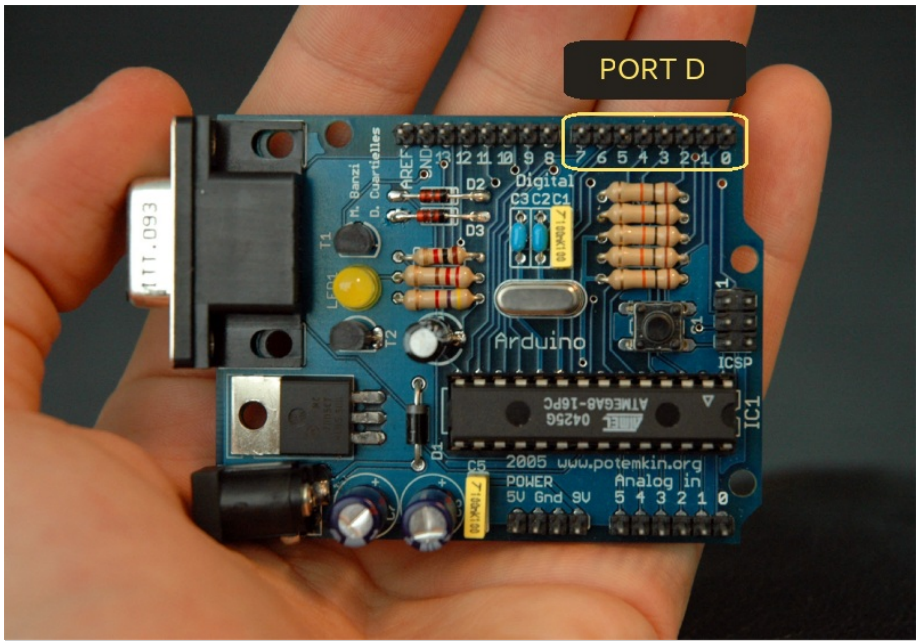

Figure 5.3.7.2: The port D pins on an Arduino computer.

These pins can, at any given time, be treated as input pins or as output pins, but not as both simultaneously. If the pins are input pins, then you can get the value being sent in on them by reading the PIND register--that is, reading I/O register number 16. If the pins are designated as output pins, then you can write a value to them by writing that value to PORTD, i.e. I/O register 18. To set the pins to be output pins, you set the corresponding flag in DDRD, and if you want that pin to be set as an input pin, you clear the corresponding flag. If a pin is an output pin, then reading PIND will produce results that need not correlate to what is happening on the pin at all, and likewise if the pin is an input pin, then writing to PORTD will have no effect.

For a simple example, if we want to make every other pin be sending electrons out, then we first need to make sure we can control what's coming out of all the pints by setting all the pins to be output pins. We do this by writing 255 (which is represented as 11111111 in binary) to I/O register 17, or DDRD. Then, now that we control the output of all the pins, we want all the pins to turn on, so we write 170 (represented by 10101010) to PORTD, causing every other pin to have electrons flowing out of it.

5.4: The Instructions

Summary: Now that we've described the AVR computer architecture--what are the components that make up an AVR computer--we move on to the second thing in the ISA: The instructions. Specifically, now that we know what memories the computer has available to use, we now learn all the ways that the computer can use them. As we said at the very start of this text, these will fundamentally be:

- Storage operations (putting values into memories or moving values between memories)
- Arithmetic operations (adding values in memory and putting the result in other memory)
- Branching operations (deciding based on some value in memory what instruction should be executed next--that is, what value to store in the PC)
- I/O operations (dealing with the I/O registers to interact with the external world)

We will start with a basic introduction to what the instructions look like and then describe all of the instructions that an AVR computer understands, classifying them into these four types.

Now that we've described the AVR architecture, we describe the instructions that comprise the AVR ISA, which will allow us to manipulate the components of the architecture--e.g. to store values in the various memories, add the contents of various registers, write values to I/O registers, etc.

5.4.1: A simple example

As we did with Python, we'll start with an initial example of a program that runs the 3x+1 procedure on the number 20, this time using AVR instructions:

```
ldi r16,20
ldi r17,1
out 26,255
out 27,r16
cp r16,r17
breq 11
mov r18,r16
andi r18,1
cp r18,r17
breq 2
shr r16,1
rjmp -10
mov r19,r16
add r19,r19
add r16,r19
inc r16
rjmp -15
```

This program is rather opaque, with few clues to what lines of code are doing what, but even without entirely understanding what is going on, a few things jump out at us:

- Every line is doing exactly one basic operation, in contrast to Python, where one line such as

  ```
  print(3*x+1)
  ```

  could perform two arithmetic operations and an output operation.
- Many operations such as 'add', are followed by either register names ('r16' or 'r17' or similar) or numbers.

This is a bit further removed than the corresponding Python program from the English language description of the algorithm, and there is little indication of what portions of the program perform what functions. Just like in Python, we could add comments by preceding them with a "#" symbol, we can also do this in ISA programs, but this time with the ";" symbol. So to improve readability, we will annotate this program with comments to give some idea of what's going on:

```
ldi r16,20          ; r16 = 20
ldi r17,1           ; r17 = 1
out 26,255
out 27,r16          ; Output r16 on port A
cp r16,r17
breq 11             ; Compare r16 and 1, and jump 12 instructions ahead
                    ; (to the end of the program) if they are equal
mov r18,r16
andi r18,1          ; Set r18 = r16 % 12
```

```
cp r18,r17
breq 2            ; Compare r18 to 1 and if equal, jump 3 instructions
                  ; ahead (to 'mov r19,r16')
shr r16,1         ; r18 = r18 / 2
rjmp -10          ; Jump back 9 instructions (to 'out 26,255')
mov r19,r16
add r19,r19
add r16,r19
inc r16           ; r16 = r16 * 3 + 1
rjmp -15          ; Jump backward 14 instructions (to 'out 26,255')
```

So in fact, all the high-level operations we used in our Python program are present in this program, but because each ISA instruction only performs a single basic operation, each of these high-level operations takes more ISA instructions to express. For instance, if we want to modify the value stored in r16 like according to the formula (not valid ISA code):

```
r16 = 3*r16+1
```

then we would use the following sequence of actual ISA operations (and since we haven't explained the meaning of the operations yet, we again sprinkle in some comments to explain what the instructions are doing):

```
mov r19,r16     ; r19 = r16
add r19,r19     ; r19 = r19 + r19
add r16,r19     ; r16 = r16 + r19
inc r16         ; r16 = r16 + 1
```

If you stare at this for a moment, you see that the result of this is that it indeed triples r16 and then adds 1 to it. But why did we have to take so many steps? Isn't this less efficient than the one-line Python version? The point is that the ISA operations are the only operations your computer can actually perform. When you write a more nicer-looking Python program that just does

```
x = 3*x+1
```

the computer has no idea what this means, since it isn't in the ISA. So instead a compiler has to figure out how to turn this into ISA operations, and it will come up with something like the sequence of ISA operations above. So one should not confuse brevity of code for savings in the amount of actual work that the computer will do in the end. Ultimately, to understand how efficient a piece of code in any language actually is, one has to understand what ISA operations will be actually run as a result of the code you've written.

5.4.2: Introduction to ISA instructions

So it first behooves us to understand better what ISA operations there actually are, and how to write them. In general, operations are written in the form

```
[operation name] [list of operands, separated by commas]
```

Operands are usually either constants or general purpose registers, but each operation comes with a specification of exactly what operands are allowed: For instance, to subtract two registers, we would use the sub operation. This operation specifies that its operands must be two general purpose registers, so we could legally do:

```
sub r16,r17
```

to perform the operation

```
r16 = r16 - r17
```

If, however, we wanted to subtract 1 from r16, we could not do

```
sub r16,1
```

since sub requires two registers as its operands! There is a separate instruction, called subi, which takes as its operands a register and a constant, and subtracts the constant from the register, so we could perform

```
r16 = r16 - 1
```

with the instruction:

```
subi r16,1
```

We have now almost described the syntax completely--every line is an operation followed by corresponding types of operands. All that remains is to explain what are the valid operations, and what kinds of operands each operation may have. Then, beyond syntax, we need to describe the semantics of each operation--what each operation actually does to the various memories in the computer (i.e. how each operation affects the architecture).

Before we actually present the operations, a word about the style of presentation: Because the format of an operation is so consistent, we can really just explain each operation by a table that shows:

- Operation name
- The permitted operands for that operation
- What the operation does with the operands
- What the operation does to the special registers pc and sreg.

We will resort to using exclusively this summary form eventually, but to start out with, we'll explain the operations in some more detail than this, providing this summary afterward. You will observe that in fact the summary contains ALL of the data found in the English explanation, and is more concise, and so would benefit from eventually getting comfortable programming by reference only to the summaries.

Now, on to the actual operations. Recall in Python we had essentially three functions each line could perform: Assignment, printing, and flow-control. Here, we will split the ISA instructions into four broad categories:

- **Storage**, which generalizes the aspect of Python assignment where we stored values in memory
- **Computation**, which generalizes the aspect of Python assignment where we computed the values to store using expressions
- **Input/Output**, which generalizes printing in Python
- **Flow-control**, which allows us to control which instruction was executed next.

5.4.3: Storage

We start with the analogues of Python's assignment lines (e.g. x = 2). There is an instruction called ldi--short for 'load immediate'--which we can use to specify a number to store into a general-purpose register. There are some limitations, however: We can only load numbers 0-255 (which makes sense, since the general-purpose registers have width 8), and we can only put these values into registers 16-31 (which is more arbitrary-sounding at this point and will be discussed when we talk about instruction encoding). So the syntax is:

```
ldi [general purpose register r16-r31],[number 0-255]
```

And the semantic meaning of this instruction is what it does to the architecture:

- The specified register gets filled with the specified value
- The pc increments (so that once this instruction is done we move on to the next instruction)
- The sreg is unchanged.

So, for example:

```
ldi r17,100
```

will put the value 100 into register 17, whereas

```
ldi r15,9
```

and

```
ldi r19,300
```

are both not legal ISA instructions--the first because we said ldi can only load into registers 16-31, and the second because we said ldi can only load values 0-255 into those registers.

But say we want to put a value into r10--what then? We cannot use ldi, but we can copy a value from one register into another using a second instruction--the "move" instruction, called mov. This instruction accepts any pair of general-purpose registers for its two operands, and will copy the value in the second operand register into the first. So, for example, if we did want to get the number 9 into register 10, we could do:

```
```

```
ldi r16,9
mov r10,r16
```

The first is a legal ldi instruction, which does the operation

```
r16 = 9
```

and the second is a legal mov instruction, which does

```
r10 = r16
```

And since we had previously set r16 to 9, this does the job of setting r10 to 9.

To summarize:

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials
ldi	Load immediate	reg 16-31	value 0-255	op1 = op2	pc: increment sreg: unchanged
mov	Move	reg	reg	op1 = op2	pc: increment sreg: unchanged

Note that these instructions allow you to write to the general-purpose registers only. In the architecture, the register file (which contains these registers) was the fast internal memory, but it was small--it only has 32 slots! What if we need to store more than 32 numbers in our program? But we had another component of the architecture--the RAM--which allowed us to store a very large number of values indeed--up to 65536 of them! The ldi and mov instructions don't write anything to RAM, so we need separate instructions for this job.

Enter the "load" and "store" instructions--ld and st. The instruction ld--or "load from RAM"--takes two operands. The first one is any general-purpose register, which will receive the value we load from RAM. The second is just the letter "X". So this instruction can be used like, for example:

```
ld r14, X
```

This will load a value from RAM into the register r14. But which value? In AVR, 'X' is a notation which stands for the value

```
r26 + 256 * r27
```

(Recall that in chapter 4 we had a notation for this value: r26:r27.)

Note that the biggest r26 and r27 can be is 255, so the biggest X can be is 255+256*255 = 65535. This is good, since that is the largest possible address in RAM. Thus if we want to read whatever is stored in the last slot in RAM and put that into r10, we need to make r26 and r27 both 255, and then use the ld instruction:

```
ldi r26, 255
ldi r27, 255
ld r10, X
```

This scheme allows us to read from any of the addresses 0-65535 in RAM. For example, to read from slot 10001, we can do

```
ldi r26, 17
ldi r27, 39
ld r10, X
```

since

```
17 + 256*39 = 10001
```

Complementary to this is the st--or 'store into RAM' instruction. The first operand for this is again always X, and the second is the register whose value we want to store at address X. Thus, to store the value 200 into RAM at address 999, we can do:

```
ldi r26, 231
ldi r27, 3
ldi r19, 200
```

```
st X, r19
```

The first two instructions will cause X to be 231 + 256*3 = 999. The third instruction will cause r19 to hold the value 200. Then the st instruction will store whatever's in r19 (in this case 200), into the slot with address X (in this case 999).

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials
ld	Load from RAM	reg	X	op1 = RAM[r26 + 256*r27]	pc: increment sreg: unchanged
st	Store in RAM	X	reg	op1 = RAM[r26 + 256*r27] = op2	pc: increment sreg: unchanged

One convention that we see in use in these instructions, and which will be common in the instructions that follow, is that when an instruction takes two operands, the first operand is the one that gets modified, and the second provides the source for this modification. One could read

```
ldi r19,129
```

as 'load into r19 the value 129'. Here, r19 is the value being modified, and 129 is the value we are using to modify it. Likewise,

```
st X, r5
```

could be read 'store into RAM address X the value in r5'. Once again, the first operand--X--describes what is being modified, and the second operand is where the modification comes from.

For this reason, the first operand is often referred to as the instruction's **destination**, and the second operand as the **source**.

5.4.4: Computation

Recall that in a Python assignment statement we could use as the right-hand side not only a number, as in

```
x = 9
```

but also any expression, e.g.:

```
x = 9
y = 7+x
```

ldi and mov allow us to accomplish the first thing for registers, but how does one accomplish something like the second, where we actually do some computation?

For starters, there are instructions for adding and subtracting registers, called add and sub respectively. These instructions take two operands--both any general-purpose register r0-r31. For example,

```
add r31, r0
sub r9, r8
```

are valid instructions. The add instruction will take the value in the second operand and add it to the value stored in the first operand, updating the first operand with this new value. So

```
add r31, r0
```

performs the operation

```
r31 = r31 + r0
```

Likewise, sub subtracts the second operand from the first.

So to mimic the behavior of the Python code above which adds 9 and 7, you could do, with r31 playing the role of x, and r30 the role of y: ldi r31,9 ldi r30,7 add r30,r31

In addition to updating the destination register (and incrementing the pc, as usual), these operations also update the sreg--specifically the Z, C, and N flags.

- Z flag: Most simply, the Z flag is set to 1 if the result of the operation is 0, and set to 0 otherwise. Note that it is possible for the result of an add operation to be 0: If we do 255+1 where the two values being added are 8-bit integers, then the result will be 256, which in binary is 100000000. The value that gets stored as the result of this, then, is the last 8 bits, or 00000000, representing an answer of 0.
- C flag: In the above case, what happened is that the addition carried over to the 9th bit. There is no 9th bit in the destination register, of course, so this 9th bit is stored in the C flag. So in the example of 255+1, the 9th bit is set to 1, i.e. the C flag will be set to 1.
- N flag: The N flag is set to 1 if the result of the operation would be a negative number if it is interpreted as an 8-bit two's complement number. In other words, if in the result, the most significant bit is set.

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials

`add`	Add	`reg`	`reg`	`op1 = op1 + op2`	`pc: increment`
					`N: set if MSB of result is set`
					`Z: set if result is 0`
					`C: set if op1 + op2 > 255`

`sub`	Subtract	`reg`	`reg`	`op1 = op1 - op2`	`pc: increment`
					`N: set if MSB of result is set`
					`Z: set if result is 0`
					`C: set if op1 - op2 > 255`

There are further instructions for computation beyond just addition and subtraction, but we will discuss only two further--increment and decrement:

As we saw when dealing with while loops, having an incrementing counter is a relatively common operation. In particular, to do the incrementing, we would use the line:

```
counter = counter + 1
```

Supposing you are doing this in the ISA and are using r31 for your counter, you could do this with the add instruction. However, the add instruction requires that the number you're adding to r31 be also put into a register. Therefore we would have to requisition another register for this purpose--say r30:

```
ldi r30,1
add r31,r30
```

Because incrementing a register by 1 is such a common operation, there is a special instruction that just does this without requiring the use of another register, called inc. It takes only one operand--any general-purpose register--and is used instead of the above like:

```
inc r31
```

which performs

```
r31 = r31 + 1
```

There is similarly a decrement instruction called dec for subtracting 1.

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials

`inc`	Increment register	`reg`	`[None]`	`op1 = op1 + 1`	`pc: increment`
					`N: set if MSB of result is set`
					`Z: set if result is 0`
					`C: set if op1 + 1 > 255`

`dec`	Decrement register	`reg`	`[None]`	`op1 = op1 - 1`	`pc: increment`
					`N: set if MSB of result is set`
					`Z: set if result is 0`
					`C: set if op1 - 1 > 255`

Similarly to add and sub for arithmetic operations, we have operations for performing the boolean operations we described at the end of chapter 4: and, or, xor, and not. These behave entirely analogously except that they do not modify the C flag and the minor note that the operation performing the XOR operations is called "eor" instead (the AVR ISA designers preferred "eor" to stand for "exclusive or" than "xor").

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials
					`pc: increment`

and	And	reg	reg	op1 = op1 & op2	N: set if MSB of result is set Z: set if result is 0 C: unchanged
or	Or	reg	reg	op1 = op1 | op2	pc: increment N: set if MSB of result is set Z: set if result is 0 C: unchanged
eor	Exclusive-or	reg	reg	op1 = op1 ^ op2	pc: increment N: set if MSB of result is set Z: set if result is 0 C: unchanged
not	Not	reg	[none]	op1 = ~op1	pc: increment N: set if MSB of result is set Z: set if result is 0 C: unchanged

Worth noting is that all these computation instructions act only on registers. This is standard--if we are going to be operating on some values, we need to store them in registers. So if ever we want add two numbers that are stored in RAM, we need to first load those values into registers using ld, then add them using the add instruction, and then finally store the sum back into RAM with an st instruction.

This means that doing arithmetic on numbers stored in RAM takes more operations, and hence more time. For this reason, we often avoid using RAM unless we definitely need to store more than 32 numbers. A good heuristic is that when programming in the ISA, we use RAM when we would have used an array in Python, and use registers when we would have used single variables.

5.4.5: Input/Output

We have seen how to write to the general-purpose registers and to RAM. Now we shall talk about writing to the I/O registers. Recall from our discussion in 5.3.7 that these are not actual registers. Writing a value to an I/O register usually doesn't store that value anywhere, but instead controls some behavior of the computer (e.g. what's being outputted on some of the pins). Likewise, reading a value from an I/O register doesn't retrieve a stored value, but instead gives you information about the computer (e.g. what's being inputted on certain pins).

The instructions for reading and writing I/O registers are relatively simple. To read an I/O register, we use the 'in' instruction, whose first operand is the register into which we will store the value we read, and whose second operand is the number of the I/O register we want to read. Recall there are 64 I/O registers, so this is any number 0-63, (though we are only interested in this text in I/O registers numbered 16, 17, and 18).

Thus,

```
in r14, 16
```

will read the 16th I/O register and store the result in register r14. Recall that I/O register 16 is the PIND register, which is comprised of bits that specify which of the port D pins on the chip have current flowing to them (except for the pins that are currently functioning as output pins, whose corresponding bits will be zero regardless of what is happening).

Recall also that I/O register 18 is PORTD, which, when written to, sends current on the corresponding pins of port D (or at least, those designated as output pins). Reading this I/O register is undefined--that is, the processor provides no guarantees about what will result from the instruction

```
in r14, 18
```

This is not an illegal instruction, but the value that this will place in r14 is simply not specified. It might be always 0, it might be something random, it might be related to the current input values on the pins, say, or it might be determined by something else entirely. This depends on the details of the implementation of the computer, and is not a part of its interface, and hence should never be relied on in a program.

On the other hand, writes to I/O register 18 do mean something--this is how we communicate with the outside world! The instruction that writes I/O registers is called 'out'. Its first operand is the number of the I/O register that we are writing to, and its second operand is the register whose value is to be written to that I/O register. In particular, this means that to write a value to an I/O register, we first have to have that value in a general purpose register.

Thus to write the number 100 to I/O register 18, we do:

```
ldi r30, 100
out 18, r30
```

Recall according to the architecture, this may not actually affect the output on all the port D pins, but only those that are set to output mode. If we want to set all the pins to output mode, we have to write

a number that is all 1s in binary--i.e. 255--to the DDRD I/O register, or I/O register 17. Now that we have the out instruction, we can do this:

```
ldi r30, 255
out 26, r30
ldi r30, 100
out 27, r30
```

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials
in	Read I/O register	reg	value 0-63	Read op2 into op1	pc: increment sreg: unchanged
out	Write to I/O register	value 0-63	reg	Write op2 to op1	pc: increment sreg: unchanged

5.4.6: Flow-control

So far we have seen how the computer can perform certain arithmetic and memory operations, but currently all we can do is specify a sequence of these operations that will get executed in order from start to end without divergence. But as we have learned, a lot of the interesting behavior of a computer comes from the ability to run different code depending on what input we get. So now the fun begins: We'll see how to execute different code depending on certain conditions.

To jump to different points in the code unconditionally, we use the "relative jump", or rjmp instruction. The rjmp operation takes one operand--a number between -2048 and 2047. This instruction simply adds its operand to the current pc. One important caveat, however, is that just like all other instructions, the pc is incremented once the instruction is finished. So, for example,

```
rjmp 5
```

will add 5 to the pc. But if we want to jump ahead by precisely 5 instructions, this will not do: It will add 5 to the pc, but then the pc will be incremented (just like after any other instruction), so this will actually jump ahead 6 instructions. Instead, to advance 5 instructions use rjmp 4 to add 4 to the pc, and then the final increment brings us up to 5.

So, for example,

```
rjmp 0
```

will add 0 to the pc, leaving it unchanged. Then once this is done, the pc will be incremented. So this instruction effectively does nothing. On the other hand,

```
rjmp -1
```

will subtract 1 from the pc. Then the pc will get incremented. So after this instruction is done, the pc will still be the address of this rjmp instruction, and so it will get executed as the next instruction as well. So this instruction actually jumps back to itself forever, causing an infinite loop.

Play around a little with the following example to get a feel for this behavior:

```
ldi r30,0
inc r30
add r30, r30
rjmp -2
```

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials
rjmp	Relative jump	value -2048-2047	none	none	pc = pc + op1 pc increment sreg: unchanged

Now to actually see the 'conditional jump' functionality analogous to Python's if statements, we introduce instructions that perform relative jumps exactly like rjmp, except only under certain conditions based on the flags in the sreg.

Instructions that perform conditional jumps are called **branch** instructions, since they split the program into two possible branches--one where the jump is taken and one where it is not. These instructions are like rjmp in that they take a single operand--though this time a number -64 to 63. There are branch instructions corresponding to 'jump if Z is set' (called breq), 'jump if Z is clear' (called brne), 'jump if C is set' (called brsh), and 'jump if C is clear' (called brlo).

Instruction	Description	Operand 1	Operand	Effect on	Effect on specials

			2	operands	
breq	Branch if equal	value -64-63	none	none	If Z set, pc = pc + op1 pc increment sreg: unchanged
brne	Branch if not equal	value -64-63	none	none	If Z clear, pc = pc + op1 pc increment sreg: unchanged
brsh	Branch if same or higher	value -64-63	none	none	If C set, pc = pc + op1 pc increment sreg: unchanged
brlo	Branch if lower	value -64-63	none	none	If C clear, pc = pc + op1 pc increment sreg: unchanged

This is nice, but it is not literally the 'if statement' functionality we are used to. That is, what we do not have is an instruction that compares two numbers and performs a specified relative jump if the two are equal.

However, we do have an instruction that will perform a relative jump if the Z flag is set. And we have another instruction--called cp--which takes two registers as its operands and sets the Z flag if the two numbers are equal, and clears it otherwise. Thus 'jump ahead 10 instructions if r10 and r11 are equal' can be accomplished by:

```
cp r10, r11
breq 9
```

And 'jump ahead 10 instructions if r10 and r11 are not equal' can be done with

```
cp r10, r11
brne 9
```

cp also affects C: It sets the C flag if the second value is bigger than the first. So

```
cp r10, r11
```

will set the C flag if r10 < r11, so we can get the functionality of 'jump ahead 20 instructions if than r10 is smaller than r11' by using the branch instruction that jumps if C is set--namely 'branch if lower', i.e. brlo:

```
cp r10, r11
brlo 20
```

Finally, we can get 'jump ahead 5 instructions if r10 is greater than or equal to r11', since in this case

```
cp r10, r11
```

will clear the C flag, and we have an instruction--'branch if same or higher', i.e. brsh--which jumps if C is clear.

Note that we haven't mentioned the case of 'jump 10 instructions ahead if r10 is greater than r11', but this is the same as 'r11 is smaller than r10', so we would do this test with brlo like:

```
cp r11, r10
brlo 10
```

The final case that we have not explicitly addressed of 'jump if r10 is less than or equal to r11', is an exercise for the reader.

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials
cp	Compare two registers	reg	reg	none	pc: increment N: set if op1 - op2 < 0 Z: set if op1 - op2 is 0 C: set if op1 - op2 > 255

So appropriate combinations of cp and various branch instructions allow us to perform conditional jumps corresponding to all of our comparison operators:

Comparison	ISA ops to jump 50 instructions if the comparison holds
r30 is equal to r31	cp r30, r31

	`breq 50`
r30 is not equal to r31	`cp r30, r31` `brne 50`
r30 is less than r31	`cp r30, r31` `brlo 50`
r30 is greater than r31	`cp r31, r30` `brlo 50`
r30 is greater than or equal to r31	`cp r30, r31` `brsh 50`
r30 is less than or equal to r31	`cp r31, r30` `brsh 50`

5.4.7: Debrief

These instructions are not all that there is to the AVR ISA, but they comprise the basic examples of the four kinds of operations it can perform. Just as we could do these four kinds of operations in Python, we can now do them in the ISA. There are some differences yet:

- We have not talked about using strings.
- Instead of printing things to the screen, we are outputting them on pins
- In Python, when we dealt with numbers, these could be arbitrarily large or decimal numbers or anything else. In the ISA, registers are limited to integers 0-255.
- Python had several advanced features, such as arrays

Now that we have an understanding of how instructions in the ISA work, we'll go into some advanced features that will help bring us the rest of the way to understanding how Python programs can be completely turned into ISA programs.

5.5: Advanced ISA features

The above instructions are theoretically sufficient to do most things you would want to do, much like the subset of Python we learned in chapter 2 was enough to get by, at least in principle. But just in chapter 3 we learned that Python had some more advanced features that made our lives much easier, in this section we will see further instructions from the ISA that can simplify our lives when coding in the ISA.

5.5.1: Immediate operands

Summary: Most of the computation instructions we learned about previously combined two registers. Sometimes we just want to e.g. subtract 2 from a number, but to do this we still had to load 2 into a register to use the sub instruction. AVR, it turns out, also has instructions that allow us to e.g. perform the operation "subtract a given value from a specified register" without that value coming from a register.

When discussing computation instructions, we mentioned the restriction that if we wanted to subtract a number from a register, then that number has to also be in a register because the only subtraction instruction we have requires both its operands to be registers. So for example, to subtract 5 from r30, we had to do:

```
ldi r31, 5
sub r30, r31
```

To simplify this, AVR has an instruction that subtracts a given number from a given register, called subi (for "subtract immediate"). Its first operand is a register and the second operand is a value to subtract from that register. Like ldi, however, the register can only be one of r16-r31, and the value can only be in the range 0-255.

Just as there is a version of sub with immediate operands, there are also versions of "and" and "or" that likewise take an immediate operand, called "andi" and "ori" respectively (note that there is no "eori" operation though).

And much like sub had a related instruction--cp--for comparing registers, subi has a related instruction called cpi for comparing a register and a number. cpi has the same operands and effects as subi, except that like cp, cpi does not modify any register values.

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials

`subi`	Subtract immediate	`reg 16-31`	`value 0-255`	`op1 = op1 - op2`	`pc: increment`
					`N: set if MSB of result is set`
					`Z: set if result is 0`
					`C: set if op1 - op2 > 255`

`cpi`	Compare with immediate	`reg 16-31`	`value 0-255`	`none`	`pc: increment`
					`N: set if MSB of result is set`
					`Z: set if result is 0`
					`C: set if op1 - op2 > 255`

	Subtract				`pc: increment` `N: set if MSB of result is set`

andi	immediate	reg 16-31	value 0-255	op1 = op1 & op2	Z: set if result is 0 C: unchanged
ori	Subtract immediate	reg 16-31	value 0-255	op1 = op1 | op2	pc: increment N: set if MSB of result is set Z: set if result is 0 C: unchanged

5.5.2: Add-with-carry

Summary: In Python we could operate on numbers as large as we want. So far, we have only see the ability to add registers, which can only store values 0-255. To operate on larger numbers, we can use a scheme to use multiple registers to all represent one big number. But then our basic add instruction isn't enough to add such numbers, so here we introduce instructions that will help us in this endeavour.

In all the instructions we have described so far, we have only been able to operate on numbers in the range 0-255. This may seem maddeningly limiting--we couldn't even run our 3x+1 example on the small input of 27 because the numbers involved get into the thousands.

However, recall that our registers are just 8-bit memories. So if we want to store a 16-bit number, we can take any two registers and think of them as comprising one larger number by concatenation. For example, we could take the registers r16 and r19 and use them together to store one 16-bit number, denoted as r16:r19, and which represents the number

```
r19 + 256*r16
```

This is a lot like the "compound register" X that we learned about earlier. But unlike X, which is a notation built into the AVR ISA, this compound number that we've contrived doesn't have any supported notation. It is just two registers, which we can manipulate separately using all the usual instructions, but which we are thinking of as two parts of one big number.

Now how to we operate on this number? For the simplest example, how do we add 1 to it? Suppose r19 and r16 are both 0. These represent the number

```
0:0 = 0 + 256*0 = 0
```

If we simply add 1 to both registers, then r16 and r19 will both be 1, so will represent the number

```
1:1 = 1 + 256*1 = 257
```

That's not right. Clearly what we need to do is to add 1 to only r19. Then we would get the number

```
0:1 = 1 + 256*0 = 1
```

Excellent. So is the procedure for incrementing our big number just to add 1 to r19 and leave r16 alone? Well, not entirely. The problem happens when r19 = 255, and say r16 = 0. Then if we add 1 to r19, this will set r19 to be 0. But this is no good! This means both r19 and r16 will be 0, so our big number will be 0, whereas what we wanted was 256. That is, we wanted r16 to be 1 and r19 to be 0. So r19 actually got the correct value, but in response to the overflow, we should have also incremented r16.

The trick is to notice that when we added 1 to r19 above, and it overflowed back to 0, this caused the C flag to be set in the sreg. So what we want to do is not just

```
r19 = r19 + 1
```

but rather

```
r19 = r19 + 1
r16 = r16 + C
```

Happily, there is an instruction which does this and more--adc. It takes the same register operands as add and adds the two registers together, much as add did, but also adds C on to the result.

Thus we can use it to implement our increment operation by doing a normal add instruction to increment r19, and then using adc to add 0 to r16. This will leave r16 unchanged unless C is set, in which case it will add 1 to r16--exactly what we wanted:

```
ldi r30, 1
ldi r31, 0
add r16, r30
adc r19, r31
```

This, finally, is ISA code that increments our combined number. We will later write code to do more interesting operations to such combined numbers, like adding two such numbers.

Note also that if this number has its largest possible value, where r19 = 255 and r16 = 255 (so the number is 65535), then incrementing it using this procedure first increments r19, making it 0 and setting C. But then r16 gets incremented since C was set, and because it is also 255, it also gets set to 0. So our big combined number exhibits the same 'wraparound' behavior at its maximum value of 65535.

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials
adc	Add with carry	reg	reg	op1 = op1 + op2 + C	pc: increment N: set if MSB of result is set Z: set if result is 0 C: set if op1 + op2 + C > 255
sbc	Subtract with carry	reg	reg	op1 = op1 - op2 - C	pc: increment N: set if MSB of result is set Z: set if result is 0 C: set if op1 - op2 > 255

5.5.3: Advanced addressing modes

Summary: AVR has instructions for easily reading or writing a set of successive of addresses in RAM.

Earlier we discussed instructions for reading from and storing to address X. But suppose we have 20 values in RAM, starting at address 500, and we want to do something to each of them.

```
500 = 244 + 256*1
```

So to read the first slot in this sequence, we set r26 = 244 and r27 = 1:

```
ldi r26, 244
ldi r27, 1
ld r0, X
...[Do stuff with the value in r0 that we just read from RAM]
```

The next slot will be at 501, so since

```
501 = 245 + 256*1
```

we need can just increment r26 and leave r27 alone:

```
ldi r26, 244
ldi r27, 1
ld r0, X
...[Do stuff with the value in r0 that we just read from RAM slot 500]
inc r26
ld r0, X
...[Do stuff with the value in r0 that we just read from RAM slot 501]
```

We can do this several more times without issue, but around slot 511 we will start to have a problem, since

```
511 = 255 + 256*1
```

So r26 will be 255, and incrementing it will cause an overflow, making it 0. Then we'll be reading from slot 0 + 256*1 = 256, rather than slot 512.

So this is exactly the problem we faced before of incrementing a combined number. So to fix this, we could also add C to r27 as we discussed in the previous section. However, it turns out that this operation of reading a sequence of values in memory is common enough that there is a modified version of the 'read from X' instruction which again reads from X, but then increments X correctly after it does so.

This instruction is again called ld, and its first operand is again a register in which to store the value read from RAM, but for its second operand we write X+, telling it to increment X after reading the value.

With this instruction, our code becomes:

```
ldi r26, 244
ldi r27, 1
ld r0, X+
...[Do stuff with the value in r0 that we just read from RAM slot 500]
```

```
ld r0, X+
...[Do stuff with the value in r0 that we just read from RAM slot 501]
ld r0, X+
...[Do stuff with the value in r0 that we just read from RAM slot 502]
...
```

and the ld __, X+ instruction takes care of incrementing X carefully, exactly as we learned to do by hand in the previous section.

There is an analogous 'st X+, register' instruction that stores to address X and then increments X afterward. Further, because going backwards through memory is also a reasonably common operation, there are also instructions for decrementing X before reading or storing. These are all described in the table below:

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials
`ld`	Load from RAM and increment address	`reg`	`X+`	`op1 = RAM[r26 + 256*r27]` `X = X + 1`	`pc: increment` `sreg: unchanged`
`st`	Store to RAM and increment address	`X+`	`reg`	`RAM[r26 + 256*r27] = op2` `X = X + 1`	`pc: increment` `sreg: unchanged`
`ld`	Decrement address and load from RAM	`reg`	`-X`	`X = X - 1` `op1 = RAM[r26 + 256*r27]`	`pc: increment` `sreg: unchanged`
`st`	Decrement address and store to RAM	`-X`	`reg`	`X = X - 1` `RAM[r26 + 256*r27] = op2`	`pc: increment` `sreg: unchanged`

5.5.4: The stack

Summary: AVR also provides a mechanism for quickly saving/restoring the values of registers by means of the stack. The stack is a region in RAM that can be accessed by special instructions that make it convenient to store a bunch of register values there and later restore those values to the actual registers.

Definitions: push, pop, stack pointer

This next ISA feature may seem oddly specific and unmotivated, but it exists in almost every ISA you'll come across. The basic reason for this is that it is absolutely necessary for functions to work. It can also be used more simply as a convenient way to stash some values in RAM for quick and easy retrieval later. The feature that enables all this and more is called the "stack".

The name comes from the reasonably obvious metaphor: A stack of plates, say. If I have a stack of plates, there are only really two things I can do: I can place another plate at the top, or I can pull a plate off the top. I cannot pull a plate out of the middle, nor can I insert a plate into the middle of the stack.

We will refer to the operation of putting a plate at the top as "pushing" a plate onto the stack, and of taking a plate off the top as "popping" a plate off the stack.

In the AVR ISA, we have likewise two instructions called push and pop, both of which take just any one register for an operand. When you push a register, e.g.

```
push r0
```

This will save the value in r0 somewhere by "pushing it onto the stack". If you then do

```
pop r1
```

this will take that stored value off of the stack and put it in r1.

If we instead push multiple registers, like:

```
push r1
push r1
push r0
push r5
```

then this saves the values of those registers in that order. So if at this point we imagine r0 was 34, r1 was 99, and r5 was 18, then the stack looks like:

Top	18
	34

	99
Bottom	99

The pop instruction will take one value off the top and place it into its operand register. So if we do

```
pop r10
```

then this will take the top value--18--off the top of the stack and store it in r10. The stack will now look like:

Top	34
	99
Bottom	99

If we then do:

```
pop r11
pop r12
pop r13
```

Then r11 will get the value 34, r12 will get 99, and r13 will get the value 99, and the stack will be empty.

How it actually works:

The stack of plates metaphor allows us to think effectively about what the push and pop instructions do, but what is the architectural reality behind this conceptual idea? After all, we say that things we push get stored on the stack, but there wasn't a "stack" memory in the architecture. So where does push actually store things?

The answer, as we mentioned very briefly, is in RAM. But where in RAM?

There is a 16-bit register called the **stack pointer**, or often called **sp**, which stores the address in RAM of the top of the stack. When we push a register, its value gets written to RAM at the address sp, and then sp gets decremented by 1 (you read that right--the top of the stack moves downward in RAM for some reason). When you pop a register, the stack pointer gets incremented by 1 and then the register receives the value in RAM at the address sp.

Let us take the above example again, with r0 = 34, r1 = 99, r5 = 18, and let us say that at the beginning, sp = 5000. So before we do anything, the stack looks like:

Address	**Value**
sp = 5000	??

(There is some value at this address, but we don't care what it is, so we show ?? in its place rather than a random number).

Then we push r1. This stores the value of r1 (99) in RAM at address sp (5000), and then decrements sp, so sp is now 499 and the stack now looks like:

Address	**Value**
sp = 4999	??
5000	99

(After push r1)

Subsequently pushing r1 again, then r0, then r5 walk the stack through states:

Address	**Value**
sp = 4998	??
4999	99
5000	99

(After push r1)

Address	**Value**
sp = 4997	??
4998	34
4999	99

5000	99

(After push r0)

Address	Value
sp = 4996	??
4997	18
4998	34
4999	99
5000	99

(After push r5)

Then we pop r11. This first decrements the sp, so it is now 4997, and then reads the value there (18) into r11. So afterward the stack looks like:

Address	Value
sp = 4997	??
4998	34
4999	99
5000	99

(After pop r11)

And r11 contains the value 18. We note that for clarity, we have written the value at address 4997 as ??, but it is actually still 18. Just that this is no longer accessible to pop instructions (the next one will get the value 34) and it will be overwritten by a push instruction, so it no longer matters what it is specifically.

What is the stack pointer?

So push and pop behave as in the mental model, but actually using a region in RAM to store the values. But what, in turn, is this mysterious stack pointer? We didn't see "sp" as a separate memory in the architecture either. It turns out that the sp is actually stored in the I/O register file. But sp was 16 bits, and we said the I/O register file has width 8, so in fact sp is a combined number from two I/O registers (much like X was made up of two general-purpose registers). Specifically, sp is the combination of I/O registers 61 and 62 as IO[62]:IO[61]. That is:

```
sp = IO[61] + 256*IO[62]
```

In particular, if we are going to use push and pop instructions, we have to first set the sp to some sensible place in RAM where it doesn't get in the way of anything else we want to do with RAM. For example, maybe we decide to set it to 5000 as in the above example. Then we compute:

```
5000 = 136 + 12*256
```

we need to set I/O register 61 to store value 136 and I/O register 62 to have value 12. Remember that we write the I/O registers using the "out" instruction, so we do:

```
ldi r31, 136
out 61, r31
ldi r31, 12
out 62, r31
```

Now the stack is set up, and the first push will write to address 5000 in RAM, the next push will write to address 4999, etc.

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials
`push`	Push register to stack	`reg`	`none`	`RAM[sp] = op1` `sp = sp - 1`	`pc: increment` `sreg: unchanged`
`pop`	Pop register off stack	`reg`	`none`	`sp = sp + 1` `op1 = RAM[sp]`	`pc: increment` `sreg: unchanged`

5.5.5: Functions

Summary: The construction of a function in Python was a useful way to reuse a chunk of code in several places without just copying it. The AVR ISA has support for this behavior in the form of its rcall and ret instructions.

Definitions: callee-saved, caller-saved

In Python, if we ever wrote a piece of code that we want to reuse elsewhere, we could package it up as a function and then call it from multiple places elsewhere in our code. In the ISA, we have already seen code that would be useful to treat this way: Our code for outputting a number (say, the contents of r30):

```
ldi 255, r31
out 26, r31
out 27, r30
```

If we end up wanting to output more than one thing in our program, we'll have this snippet of code appearing everywhere, so if we could reuse it more efficiently that would be excellent. However, it turns out that having a chunk of code that we can reuse in this way is tricky: The main challenge comes from the fact that we have to be able to jump to this chunk of code from anywhere, run the code, and then have the code jump back to wherever it was called from.

This should sound like an decent use-case for the stack: We are going merrily along through our program, incrementing the pc, business as usual. At some point we want to output a number using the above code, which already exists at some point in our program. The basic procedure is that we quickly stash the current pc on the stack, jump to that code to have it output what we want, and then pop the pc back off the stack to return to where we started.

Of course, the push and pop instructions only take general-purpose registers as operands, so we cannot do this with our existing instructions. So the AVR ISA comes with two further instructions--called rcall and ret--which accomplish this. rcall is exactly like rjmp: It takes a single operand, which is a number in the range -2048 to 2047. It adds this to the pc and then it increments the pc. However, unlike rjmp, before it modifies the pc, it pushes onto the stack the value pc + 1, which is the address of the next instruction that should get executed when we finish with the function. So whereas rjmp irrevocably dumps us somewhere else in the program, rcall stashes in the stack the address that will bring us back.

So we've rcalled some other code. That code executes, and now we want to resume normal execution back where rcall left off. To do this, all we have to do is pop the pc off the stack, as then the pc will have its old value. But the pop instruction doesn't allow the pc as an operand, so there is instead an instruction called ret. ret takes no operands and very simply pops the pc off the stack, exactly as we wanted.

Instruction	Description	Operand 1	Operand 2	Effect on operands	Effect on specials

`rcall`	Call relative address	`value -2048 to 2047`	`none`	`push pc+1`	`pc = pc + op1`
					`increment pc`
					`sreg: unchanged`

`ret`	Return from most recent call	`none`	`none`		`pop pc` `sreg: unchanged`

For example, we can now try to write code that reuses our above output code. Let us consider writing code that first outputs the number 13, then multiplies it by 8 and outputs that. We can start with the outline:

```
ldi r30, 13
[call to output code to output r30]
add r30, r30
add r30, r30
add r30, r30
[call to output code to output r30]
```

We need to put our code for outputting stuff somewhere, so maybe we'll put it at the beginning:

```
ldi 255, r31
out 26, r31
out 27, r30
ldi r30, 13
[call to output code to output r30]
add r30, r30
add r30, r30
add r30, r30
[call to output code to output r30]
```

But now when we first run this program, our output code will get run first, which is not what we wanted. We could put it at the end, but then that just moves the problem to the end of the program, where the output code will be run an extra time once everything else is finished. To resolve this, we make the first instruction an rjmp which skips over the output code to the first line of our computation code:

```
rjmp 3
ldi 255, r31
out 26, r31
out 27, r30
```

```
ldi r30, 13
[call to output code to output r30]
add r30, r30
add r30, r30
add r30, r30
[call to output code to output r30]
```

Now we can fill in the call lines: the first one is making a call 4 instructions back, so it will be rcall -5, and the second call needs to call the code 8 instructions back, so it will be rcall -9:

```
rjmp 3
ldi 255, r31
out 26, r31
out 27, r30
ldi r30, 13
rcall -5
add r30, r30
add r30, r30
add r30, r30
rcall -9
```

But this is no good still--once we rcall -5, we jump back to the line

```
ldi 255, r31
```

and then the program continues execution from there--basically starting the program over. Rather, we want our output code to use the ret instruction once it finishes:

```
rjmp 4
ldi 255, r31
out 26, r31
out 27, r30
ret
ldi r30, 13
rcall -6
add r30, r30
add r30, r30
add r30, r30
rcall -10
```

(Note we've had to shift all the rcall and rjmp operands.)

This is now great, but there is another issue: rcall is using the stack, but we don't actually know at this point that sp is an address that makes sense (remember that RAM addresses 0-92 are reserved, so if the sp defaults to address 0, then we may get unexpected behavior), since we never set it up. So we do this as the first thing:

```
ldi r31, 136
out 61, r31
ldi r31, 12
out 62, r31
rjmp 4
ldi 255, r31
out 26, r31
out 27, r30
ret
ldi r30, 13
rcall -6
add r30, r30
add r30, r30
add r30, r30
rcall -10
```

And now, finally, we've got things set up so that we can reuse the outputting code as we'd hoped.

There were quite a few pitfalls in getting this set up, so we record for future reference our code-reuse checklist:

- Ensure that the sp is set to something sensible
- Place the reuseable code somewhere where it won't get executed without rcalling it, e.g. by placing it at the beginning and jumping over it.
- Ensure the reuseable code ends with a ret instruction
- Call the code using rcall with an operand that shifts you to the start of the reuseable code, being careful to update these offsets if you change any intervening code.

Caller vs. callee-saved registers:

There is one further issue that doesn't actually cause a problem in the above example, but can easily become a problem in more complicated examples, namely that the reusable code modifies the r31 register. Specifically, it fills it with a value that it uses for its own purposes, overwriting whatever value was in it before.

To see this in action, suppose instead of multiplying r30 by 8, we wanted to add the number 41 to it two times, output r30, and then add 41 to it three more times and output the result of that:

```
ldi r31, 136
out 61, r31
ldi r31, 12
out 62, r31
rjmp 4
ldi 255, r31
out 26, r31
out 27, r30
ret
ldi r30, 13
rcall -6
ldi r31, 41
add r30, r31
add r30, r31
rcall -10
add r30, r31
add r30, r31
add r30, r31
rcall -14
```

In this example, we use r31 to hold our value of 41 that we are adding to r30. But after the line

```
rcall -10
```

we continue adding r31 to r30, expecting that r31 still has the value 41, oblivious to the fact that our output code overwrites r31 with the value 255!

There are two possible remedies: We can make our output code save the previous value of r31 using the stack, and then restore it before it returns:

```
ldi r31, 136
out 61, r31
ldi r31, 12
out 62, r31
rjmp 6
push r31
ldi 255, r31
out 26, r31
out 27, r30
pop r31
ret
ldi r30, 13
rcall -8
ldi r31, 41
add r30, r31
add r30, r31
rcall -12
add r30, r31
add r30, r31
add r30, r31
rcall -16
```

(Note again the need to adjust the operands to the jumps and calls.)

In this case, r31 is being saved by the function we're calling, so r31 is referred to in this case as a **callee-saved register**: Anyone who calls this function is guaranteed that r31 will have the same value when the function returns as it had when it started.

The other option would be that before we call the code, we note that it will trash r31 and so we save it onto the stack before calling and then restore it after the code returns:

```
ldi r31, 136
out 61, r31
ldi r31, 12
out 62, r31
rjmp 4
ldi 255, r31
out 26, r31
out 27, r30
ret
ldi r30, 13
rcall -6
ldi r31, 41
add r30, r31
add r30, r31
push r31
rcall -11
pop r31
add r30, r31
add r30, r31
add r30, r31
rcall -16
```

In this case, the output code will happily trash r31, and it is the responsibility of any code that calls it to save its value. In this case, the register is known as a **caller-saved register**: Anyone who calls the code must save r31 if they want its value preserved.

Aside: Abusing rcall

You'll have noted that most of our operations thus far only act on general-purpose registers. However if we really wanted, we could make them act on sreg or sp by reading these from the appropriate I/O registers into the general-purpose registers, computing with these general-purpose registers, and then writing the results back to the relevant I/O registers. However none of our instructions can take pc as an operand, and pc doesn't actually live in the I/O register file either.

So if we're merrily going through a program and for whatever reason we want to get the current pc in a register, there is not an instruction we can use that does this. However, we can accomplish it in several instructions with a slight abuse of rcall: note that rcall places the address after the current pc on the stack. And we have an instruction that reads a value off the stack into a register--namely pop. So we can do:

```
rcall 1
pop r0
```

and this will put the stored pc into register r0! Note that this stored pc is the address of the pop instruction in program memory, but we can subsequently adjust r0 with arithmetic operations to get r0 to be the address of whatever instruction we're actually interested in.

5.5.6: Further ISA features

We have described most of the major families of instructions in the AVR ISA, but there are a very large number of instructions defined as part of the ISA beyond what we've mentioned. Some of these perform functions that we have not mentioned, such as reading an actual encoded instruction from the program memory into a register. Others perform functions that we can perform by using the instructions described here, but which are packaged into a single extra instruction for convenience. An example of such an instruction that is not strictly necessary is cp. We note that cp sets the sreg flags in exactly the same way as sub, so everywhere we used cp, we could have used sub instead. The inconvenience is that sub also modifies one of its operands, and if we're just comparing two numbers we often don't want this comparison to also modify one of them. So we can do literally the same thing as

```
cp r10, r11
```

by saving off the value (r10) that will get modified by sub and then restoring it after:

```
push r10
sub r10, r11
pop r10
```

Not all of these "convenient but not strictly necessary" instructions are available on all models of AVR computer; cheaper chips may have a more bare-bones set of available instructions.

As of 2015, most AVR chips are manufactured by a company called Atmel. For each model of chip that they sell, they publish a document that explains the features of that model (including all the supported instructions) called a **datasheet**. For example, to see what the relatively simple Atmega103 model of AVR chip supports, a quick web search for "Atmega103 datasheet" will turn up a copy of the relevant document (usually as a PDF) for your perusal.

5.6: Assembly language

Summary: Assembly language is the programming language that one often uses to write code in the ISA. It is easy to convert from assembly to nothing but pure ISA instructions--a job that is done by an 'assembler'--but assembly has features that make it easier to use than the ISA itself.

Definitions: assembly language, assembler

Writing useful code using the ISA can be a pain. In one sense coding at this level is going to be difficult no matter what, since each instruction only performs one small operation at a time. At the same time, there are some things that are sufficiently difficult that a very slight additional layer was created to put on top of the ISA: assembly language.

Assembly language is a programming language, so like Python is defined in terms of its syntax and semantics. Unlike Python, however, assembly language's syntax mostly consists of raw ISA instruction mnemonics exactly as we learned them in the previous sections, with just a few additional features to simplify some of the more painful tasks when programming in the ISA directly. The tasks of this sort that we shall address specifically are the following:

- **Specifying offsets for relative jumps and branches:** Recall our example rcall code: We have multiple rcalls all calling back to a single place in the code. But because rcall requires an offset, we had to recompute the operand for each individual rcall. Further, if we ever want to add anything to the middle of the program, we would have to recalculate all these offsets again! Assembly language includes a mechanism called labels for naming locations in the code, and allows these to be provided as operands to rjmp and rcall instead so that we can effectively think "jump to this point in the code" rather than "jump 3 instructions backward".
- **Initializing RAM with specified data:** Currently, if we want a large amount of data stored in RAM, then since the only way of storing data we have is with the st instructions, we will have to do one st instruction for each number we want stored. This can get tedious if we have hundreds of things to store, so assembly provides a mechanism to pre-populate RAM at specified locations with whatever values are desired.
- **Strings:** We haven't yet discussed how Python's strings can be realized in the ISA. This discussion will largely be deferred to the next chapter, but for now we mention that there is some way to correspond letters in a string to numbers 0-255. So storing a string would involve looking up which letters correspond to which numbers and using st instructions to store the appropriate sequence of numbers. Assembly includes syntax to let us simply type the desired string and have the assembler convert this to numbers for storage.

Now, with assembly language, we're still trying to program the computer--that is, recall, we're trying to get the right numbers corresponding to our desired instructions into the program memory of the computer. When we write a program in assembly language, a program called an **assembler** (analogous to the Python compiler) will convert this into numbers for storage into the program memory. However, assembly language not only allows you to populate program memory with numbers corresponding to instructions, but also allows you to pre-populate RAM with numbers if desired.

As mentioned, assembly language is a programming language, so just as we explained the Python programming language in terms of its syntax--what constructions are legal, and its semantics--what the constructions mean, we can do this with assembly. Further, since all an assembly language program is meant to do is turn into certain numbers that will go into program memory and certain other numbers that go into RAM, describing the semantics of an element of assembly language is as easy as explaining what numbers it corresponds to and where in memory those numbers are stored.

Language element	Syntax	Semantics	Examples
Line	Either an *ISA operation*, a *label*, or an *assembler directive*.	A line is the basic unit of assembly language code. Each line in an assembly program is run in sequence.	

ISA operation	An ISA operation can be any of the above operations we described. The added features in assembly language are:
	• In place of any operand that represents an offset, you may use a label name
	• In place of a number you may use a character
		Performs the specified ISA operation.	`ldi r31, 51`
			`cpi r14, 'A'`
			`breq -17`
			`rjmp hello`

Label	*`label_name`*`:`	Marks this offset in the program as being named by the given label name.	`hello:` `this_is_a_label:`

Assembler directive	Either a *byte directive*, or a *string directive*.	Provides methods for easily getting data into RAM.	`.byte(20881) 101`
			`.byte(106) 121,87,0,33`
			`.string(8782) "hello"`

Label name	Any combination of letters, numbers, and underscores (no spaces), except that a variable name cannot begin with a number.	Specifies the name of a label.	`helo`
			`hellow_world`
			`abc1DEF0__0GHIJ`
			
			Non-examples:

			2hello 7 my name
byte directive	.byte(*RAM address*) *comma-separated list of bytes*	Instructs that these bytes should be inserted into the program at this point. Note that a byte can either by a number 0-255 or a character.	.byte(1872) 42 .byte(110) 0, 0, 0, 0, 0 .byte(37373) 123, 'A', 1
string directive	.string(*RAM address*) *string*	Translates each character in the string into a byte and inserts those bytes into the program at the location of the string directive in order.	.string(12345) "Hello world!" .string(919) "AAA&AAAA!!"
Character	Any single character enclosed in single quotes.	Stands in for a number that represents that character (see ASCII in the next chapter).	'h' 'A' '%' ' '
String	Any sequence of characters enclosed in double-quotes.	Strings provide a method to store and manipulate text.	"Hello world!" "'That is beautiful', he said"
Comments	Anything on a line after a semicolon (;).	Comments can be anything and are ignored by the assembler	; rhbuybshdbfkjshbfIFB pop r0 ldi r31,40 ;loads value 40 ;into register r31

As with Python, assembly language has in fact many features beyond those explained here, but they are much less powerful than the extra features of Python, and from this book we shall elide them entirely.

5.6.1: ISA Operations

We have already discussed in detail the syntax and semantics of the various ISA operations. The only thing new in assembly language is the ability to replace certain operands with characters or labels. Specifically:

- Any operand designated above as an 'offset' can be replaced with the name of a label. This covers the operands to rjmp, rcall, and all the branch instructions. The assembler will then compute the offset in the program between the jump/branch instruction and the location in the code where the label appears and will substitute that offset as the actual operand.

  For example, take the program:

```
hello:
ldi r31,0
cpi r31,8
breq stuff
inc r31
rjmp hello
stuff:
dec r31
cpi r31,8
breq stuff
```

The label 'stuff' appears two instructions after the first 'breq stuff' instruction, so the assembler replaces the first 'breq stuff' with 'breq 2'. On the other hand, the stuff label appears three instructions before the second 'breq stuff', so this is replaced with 'breq -3'. Likewise, 'rjmp hello' gets replaced by 'rjmp -5'. So the above program is equivalent to:

```
ldi r31,0
cpi r31,8
breq 2
inc r31
rjmp -5
dec r31
cpi r31,8
breq -3
```

Of course, writing the program using labels means we don't have to adjust all the offsets of our jumps and branches if we change the program a little, so labels do indeed solve that problem.

- Any character corresponds to a byte in a standard way. So with just the naive ISA operations, we could already do string manipulation by taking any string we wanted to use, figuring out which bytes its characters corresponded to, and using those numbers in place of the actual string. This would make the purpose of a program rather hard to divine, however. Imagine:

```
ldi r30, 255
out 26, r30
ldi r30, 72
out 27, r30
ldi r30, 255
out 26, r30
ldi r30, 105
out 27, r30
```

We know from earlier that this is outputting the numbers 72 and 105, but these numbers turn out to correspond to the characters 'H' and 'i', respectively. So this is outputting the text 'Hi', and we can make this more evident by replacing the numbers with the characters they correspond to:

```
ldi r30, 255
out 26, r30
ldi r30, 'H'
out 27, r30
ldi r30, 255
out 26, r30
ldi r30, 'i'
out 27, r30
```

The assembler will turn this into the earlier program anyway, since ldi of course requires a number as its second operand, but we are free to use characters when they are more convenient than the raw numbers and let the assembler take care of the conversion.

As we mentioned, the assembler's job is to put numbers into program memory corresponding to the specified instructions. We've seen how to specify ISA operations, even using labels and characters when convenient, but have not discussed the conversion of ISA operations into numbers. This will be the topic of the next chapter, and for now all we mention is that there is some correspondence. For example, the instruction

```
ori r22,0x58
```

corresponds to the number 26725.

5.6.2: Labels

Labels are lines that look like:

```
label_name:
```

Where a label name can be exactly what a variable name in Python could be: Anything containing only

letters, numbers, or underscores, except it cannot start with a number.

Labels are used to mark a place in the program to refer to in actual instructions, as in the examples of the previous section. They do not themselves constitute actual instructions. For example, the two programs below are identical in terms of the instructions that will be stored in program memory:

```
ldi r30, 255
out 26, r30
ldi r30, 'H'
out 27, r30
ldi r30, 255
out 26, r30
ldi r30, 'i'
out 27, r30
```

```
hello:
ldi r30, 255
out 26, r30
ldi r30, 'H'
some_label:
out 27, r30
ldi r30, 255
babel:
out 26, r30
ldi r30, 'i'
out 27, r30
```

5.6.3: Assembler directives

We will discuss only two kinds of assembler directives here, which will serve the single purpose of helping us pre-populate RAM with data. We should note that this task is not an easy one--

Byte directives

A **byte directive** looks like:

```
.byte(1231) 12,189,200,0,0
```

It will write the sequence of bytes (separated by commas) in the list into RAM at the specified address.

For example, the above directive creates the following situation in RAM:

Address	Value
1231	12
1232	189
1233	200
1234	0
1235	0

In general, however, we don't write programs using the byte directive. Instead, we use the byte directive to include data that we want to reference in our program. For example, if we want to sum the first 10 Fibonacci numbers, we could write some code to compute these, or we could just place them into RAM directly with a byte directive:

```
.byte(1005) 1,1,2,3,5,8,13,21,35,56
```

Another sort of data that we often want to manipulate is string data. Using the byte directive, we can store the string 'Hello World!' in RAM like:

```
.byte(504) 'H','e','l','l','o',' ','W','o','r','l','d','!'
```

But what bytes does this store? Consulting the ASCII table from the next chapter, which tells us how to encode characters as bytes, we see that this is equivalent to

```
.byte(504) 72,101,108,108,111,32,87,111,114,108,100,33
```

String directives

That last example for including strings may have looked a little inconvenient, so assembly language provides a second directive that makes this easier: the **string directive**.

The string directive looks like:

```
.string string
```

where a string is the same as it was in our description of Python--any sequence of characters enclosed in double quotes, where newlines, tabs, and double-quotes inside the string are all represented with escape sequences.

For example, we can recover the 'Hello World!' example with the string directive simply thus:

```
.string(504) "Hello World!"
```

5.7: Python-to-assembly recipes

Our claim is that everything we could do in Python, we can do in the ISA. Since assembly translates directly to the ISA, we only need to substantiate the claim that everything we can do in Python, we could instead do in assembly language. To this end, we take a few constructs from Python and see how they can be realized using assembly language.

5.7.1: Large numbers

One of the most constraining features of the ISA is its apparent inability to manipulate numbers above 255. For instance, the add instruction only acts on registers, and registers can only store numbers 0-255. The method for handling larger numbers is to view a collection of registers as all together holding a single number. We have introduced this concept informally before in cases where two registers r30 and r31 were said to together represent the number

```
r30:r31 = 256*r30 + r31
```

To this, we can tack on more registers if we so choose: The largest number the two-register scheme can store is 255 + 256*255 = 65535, so if we add on, say, r29 to the mix, so now the three registers r31, r30, and r29 will all represent a number, then the formula for the number they collectively represent will be:

```
r29:r30:r31 = 65536*r29 + 256*r30 + r31
```

Likewise, if we want four registers, since the largest number three can represent is

```
65535*255 + 256*255 + 255 = 16777215
```

Thus the four-register scheme should represent

```
r28:r29:r30:r31 = 16777216*r28 + 65536*r29 + 256*r30 + r31
```

We could of course repeat this until we run out of registers to use, but for many purposes, four registers worth of number is adequate.

However, just being able to store a large number is not enough--we want to be able to do things like add two such numbers. Say r28-r31 are being used to store one such number, and r24-r27 are being used to store another. We have an ISA instruction to add a register to a register, but not one to add a four-register number to another four-register number.

One first guess might be that we can add the constituent registers individually using the add instruction:

```
    add r31,r27
    add r30,r26
    add r29,r25
    add r28,r24
```

This actually works in certain cases: for instance, for adding the numbers 1:1:1:1 = 16843009 and 4:3:2:1 = 67305985, if we simply add the constituent registers, we get 5:4:3:2 = 84148994, which is the sum of the two represented numbers.

Essentially this will work until we try to add the registers and we get an overflow. For the simplest example, suppose we are trying to add 0:0:0:255 = 255 and 0:0:0:1 = 1. In this case, adding the registers individually yields and answer of 0:0:0:0 = 0, rather than the desired answer of 256 = 0:0:1:0.

The solution reflects the addition procedure many people are taught in grade school: To add two numbers, you don't just add their digits blindly, but you add the digits and you 'carry the 1' over to the next pair of digits if need be. In our case, when we add two registers, if there is a carry, the C flag in the sreg gets set, and in that case, the sum of the next two registers also needs a 1 added to it. But this is precisely what the adc instruction does! So all we have to do is:

```
    add r31,r27
    adc r30,r26
    adc r29,r25
    adc r28,r24
```

This whole scheme might easily seem a little uncomfortable--we're not actually storing the large numbers we claim to be, but simply storing pieces and 'understanding' those pieces to collectively 'represent' the big numbers. In a sense, though, this isn't unlike how we don't actually write down the number 3889, we write the four 'pieces' ('digits', we call them) '3', '8', '8', and '9' next to each other and somehow that represents the number three-thousand eight-hundred eighty-nine.

Nevertheless, one thing that might be ultimately convincing that what we have done isn't fake is if we can receive from the user a two big numbers as input (so we'll get them in string form), add them, and then output their sum in string form. This should make it believable that what we've implemented is actually arithmetic on numbers larger than 255.

To accomplish this, we'll ensconce the addition routine above into a function:

```
add_big_nums:
add r31,r27
adc r30,r26
adc r29,r25
adc r28,r24
```

5.7.2: if statements

In Python, we had an easy way of executing certain chunks of code conditionally using the if/else construct. For example, if we have two variables x and y and we want one chunk of code to run if they are equal and a different chunk to run if they are not equal, we could do:

```
if (x == y):
   stuff to do if equal
else:
   stuff to do if not equal
```

In assembly, we don't have such a construct, but we can mimic its behavior using our conditional branches--specifically breq which branches if the Z flag is set. Broadly, the structure of such a program would look like:

```
compare the two numbers and set the Z flag if they're equal
breq jump_here_if_equal
stuff to do if not equal
jump to end_of_conditional_code (so that we don't also do the stuff to do if equal)
jump_here_if_equal:
stuff to do if equal
end_of_conditional_code:
```

Under this setup, if the numbers are equal, then the branch skips over the things we want to happen if they aren't equal. On the other hand, if they weren't equal, then the branch is not taken and the code goes straight into the stuff that should happen if they are not equal, but then at the end of that code, it jumps over the code following, which should only happen if the two numbers were equal.

Replacing English descriptions with assembly language where possible, we get:

```
cp r0,r1
breq jump_here_if_equal
stuff to do if not equal
rjmp end_of_conditional_code
jump_here_if_equal:
stuff to do if equal
end_of_conditional_code:
```

We can easily modify this to represent different tests--e.g.

```
if(x >= y)
```

or

```
if(x < y)
```

by changing the branch to, say, brge, or brlo.

5.7.3: while loops

Similarly, the Python while loop construct allowed us to repeat a chunk of code for as long as a given condition remained true. For example, to do some things as long as two variables x and y have the same values:

```
while (x == y):
   do stuff
```

To do this in assembly (again supposing that the two numbers we wish to compare are stored in registers r0 and r1), we can use branches and jumps again. For example, we could say "First test if r0 and r1 are different. If not, the continue on to the 'do stuff' code, after which we should jump back to the comparison at we started with. If they were different, on the other hand, we should branch to the end of the whole thing."

In code:

```
begin_while:
cp r0,r1
brne end_while
do stuff
rjmp begin_while
end_while:
```

5.8: Simple examples

5.8.1: Summing the first n integers

Introduction:

In the earlier factorial example, we wrote a program to receive as input a positive integer n and to output the product 1*2*3*...*n. In this section, we'll discuss how to do this in ISA, except because we lack a multiplication instruction (there is one on certain AVR processors, but not on all), we'll do the problem with addition instead: Given an n, compute 1+2+3+...+n.

Algorithm:

We now have to describe a sequence of operations that performs this calculation. Once again, for a human, just saying, 1+2+3+...+n is likely enough, but for a computer, remember that all it understands are ISA operations, and 'compute the sum of the first n positive integers' was not one of them.

We can, however, draw inspiration from our Python approach: There, we had a loop with a variable counting up from 1 to n, and another variable that started at 0 and got the counter variable added to it each time. These are all things we can do once again, except there are no "variables" in the ISA. There are, however, registers, and recall our instruction that registers will be used in place of non-array variables. Thus we pick two of them--say r16 and r17, to be the counter and accumulator, respectively. Then, we'll start by getting the input into r31, and then setting r16 to 1 and r17 to 0 with ldi.

Now we need a loop, so we put a "start of loop" label. And inside the loop, we'll do the following operations:

1. Add the counter to the accumulator (r17)
2. Increment the counter (r16)
3. Check if the counter is larger than the input (r31)--if so, print r17 and exit; if not, jump to the start of the loop

Program:

Debrief:

5.8.2: Multiplication

Introduction:

We could almost convert the previous example into an assembly-language factorial program, but for two main difficulties:

- The numbers would get too big, even for two registers.
- We don't have a multiplication instruction!

Now, the AVR instruction set does include an instruction called mul, but many cheaper AVR chips you can buy do not actually support this instruction. So if we want to multiply numbers, we have to do it ourselves.

One issue that arises is that multiplying numbers is a good way to get large numbers, so it is likely that when we do multiplication, even if we're only multiplying two registers, we'll have to store the result in more than one register.

Algorithm:

We'll arrange our registers so that the numbers being multiplied are stored in r30 and r31, and the answer will be stored in the two registers r16 and r17, with r16 being the low byte and r17 the high.

Program:

```
ldi r31,13
ldi r30,9
ldi r16,0
ldi r17,0
ldi r18, 1
loop_start:
add r16, r31
adc r17, r18
dec r30
cpi r30,0
brne loop_start
```

5.8.3: Comparing two strings

Introduction:

Suppose we have two strings somewhere in RAM, say at addresses 200 and 650 respectively. That is, RAM looks like:

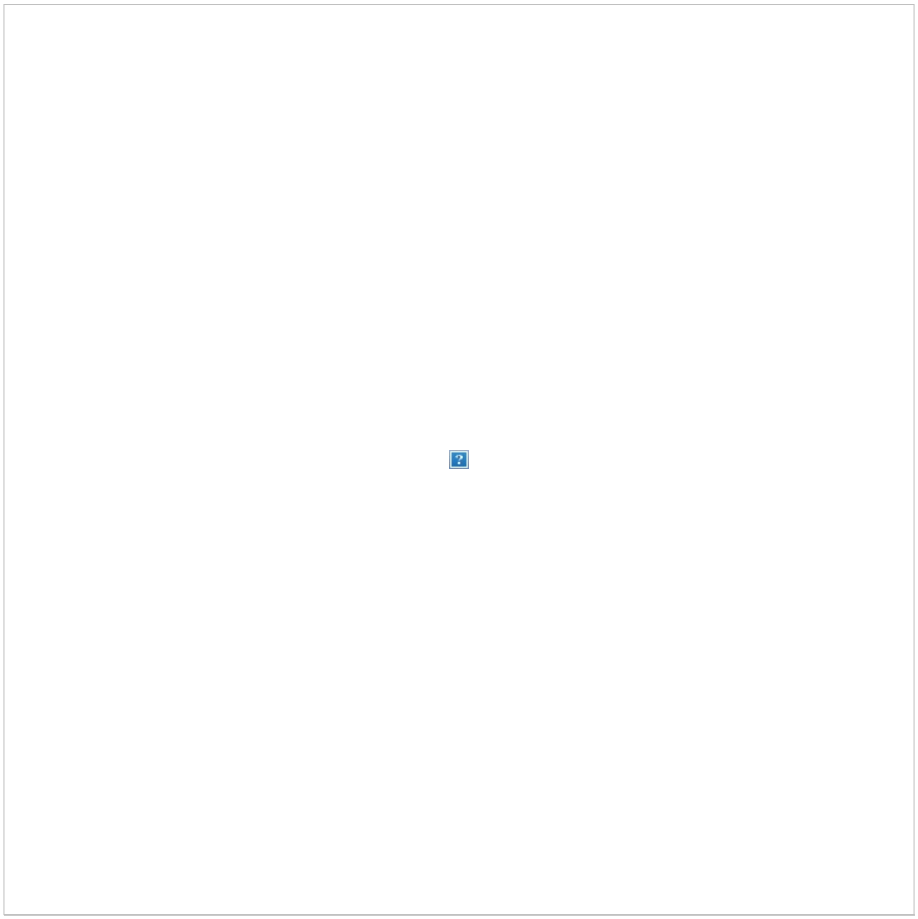

Figure 5.8.3.1:

Maybe the user typed one of them in and the other is a username, and we want to see if they match in order to, say, authenticate the user as having that username. In Python we could just do something like:

```
if(name_entered == name_stored):
   do stuff
```

We've already seen how to compare numbers like this in assembly, but haven't yet seen for strings.

Algorithm:

The idea isn't too far removed, however, from comparing numbers. Since all strings are just sequences of numbers, what we want to do is compare the numbers stored in RAM at addresses 200 and 650. Then, if those match, we need to compare the numbers at addresses 201 and 651. Then the numbers at 202 and 652. If at any point we don't get a match, then we know the strings are different. If, on the other hand, the two numbers match and are both 0, then that means we reached the end of both strings simultaneously, and so we're done.

Put into steps:

```
Put Y = 200
Put Z = 650
Read RAM address Y into r0
Read RAM address Z into r1
Compare r0 and r1
If they're the same, do the following:
   Compare r0 with 0
   If so, branch ahead to the code we run if the strings are equal
   Jump back to step 3
If we've got here, then the strings differ
```

Program:

Turning the above into a program is relatively straightforward, bearing in mind the constructs from the previous section about how to do conditional execution and while loops:

```
    ldi r28,
    ldi r29,
    ldi r30,
    ldi r31,
    comparison_start:
    ld r0,Y+
    ld r1,Z+
    cp r0,r1
    brne strings_not_equal
    cpi r0,0
    breq strings_equal
    rjmp comparison_start
    strings_not_equal:
    Whatever we want to do if they're not equal
    rjmp end_comparison
    strings_equal:
    Whatever we want to do if they are equal
    end_comparison:
```

5.9: Applications

The large-scale systems we have been describing are largely programmed using high-level languages like Python, with few to none of the pieces constructed by doing actual ISA programming. Having said that, Python is an abstraction over the ISA layer, and like all abstractions, this one sometimes leaks. That is, to make decisions about our Python programming, we need to understand what happens at the ISA level. We give some examples of that in this section.

5.9.1: Search engine -- Searching for a word in a string

Introduction:

In Python, we identified

```
if("banana" in s):
  do stuff
```

So in Python, testing if "banana" is found somewhere in a given string looks simple enough--it's just one command! But its being one command in Python doesn't mean that it takes the computer one step. Remember that the computer only runs ISA operations, and any Python code we write has to be turned into ISA operations before it can be run. So to understand what this very simple-looking Python is hiding, we need to understand what ISA operations this command corresponds to.

To this end, let us think about how to search for "banana" inside another string using the ISA. First, the strings have to be stored somewhere, and RAM is a natural location for them. So our first step will be to store the strings into RAM. Once all the strings are in RAM, then we want to use ISA operations to search for "banana" in the longer string.

Algorithm:

As we discussed, the first step of our algorithm will be to put the strings into RAM. But having done that, we now need to search for "banana". Because we know how to compare two strings already--that is, given two addresses, we know how to write ISA operations that detect whether the string starting at the one address is the same as the string starting at the other. Searching for one string inside another can be understood as doing this repeatedly:

Let Y be the address of the string "banana" stored in RAM, and Z be the address of the string we're searching through. Then we can search by comparing, byte-by-byte, the bytes from address Y to address Y+6 with the bytes from address Z to address Z+6. This will determine whether "banana" appears at the start of the string. Then we can compare the string "banana" with the bytes at Z+1 to Z+7, testing whether it appears one character after the start. And we can repeat this indefinitely until we reach the end of the string.

So our strategy--our algorithm--will be to take all the possible places where "banana" might appear in the string, and to perform the string comparison to check this. In particular, if the string has length L, then "banana" might appear starting anywhere from position 0 in the string to position L-6. So we do:

1. Put the strings into RAM
2. Check whether "banana" occurs at offset 0 in the string
3. Check whether "banana" occurs at offset 1 in the string
4. ...
5. Check whether "banana" occurs at offset L-6 in the string

Phrased without ellipsis:

1. Put the strings into RAM
2. Set a counter to 0

3. As long as the counter is less than L-6, do the following:
    1. Check whether "banana" occurs at address Z+counter
    2. If so, the answer is yes and we're done
    3. If not, then increment the counter

Program:

```
rjmp start
target:
.ascii "banana"
src:
.ascii "I like watermelons, kiwis, and bananas"
start:
...[move data into RAM]
ldi r17,0
ldi r20,0
loop_start:
cpi r17,32
breq loop_end
ldi r18,0
inner_loop_start:
cpi r18,6
breq inner_loop_end_found
ld r0,Y+
ld r1,Z+
cp r0,r1
brne inner_loop_end_not_found
inc r18
rjmp inner_loop_start
inner_loop_end_found:
ldi r20,1
rjmp loop_end
inner_loop_end_not_found:
inc r17
rjmp loop_start
loop_end:
cpi
```

Aside:

It is worth doing a back-of-the-envelope calculation to see whether the search-engine code we built in chapter 3 could function in an environment like the Google search engine. This program is going to perform at least 6 ISA operations for every character in the target string. As of 2014, Google had collected 200 trillion bytes of data from the internet. So to search all of these for "banana", we would have to perform 200*6 = 1200 trillion ISA operations in total. The fastest computer could perform maybe 4 billion of these a second, meaning it would take such a computer 300000 seconds--or over 3 days--to perform this search.

In contrast, an actual Google search for "banana" claims to complete in .25 seconds. Of course, Google has a lot of computers, but to go from 300000 seconds to .25 seconds, they would have to use 300000/.25 = 1200000 computers--all that just for a single search. In practice, they're handling well into the thousands of searches per second, so it is clear rather than simply needing more computers, they need a better algorithm. Just using the Python 'in' test, it is not immediately obvious what the ramifications are until we think of the operation in terms of the ISA.

In fact, the ISA code that Python uses to run the 'in' operation is quite sophisticated and substantially faster than the code described above, and does have a decent running time. The point is that the operation hides exactly what is happening and hence judging its use as appropriate based on the fact that the code looks simple can be quite misleading.

5.9.2: Game console -- Integer overflow

Introduction:

Algorithm:

Program:

5.10: Exercises

5.1:	In what ways is programming in a high level language different than programming in ISA?
5.2:	What does an ISA consist of? Explain the different parts.
5.3:	List some different types of memory on the AVR chip as discussed in this text.
5.4:	What are the significant flags used in the status register and when are they set or cleared?

5.5: What is persistent memory and why is it useful?

5.6: Explain three aspects of the ISA.

5.7: Explain the task of the assembler.

5.8: Define what the AVR processor's memory size and width are?

5.9: What registers can be used in the ldi instruction?

5.10: Whereas Python had three main functions, assignment, printing, and flow-control, what categories does the ISA instructions split into?

5.11: What type of instructions can be used to execute a non-sequential instruction?

5.12: Which 7 specific ISA operations can alter the program counter by more than 1?

5.13: Fill in each comment in the following AVR program saying which flag is set (N, Z, C, or none) after each of the specified lines finish executing

```
ldi r16 255
ldi r17 1
ldi r18 1
add r16 r17  ; __ flag is set
sub r17 r16  ; __ flag is set
sub r16 r18  ; __ flag is set
add r17 r18  ; __ flag is set
add r17 r16  ; __ flag is set
dec r17      ; __ flag is set
```

5.14: Can you think of a situation in which both the N and C flags would be set at the same time? N and Z? C and Z? What about all three flags?

5.15: Explain how the following program implements a kind of "loop" behavior. Is this an infinite loop? If yes, what makes it infinite? If no, at what point will the loop stop? How would your answer change if the second line of the program was "ldi r17 11" instead? Which operation could you replace the "brne" with that would give the program the same exact functionality?

```
ldi r16 10
ldi r17 3
inc r17
cp r16 r17
brne -3
```

5.16: Explain why simply switching the cp operands in the table from section 5.3.4 switches the functionality from >= to <=

5.17: What function does the immediate serve in immediate instructions?

5.18: Briefly explain the stack.

5.19: What is callee-saved registers and how do they differ from caller-saved registers?

5.20: The following program is unnecessarily long. Shorten it by replacing some of the lines with the instructions that were introduced in section 5.4.1. Make sure you do not alter the overall functionality of the program

```
ldi r16 2
ldi r17 4
ldi r18 6
ldi r19 8
sub r20 r16
cp r21 r17
sub r22 r18
cp r23 r19
```

5.21: What is AVR assembly language,what does it consist of, and how does it differ from AVR ISA operations?

5.22: What do the following assembly statements do? Draw the relevant portion of RAM with any updates made because of the statement.

1. `.byte(1204) 12,189,200,0,0`

2. `.byte(440) 'H','e','l','l','o'`
3. `.byte(440) 72,101,108,108,111`
4. `.string(440) "Hello"`

5.23: Explain how to operate with large numbers in AVR assembly language.

5.24: Which branch instruction is used largely when mimicking an if/else statement?

5.25: Why is complexity important when coding, using the search engine example as an example or springboard for your response?