CISP 317 (Assembly Programming)

Tak Auyeung, Ph.D.

November 10, 2003

- 20031028: add assignment 19
- 20030925: add section 15 for memory addressing mode analogies
- 20030905: add figure 1 and 2 to help illustrate the effects of `sbic`
- 20030904: add appendix for list of assignments 31.4
- 20030816: creation

Contents

1 Copyright Notice **6**

2 Instructor Information **7**

3 Class background (generic to all classes) **7**
3.1 Teaching philosophy 7
3.2 What not to do in a classroom or laboratory 7
3.3 Cheating 8
3.4 Grading 8

4 Background information **9**
4.1 Types of computers 9
4.2 What is a computer? 9
4.3 Programs 10
4.4 Computer architecture 10

5 Memory and I/O Organization **12**
5.1 Addresses 12
5.2 Busses 12
5.3 Memory Read 12
5.4 Memory Write 13
5.5 Address versus contents 13
5.6 Memory and I/O 13

6 Tools **14**
6.1 Getting the Tools 14
6.2 Installing the Tools 14
6.3 Managing your projects 15

7 Installing and Using AVR Studio **15**
- 7.1 Installation . . . 15
- 7.2 Starting AVR Studio . . . 15
- 7.3 Starting a Project . . . 15
- 7.4 Switching between School and Home . . . 16

8 Introduction to the AVR architecture **16**
- 8.1 What is it? . . . 16
- 8.2 The Simulator . . . 17

9 The AVR Core **17**
- 9.1 The ALU . . . 17
- 9.2 Registers . . . 17
- 9.3 Data Memory . . . 17
- 9.4 Code Memory . . . 18

10 Basic Input and Output **18**
- 10.1 General Concept . . . 18
- 10.2 The AT90S8515 . . . 18
- 10.3 GPIO on an AVR AT90S8515 . . . 19
- 10.4 Pull-up, What's that? . . . 19
- 10.5 Configuring GPIO . . . 19
- 10.6 Software Control . . . 20
- 10.7 Writing Your First Program (do not turn in) . . . 20
- 10.8 Reading the state of a push button . . . 21
- 10.9 Reading a Bit in an I/O Location and Making a Decision . . . 21
- 10.10 What have We Learnt? . . . 23

11 Homework Assignment (200 points), due one week from assignment date **23**

12 Binary Numbers and Other Bases **24**
- 12.1 Why binary numbers? . . . 24
- 12.2 Binary numbers are just as powerful! . . . 25
  - 12.2.1 More stuff later . . . 25
- 12.3 Negative binary numbers . . . 25

13 Register and Memory Instructions **26**
- 13.1 `ldi` . . . 26
- 13.2 `mov` . . . 27
- 13.3 `lds` . . . 27
- 13.4 `ld` . . . 27
- 13.5 `sts` . . . 28
- 13.6 `st` . . . 28

14 Memory Handling **28**
- 14.1 Data versus Code memory . . . 28
- 14.2 Labels . . . 29
- 14.3 Reserving Memory . . . 29
- 14.4 Loading an Address to `X`, `Y` or `Z` . . . 30

15 Analogy for Addressing Modes **30**
15.1 `ldi` . . . 30
15.2 `lds` . . . 30
15.3 `ld` . . . 31
15.4 `st` and `sts` . . . 31

16 Register Arithmetics and Conditional Branches **31**
16.1 Status flags . . . 31
16.2 Adding and Subtracting . . . 31
16.3 Adding an 8-bit Constant . . . 33
16.4 Conditional Branches . . . 33
16.5 Comparison . . . 34

17 Autoincrement **34**
17.1 Post Increment . . . 34
17.2 Pre Decrement . . . 35
17.3 Caution . . . 35

18 Homework Assignment (Assigned on 10/21, due in a week) **35**

19 Assignment 2: String Comparison 200 points (due two weeks from 10/28) **35**
19.1 What it does . . . 35
19.2 The Method (Pseudocode) . . . 36
19.3 The entire program . . . 36
19.4 How to Turn in . . . 36

20 The Basics of Programming Logic **36**
20.1 Sequences . . . 36
20.2 Uncondition branch . . . 37
20.3 Conditional branches . . . 37
20.4 Is that all to it? . . . 38

21 Pseudocode and Assembly Code **38**
21.1 hierarchical blocks . . . 38
21.2 if-then-else . . . 38
21.3 as-long-as-do, aka while-do . . . 39
21.4 do-while . . . 40
21.5 combining constructs . . . 40

22 Multiplication as an Example **41**
22.1 A quick reminder of decimal multiplication . . . 41
22.2 Binary multiplication . . . 41
22.3 Shifting . . . 41
22.4 Rotation . . . 41
22.5 AVR shift and rotate instructions . . . 42
22.6 Shifting more than 8 bits . . . 42
22.7 Multiply Pseudocode . . . 42
22.8 The assembly code . . . 43

23 One Address Leads to Another... 43
23.1 The Stack Pointer . . . 43
23.2 All Kinds of “Memory” . . . 43
23.3 `0x0000` to `0x001f` . . . 43
23.4 `0x0020` to `0x005f` . . . 43
23.5 Names of I/O Memory Locations . . . 44
23.6 Getting Back to the SP . . . 44

24 The Stack 45
24.1 The Concept . . . 45
24.2 The Terms . . . 45
24.3 Subroutines . . . 45
24.4 `call` and `ret` refined . . . 46
24.5 The Stack Pointer (SP) . . . 46

25 Subroutines 48
25.1 Example: 16-bit Integer Multiplication . . . 48
25.2 Copy and Paste, What’s the Problem? . . . 48
25.3 A Subroutine . . . 49
25.4 Calling a Subroutine . . . 49
25.5 A Simple Example . . . 50
25.6 16-bit Multiply . . . 50

26 Pseudocode to Real Code 51
26.1 Conditions . . . 51
26.2 Boolean and Branching . . . 52
26.3 Compound Conditions . . . 52
26.3.1 Conjunction . . . 53
26.3.2 Disjunction . . . 53
26.3.3 Mutual Exclusion . . . 53
26.4 Examples . . . 54
26.5 Long Compound Conditions . . . 54
26.6 Pseudocode Translation . . . 55
26.6.1 if-then-else . . . 55
26.6.2 while-do . . . 55
26.6.3 repeat-until . . . 55

27 Text Representation and Handling 56
27.1 Introduction . . . 56
27.2 Character Comparison . . . 56
27.3 Strings . . . 56
27.3.1 Length Delimited . . . 57
27.3.2 Null Terminated . . . 57

28 Pushing and Popping 57
28.1 An Analogy . . . 57
28.2 Example . . . 58
28.3 Saving and Restoring Multiple Registers . . . 59

29 Arrays in Assembly **60**
29.1 Uses of Arrays 60
29.2 Implementation of Arrays 60
29.3 Computing Addresses 61
29.4 Neighbors 62

30 The Frame **63**
30.1 The Stack as a Medium 63
30.2 The `strlen` Subroutine as an Example 63

31 Efficient I/O Instructions **65**
31.1 `sbi` 65
31.2 `cbi` 65
31.3 `sbis` 65
31.4 `sbic` 66

1 Copyright Notice

All materials in this document are copyrighted. The author reserves all rights. Infringements will be prosecuted at the maximum extent allowed by law.

You are permitted to do the following:

1. add a link to the source of this document at www.mobots.com
2. view the materials online at www.mobots.com.
3. make copies (electronic or paper) for *personal* use only, given that:
   (a) copies are not distributed by *any* means, you can always refer someone else to the source
   (b) copyright notice and author information be preserved, you cannot cut and paste portions of this document without also copying the copyright notice

2 Instructor Information

Name and title	Tak Auyeung, Ph.D.
Office	Lower Library #17
Office Hours	Mon: 1100-1250, Wed: 1100-1150, Thu: 1400-1550
Office Phone	916-484-8250
Email Address	auyeunt@arc.losrios.edu
General Webpage	http://www.drtak.org/teaches/ARC

3 Class background (generic to all classes)

3.1 Teaching philosophy

The classroom is a medium to exchange ideas particular to a course so that everyone can learn what the curriculum specifies. Students are not the only ones learning, though. Professors can always learn something new, such as better ways to teach a certain concept, more appropriate assignments to exercise learnt materials and etc.

Questions are welcome in my classes. A question reflects an attempt to understand or even to improve materials introduced in the class. Please raise your hand as soon as you have a question so I can address you as soon as possible.

Any disruption that makes it more difficult to learn is prohibited in a classroom or laboratory. I am not the only one who has a say of what is disruptive, a student can also tell me what or whom is disrupting the class.

Now, let's move on to grading. The main purpose of grading is *not* to differentiate students. The main purposes of grading is to:

- provide feedback of how a student is doing
- provide a scale for employers to evaluate a potential employee
- provide a scale for universities to evaluate a potential candidate

What should be used to assign grades? Grades are assigned based on how well a student demonstrate the understanding and application of knowledge that a class is supposed to teach. To be fair, all students are evaluated by the same means. Assignments and examinations are used to provide the necessary "demonstration" for grading purposes. There is more on this topic later.

Although when it comes to assignments and examinations everyone is treated the same (same deadlines, same grading criteria, etc.), there is help for those who need it. The lab time for programming classes and office hours for all classes are intended for students who need extra help in addition to classes. I will try to help you as much as I can during the lab time and office hours if you ask for help. However, there is no exception when it comes to the grading of assignments and examinations.

3.2 What not to do in a classroom or laboratory

Here is a list of specific things *not* to do in a classroom or a laboratory. Consider this an additional list to the list already posted in a classroom or laboratory, or specified in the contract that you sign at the beginning of a class.

- no food nor drink in the classroom, with the exception of plain water in a classroom without computers;

- absolutely no food nor drink (including plain water) in the lab or any classroom with computers for students;
- no phone conversation (this includes CB radio and other communication devices);
- no phone or pager ringing;
- no disruptive behavior.

I normally upload most (if not all) of my classnotes to my website at www.drtak.org/teaches/ARC. You are welcome to check and view my classnotes. However, all the classnotes are copyrighted, which means you cannot redistribute the materials in any form unless you have my consent.

I understand most people do not have 24-hour internet access. You are allowed to download the HTML and PDF files on your computer *for your own use only*. You can even print these files out for easier off-line viewing. However, please do not print the classnotes at the printers in room 152. The first reason is that I change my classnotes frequently, what you have printed may be obsolete already. Secondly, printing classnotes can overwhelm the technicians.

You are strongly recommended to print the classnotes using your own printer or at a lab where there is pay-per-print. I understand this is inconvenient and possibly costly. If you think this situation (not being able to print classnotes at room 152) is unacceptable, please contact my dean (boss), Barbara Blanchard at the CIT Area Office. I am more than happy to look into and implement solutions, but I need administrative support first.

3.3 Cheating

Cheating is not fair to other students and eventually is not benefitial to the cheater. All observed and reported cheating in the class will be investigated. All confirmed cheaters will be penalized. The penality of cheating is *at least* not counting the involved assignment or examinations. The professor reserves the rights for more punitive actions.

What constitutes cheating? In the context of a class, a student cheats if the student does not personally and independently complete submitted assignments or answer questions in a submitted examination. Working on questions in an examination using resources (time, notes, textbook, calculators and etc.) other than the ones allowed is also considered cheating. If an assignment is collaborative, a student can still cheat if the student does not contribute sufficiently to the submitted assignment. Furthermore, any student who helps another student submit work without personal and independent effort is also considered an accomplice in cheating. Any accomplice is penalized exactly the same as a cheater.

If a student suspects others are cheating, incidents can be reported anonymously. In other words, I will not disclose the informer without permission.

3.4 Grading

I use a fairly complicated scheme to determine the final grade. The first part of the final grade comes from assignments, the second part of the final grade comes from examinations.

Assignments make up one third of the final grade. In other words, if you submit all assignments on time and all assignments are 100% completed, it'll make up 33.3% of the final grade. Similarly, if you answer all questions correctly in all examinations, they'll make up 66.7% of the final grade.

For example, consider the following scores:

- assignment 1: 70/100
- assignment 2: 100/100
- assignment 3: 170/200

less than 0.5	F
at least 0.5 but less than 1.5	D
at least 1.5 but less than 2.5	C
at least 2.5 but less than 3.5	B
at least 3.5	A

Table 1: Final grade point to letter grade

- midterm 1: 7/10
- midterm 2: 9/10
- final: 18/20

The *proportion* in 33.3% is (70+100+170)/(100+100+200). In other words, out of a possible 33.3%, this student gets 340/400 of 33.3% or 28.30%. Similarly, the *proportion* in 66.6% is (7+9+18)/(10+10+20), 34/40 of 66.6% is 56.61%. The combined percentage is 28.30%+56.61% or 84.91%.

The combined percentage is from the grade point of an A (4.0). The grade point in this case, therefore, is 84.91% of 4.0 or 3.40. Technically, 3.40 is a little better than a B+ (3.30 is B+).

The final grade point (a number from 0.00 to 4.00) is then converted to a letter grade at the end of the semester using the following table:

I reserve the rights to change the grading scheme throughout the semester. I will make the changes public to the entire class whenever changes are made.

In order to receive your grade, you should sign up for a ZIP account if you have not done so already. You *must* sign up at on-campus, although you can access your account via the internet once you have an account. Visit http://zip.arc.losrios.edu to sign up for an account. Your midterm grade and final grade are both sent to this account. I will also send emails to your ZIP accounts regarding course materials and homework assignments.

4 Background information

4.1 Types of computers

Aren't computers the machines sitting the computer lab? Yes, those are definitely computers. However, there are many more types of computers. For instance, there is a computer inside every cell phone. Most cars have multiple computers performing various functions such as anti-lock braking, engine management and environmental control. When a computer is not visible (but performing useful functions), it is called an *embedded* computer.

There are also huge computers that span entire rooms. There are often called super computers. They are specialized to perform complex mathematical operations very quickly. Some are used to *simulate* how the structure of a car would crumble in a collision, others are used to simulate how a bomb would explode.

4.2 What is a computer?

A "computer" is a machine that *computes*. What is computing? In the context of this class, to *compute* is to perform the following:

- logical operation
- mathematical operation

Sounds pretty simple, doesn't it. Well, it *is* very simple. In fact, the type of mathematical operations that a computer natively knows is very limited:

- add
- subtract
- multiply
- compare
- and some *boolean* (explained later) operatons

The logical operations that a computer perform are also very simple:

- go do something if the last comparison yielded greater-than
- go do something if the last comparison yielded less-than
- go do something ...

We'll look into what a computer can do a little bit later.

4.3 Programs

Okay, so a computer can perform various simple operations, such as adding, subtracting and etc. How does a computer know in which order to perform these operations?

Well, the computer does not really *know* in which order to perform the operations. Instead, it *follows* some instructions to perform the operations. Vaguely speaking, these instructions *supplied* to the computer is a program.

How do we write instructions for a computer? Unfortunately, a computer does not understand English, Russian, Japanese, Chinese or any other natural languages. Natively, a computer only understands *binary* numbers (more on this later). In other words, the vocabulary of a computer makes up of words in zeros and ones only.

Well trained programmers can write small programs using this "native" format that computers understand. However, even for well trained programmers, writing programs in this native format is inefficient. In other words, a programmer can only write very little code per day.

In this class, we benefit from a tool called an assembler. Instead of specifying instructions in zeros and ones only, we can use weird words called "mnemonics" to represent *what an instruction does*. In fact, we'll spend quite a bit of time talking about these mnemonics.

4.4 Computer architecture

ALU Before we proceed any further to discuss how to program a computer, let us first return to the very basic operations that a computer can perform. Recall that a computer can natively perform simple operations like addition, comparison and specifying what to do next depending on the results of a comparison. All these operations are performed by a very tiny component called the *Arithmatic and Logical Unit* (henceforth the ALU).

The ALU is the heart of a computer. Every operation of a computer involves the ALU. In fast processors (such as Pentiums and PowerPCs), there may be multiple ALUs in the same computer. These ALUs can perform operations in parallel to improve the overall performance of the computer. For this class, however, we can assume a computer only has one ALU.

Registers If a computer can add, how does it get the number to add, and how does it store the result (sum) of the operation? Right next to the ALU are *registers*. For now, imagine registers as little post-its. The ALU can read from and write to registers. A computer has very few registers, but the ALU can access the registers extremely quickly. The add operation, for instance, can take a number from one register, take another number from another register, then put the sum back into the first register.

Registers are very quick to access, but there is very few of it. Even a Pentium only has hundreds of bytes of registers. By comparison, most programs require millions of bytes to execute. This means registers are far from sufficient to execute any useful program. Registers are excellent for passing the result of one operation to the next, but they are not useful for storing information that will not be reused for a while.

Processor (CPU) The *processor* consists of the ALU, registers and a few other components. On most computers, the processor is an individual integrated circuit (IC). On some computers, however, the processor and the other components (such as memory) are integrated onto the same physical IC.

Memory Most information that a program needs to access during its execution is stored in memory. Compared to registers, memory is much more plantiful. For instance, a normal PC has hundreds of millions of bytes in memory, compared to just a few hundreds of bytes in registers. A computer has special operations to copy data from memory to registers and vice versa.

Memory is typically used to store two main types of information. The first type is called "code", while the second type is called "data". "Code" refers to the program. Yep, the program itself must be copied or loaded into memory before it can be executed by the computer! "Data" refers to memory used to maintain information that is required by the program. For example, if a program is to count the number of entries in a file, the *logic* to count is loaded as "code", the actual counter (changed by instructions in "code") is stored as "data".

Some computers use the same memory for code and data, while others use separate memory (one for code and one for data). Computers using one memory for code and data are typically called Von Newman, while computer using one memory for code and a separate one for data are called Havard Architecture.

If memory is plentiful and essential to a computer, why don't we skip registers and just use memory? Well, some special computers actually do not use registers. However, for most computers, it is so much faster to access registers that storing *intermediate* results between operations in registers saves much time. For instance, most PCs require ten times more time to access (read or write) memory compared to registers.

Input/Output Even with the ALU, registers and memory, a computer is fairly useless. It is useless because it cannot acquire information from the real world and output processed information. For a computer to be useful, it also needs methods to input and output information from and to the real world.

Input/Output (henceforth I/O) is an essential element of any computer. Some computers have special addressable slots that connects to peripherals that interacts with the real world. Other computers "map" certain memory locations to peripherals that interacts with the real world. The former is called to have *a separate I/O space*, while the latter is called to have *memory-mapped I/O*.

On a "normal" PC computer, the I/O of a computer connects to the keyboard, mouse, floppy disk drive, hard disk drive, sound card and video card. On an embedded computer in a cell phone, the I/O connects to the keypad, the speaker, the microphone and the LCD screen. On an antilock computer in your car, the I/O connects to valves to release brake pressure, sensors to sense the rotation of tires and the indicator at the instrument panel.

5 Memory and I/O Organization

5.1 Addresses

Recall in 4.4 we discussed the importance of memory in a computer. In this section, we examine *how* a computer accesses information stored in memory.

The smallest unit of information in a computer is a *bit* (short form of *B*inary dig*it*). A bit can only represent either 1 or 0, which is not much information. It is seldom that a program needs to access the value of a single bit. Most computers organize memory in the smallest unit of *byte* (not a short form of anything). A byte is defined to be eight consecutive bits. Although a byte is still very limited, it is useful enough to be the base unit. We'll discuss how much information a byte can represent later.

Most computers can store many bytes in its memory. For instance, a PC desktop computer typically has 128 million bytes in memory. Obviously, a program does not need to access all 128 million bytes at the same time. The computer needs a method to access parts of its memory. This method is generally called *addressing*.

Because the smallest unit that is *addressable* is a byte, there are 128 million addressable bytes in the previous example. In other words, we need a method to access the contents of only one out of 128 million bytes. In order to do this, the computer assigns a unique *address* to each byte, just as the address of each house or the social security number of each person is unique. With this unique address, a byte in memory can respond when the computer specifies the address.

Unlike social security numbers, addresses of bytes in memory are well arranged. Bytes are numbered 0, 1, 2, 3, ... up to the number necessary for the last byte. If there are 512 bytes in memory, the bytes are numbered 0, 1, 2, 3, ... 511. In other words, the address of the first byte is 0, the address of the second byte is 1, and so on.

5.2 Busses

Physically, memory is connected to the processor via busses. A bus in this context is simply a whole bunch of wires. The interface between the processor and memory typically has the following components:

Address bus The address bus specifies which byte the processor wants to access.

Data bus The data bus allows the processor and memory to transfer the *contents* of the addressed byte.

Write enable This is one wire that specifies the direction of data transfer. If it is asserted, data goes from the processor to memory, otherwise, data goes from memory to processor.

Chip select Memory is not always paying attention to the busses and the write enable wire. Each physical memory IC *only* pays attention to the busses and write enable wire if the chip select wire is asserted.

5.3 Memory Read

In order to read from memory, the following transaction occurs:

1. processor specifies which location in memory to access by specifying the address on the address bus
2. processor indicates it wants to read from memory by negating the write enable signal
3. processor indicates which memory to read from by asserting the appropriate chip select signal
4. the selected memory chip decodes the address and locate the memory location to read

5. the selected memory chip presents the contents of the addressed location on the data bus
6. processor reads the data bus and acquires the contents of the addressed location
7. processor negates the chip select line to indicate the selected memory need to present the data any longer

5.4 Memory Write

In order to write to memory the following transaction occurs:

1. processor specifies which location in memory to write to by specifying the address on the address bus
2. processor indicates this is a write operation by asserting the write enable signal
3. processor presents the new content of the specified location
4. processor selects the correct memory chip by asserting its chip select signal
5. the selected memory chip reads from the address bus and understands which location should be changed
6. the selected memory chip reads from the data bus and changes the content of the selected location to the new value
7. processor negates chip select so the memory chip can go back to listen mode
8. processor stops specifying the value to write on the data bus

5.5 Address versus contents

It is very important to distinguish an address from the content at that address. The easier way is to understand address as *how* to get to a portion of memory, and contents as *what* is the value of bytes of that portion of memory.

5.6 Memory and I/O

Memory refers to the memory chips inside a computer. As useful as memory is, it is still internal to a computer. In other words, memory only enables a computer to remember. Memory does not let a computer know what is happening in the real world.

In other words, if computers were only equipped with memory, pressing on the keyboard and clicking a mouse would not do anything. Similarly, there is no way for a computer to output information to the screen or printer.

In order to *input* and *output* information to and from a computer, certain addressable locations, called I/O locations, are needed. I/O locations are similar to memory locations because a program can read from and write to such locations.

However, writing to an I/O location does not mean the written value is memorized. Instead, some signal external to the computer is altered. Simiarly, reading from an I/O location does not mean reading a previously remember value. Instead, some signal external to the computer is read.

For example, reading from an particular I/O location reads back which key is pressed on a connected keyboard. In general, some external device is responsible for specifying the value when an I/O location is read. An example of writing is writing to a sound card I/O location. Instead of having the written value remembered, the soundcard translates the value to the amplitude of voltage sent to speakers, resulting in

sound that a user can hear. In general, some external device should listen to some external signal when and a program writes to an I/O location.

As mentioned earlier, I/O locations and memory locations share some simlarities. Not only can a program write to and read from I/O locations, but it *must* also specify which I/O location to access. This is because a computer generally has many I/O locations, and a program is only interested in accessing one I/O location at a time. The collection of all I/O locations forms an "I/O space".

In some computers, most notably PCs, I/O locations do not share the same "space" as regular memory. This means there is a location zero for memory, and there is a different location zero for I/O locations. One instruction is used to access memory location zero, a different instruction is used to access I/O location zero. Writing to memory location zero is completely different from writing to I/O location zero. This approach is often called "I/O mapped" because there is a separate "map" for I/O locations.

In other computers, such as the AVR architecture that we will use, I/O locations and memory location share the same space. In the case of the AVR, locations 32 to 95 are designated as I/O locations, while locations 96 to 608 are memory locations. The same instruction (with different operands, see later discussion of instructions) is used to access location 32 (the first I/O location) and location 97 (the second memory location). This approach is often called "memory mapped" because I/O shares the same map used for memory.

6 Tools

The tools we need for this class is all *free*. Yes, free! This means you can download every tool we use in the class at home and do your assignments at home.

6.1 Getting the Tools

I recommend you download all the files onto your computer first, then we proceed to install the files. I assume you are using a Windows operating system (95/98/NT4/2000/XP). If you are using some other operating systems, please let me know so I can help you set up the tools.

Instead of running software from a link directly, we should save the files first. I assume you are saving files to `c:`
`temp`. To save a file from a link, right-click on a link, then select either "save link as..." or "save target as..." (depending on which browser you are using). Then select the folder to save the file.

If you have a relatively slow internet connection at home, please bring a Zip disk to the lab so you can download and save the files in the lab using a high-speed internet connection. I will *mirror* some of these files on my own machine so we don't hog the internet connection too much.

The first piece of software to download is AVRStudio. If you are viewing this as an HTML file, you can click on "AVRStudio" to get to the file. Otherwise, the full URL is "ftp://www.atmel.com/pub/atmel/astudio3.exe".

AVRStudio is a Windows GUI (graphical user interface) software that simulates a processor. In other words, it makes your PC pretend to be something else. It comes with all kinds of useful feature to help you debug programs.

6.2 Installing the Tools

I assume you either save the files you downloaded into `c:`
`temp` or have a CD-R that contains the files.

The first step is to install AVRStudio. This is quite easy. You can use the *File* Explorer in Windows to go to where you saved `astudio3.exe`, then double-click it. Accept all the defaults when you install this software.

6.3 Managing your projects

All of the programs for this class are single files. You can create a folder for this class and put all of your files here. *However*, I recommend using a subfolder for each project because this way you can keep track of multiple versions of the same project.

7 Installing and Using AVR Studio

7.1 Installation

You should either download the software from Atmel's website as described in 6. Alternatively, you may have a CD-R that contains the software. In either case, I assume you have the software at the `f:\cis34\` folder. If you have it places somewhere else, replace this part of a filename with your own folder.

First, open file explorer. You can do this by right-clicking on the icon labeled "My Computer", then select "Explore". Then navigate to where AVR Studio is placed. Double-click (or single click if your Windows operating system is configured to view everything as hot links) the icon for `astudio.exe`. This will create a dialog box asking you where to unzip files. If the editor box is not `C:\TEMP`, type it in, then press the button labeled "Unzip". This will unzip the file into multiple files at the `C:\TEMP` folder. Click "Close" when this step is finished.

Next, use the file explorer to navigate to `C:\TEMP`. You will find a subfolder `cdrom`, navigate into it. In this folder, you will find an icon for `SETUP.EXE`, double-click it. This starts the *real* installation software.

I suggest you take all the default settings and just hit "Next" and "Yes" buttons. This way, I know where you have installed the software and can help you later if you get into trouble. The installation is ended with a dialog box telling you about the completion, click "Finish" to exit the installation software.

At this point, you have the tool for the class!

7.2 Starting AVR Studio

At home (where *you* installed the software) or at the lab, you can click the "Start" button, click on "Programs" and find Atmel's folder, then click on "AVR Studio".

7.3 Starting a Project

Once you have started AVR Studio, you can start your project. To create a new project, follow these instructions:

1. click "Project", select "New"; a dialog box should appear
2. in the "Project name" field, type a name (with no extension). I would choose a name with up to eight characters.
3. in the "Location" field, choose a drive and/or folder. You may want to choose a removeable media so you can transport your files to other computers.
4. in the "Project type" field, choose "AVR Assembler"
5. click "OK"

Once the project is created, you will see the project view. At this point, the project has no files. You need to provide at least one (in the class, only one) source file. Right click on "Assembler Files" in the project view, then select "Create New File". This step combines the creation of a new file and linking the file to the project. It will display a dialog box. This time, type a name with an extension of `.asm`, and make sure you select the same drive/folder in the "Location" field. Leave the "Add to project" checked. Click "OK" when you are all done.

The previous step automatically opens an editor window. This editor window completely overlaps the project window. That's fine. You can click on the"Window" menu and see what windows you have available.

For testing, type the following program verbatim (or copy and paste if you are viewing this as a web page):

```
start:   nop
         nop
         rjmp start
```

Save the file first by clicking the "File" menu and select "Save".

Now you can assemble the source file (translate it into machine code). The easiest method is to click "Project" and select "Build and Run". AVR Studio will create a dialog box so you can select the target. We should select `AT90S8515` in the "Device" field, then click "OK".

This will start the program. You should see a yellow arrow to the left of the first line, and the first line should be inverted. This means the simulator is *about* to execute line one. You can press F11 (the function key) to execute one instruction and advance to the next. This particular test program is a "loop", and it does not really do anything. However, you should see the highlighted line and yellow cursor advance (and then go back) as you press F11.

7.4 Switching between School and Home

The best method to switch between school and home is to use a floppy disk, and keep all files on the floppy disk. However, floppy disks are not necessarily the most reliable media for transporting files.

A more reliable method is to send files via email. Most email accounts have web interfaces. The trick is to keep certain messages you send to yourself on the email server so you can save the attachment on any web-enabled machine. This technique has the extra advantage that the email server automatically dates the messages so you can differential between versions of the same file.

8 Introduction to the AVR architecture

The platform that we choose for this class is called the "AVR" architecture.

8.1 What is it?

According to Atmel, "AVR" does not actually mean anything. The term "AVR" usually refer to a family of *M*icro*C*ontroller*U*nits that share a common processor core.

The AVR is a RISC. RISC stands for *R*educed *I*nstruction *S*et *C*omputer. This means there are fewer instructions available on RISC processors than on the counterpart, CISC processors. The first 'C' stands for "complex" in this case. However, this does not mean that RISC processors are inferior in any way. In fact, there are quite a few advantages to a RISC processor (compared to a CISC):

- the programmer does not need to memorize as many instructions
- the processor uses up fewer transistors
- the processor runs cooler
- each instruction can be made to run faster
- the processor is cheaper to manufacture

An MCU is a computer-on-a-chip. This means the processor, memory and I/O are all residing in the same physical IC (integrated circuit). Compared to a comparable solution using multiple ICs, the MCU approach is much more econoical because of scale of production as well as the tiny material requirement.

8.2 The Simulator

Since CIS34 is not an electronics class, nor is it an advanced class, I do not expect students to be able to use specialized electronics for assignments. We are going to complete all of our assignments without using any special hardware.

Instead of using a real AT90S8515 IC, we will simulate the behavior of an AT90S8515 on a PC. The program that enables a PC to do this is called (quite naturally) a "simulator". With a simulator, we can test our programs comfortably on a PC, all without having to solder anything!

9 The AVR Core

The AVR core refers to the processor that is common to all members in the AVR family. The one we will use in this class, the AT90S8515, is only one of the many derivatives of the AVR core. All members of the AVR family share a whole set of features and characteristics.

9.1 The ALU

The AVR has one single ALU. However, it uses a technique called pipelining to improve performance. Compared to an architecture without pipelining, the 4-stage pipeline of the AVR ALU improves performance up to 400%. Although the implementation of a pipeline is complicated, the theory is really quite simple.

Imagine working in an assembly line that requires four different stages of assembly work. Without a pipeline, the first worker works on the product, then passes the 25% finished item to the next worker. At this point, the first worker waits! Similarly, the second worker works on the product, passes it on to the third worker and wait. The third worker works on the product, passes it to the fourth worker and wait. Only when the fourth worker finishes the product will the first worker pick up the next item to work on.

With pipelining, the first worker works on the first part, passes the 25% finished product to the second worker and *immediately* get the next raw product to work on it. If the first non-pipelined approach produces 40 items in one hour, the second pipelined approach produces 160 items in one hour. We assume each worker needs to spend an equal amount of time on the product.

Although the AVR ALU uses pipelining, it is still an 8-bit ALU. This means the *internal path width* of data is only 8-bit. The ALU can perform all kinds of 8-bit operations like adding two 8-bit numbers. By comparison, the Pentium processors have 64-bit internal path width, which means it can compute much larger integers in one clock.

9.2 Registers

The AVR processor has 32 *general purpose* registers. General purpose means for most operations, the programmer, you, can choose which register to use. There are, however, a few instructions that can only use a subset of the 32 registers.

Each register is 8-bit in the AVR. The first register is called register 0 (abbreviated to R0), the second register is called register 1 (R1) and etc. For instructions, R26 and R27 are combined to a 16-bit register called X. Simiarly R28 and R29 sometimes combine to a 16-bit register called Y, and R30 and R31 combine to Z.

When we introduce instructions, we will specify which registers can be used as operands.

9.3 Data Memory

Different variants of the AVR core has different amounts of data memory. The AT90S1200, being one of the smallest and most inexpensive member, has zero data memory. The ATMega128, the flagship, has 4096 bytes of data memory. The one we use in this class, the AT90S8515, has 512 bytes of data memory.

Note that all data memory is RAM (random-access memory). When power is lost, the contents of RAM is lost as well.

9.4 Code Memory

All members of the AVR family store program code in flash memory. Depending on the member, there can be 1024 bytes to 128k bytes of flash memory for storing code. Flash memory differs from RAM in that flash memory retains its contents even when power is shutdown.

Instead of counting code memory in bytes, a better method is to count by words. A word is two bytes. Most AVR instructions fit in one word. This means with 1024 bytes, we can store up to 512 instructions. 512 instrutions may not sound like a lot, but it is enough to perform many interesting and complex tasks.

10 Basic Input and Output

One of the many advantages of an MCU is that it has built in input and output devices. This section discusses input and output in general, then focuses on particular devices available on the AVR.

This turns out to be a fairly long section. Don't read it in a single breath. Take it in a little at a time.

10.1 General Concept

A computer is useless if there is no method to input information or output information. Regular computers have screens and printers for output, while accepting mice, keyboards and scanners as input. An MCU, as a single integrated circuit, does not have such fancy input/output devices.

Instead, an MCU often take direct electrical potential (voltage) as input and output. When a pin of an MCU acts as an input, it can detect changes of electrical potential within a certain range. Likewise, when a pin of an MCU acts as an output, the MCU can change the electrical potential at that pin.

While all MCUs can handle simple input and output as described, more sophisticated MCUs can do more. For example, some MCUs can interpret changing electrical potential at a pin as serial communication. This allows an MCU communicate with other computer devices, such as another MCU or a desktop computer.

10.2 The AT90S8515

This particular MCU can perform many interesting input and output functions. Just to give you an idea:

1. input: detect whether the electrical potential at a pin is "high" or "low"
2. output: change (drive) the electrical potential at a pin so that it is either high or low
3. intput/output: utilize changing electrical potential as a means to communicate with other MCUs or computers (Universal Asynchronous Receiver Transmitter)
4. output: generate square wave of a certain frequency but with a varying duty cycle (on time)
5. input: measure the amount of time when the electrical potential of a pin goes high or low
6. input: notifies software when the electrical potential of a pin changes
7. input: quantifies (measures) the electrical potential at a certain pin

Obviously, we will not get into most of these items. We will, however, talk about the first two items. These are commonly called "general purpose input/output", or GPIO in short.

10.3 GPIO on an AVR AT90S8515

The AT90S8515 has four "ports". Each port is an 8-bit entity that can be controled completely in software. A port has eight corresponding pins, each acting either as input or output. In order to utilize a pin for input or output purposes, we must first understand how each pin can be configured.

- direction: a pin can be configured as output or input
- for input pins:
  - specify a weak "pull up" to set a default state
  - read back whether the electrical potential (high or low)
- for output pins:
  - specify whether the pin should be driven high or low
  - reack back whether the electrical potential is actually high or low

10.4 Pull-up, What's that?

When a pin is pulled up, it means it is "weakly connected to a high (usually 5V) voltage". To understand pull-up, let us use the rubber-band analogy.

An input pin, when not connected to anything else, is a light load. Imagine a rubber band tied to a pencil. Because the pencil is a light load compared to the strength of a rubber band, the pencil stays close to the rubber band. Imagine you are holding the rubber band high with your left hand. Because the pencil stays with the rubber band, the pencil also has a high position. For this discussion, assume the rubber band stays attached to your left hand and the left hand stays at a high position.

However, when there is a significantly stronger external force applied to the pencil, the rubber band is overpowered. Think of you trying to pull the pencil to a lower position with your right hand. Because you right hand is significantly stronger than the rubber band, the rubber hand "gives" and let you pull the pencil to a lower position. This is similar to connecting a pulled-up signal to ground. Because the connection to ground is much stronger than the connection to 5V, the signal has no choice but to experience the ground (0V) voltage.

Remember, the rubber band is elastic. This means as soon as you release the pencil (from your right hand), the pencil returns to the original high position of your left hand. The same thing happens to a signal that is pulled high. As soon as the external connection to ground (0V) is removed, the signal returns to a high voltage state.

10.5 Configuring GPIO

Special I/O memory locations are designated to ports. In general, three locations are allocated to each port. The first location, `DDR`*x* (substitute `x` with `A`, `B`, `C` or `D` for the four ports), specifies the *direction* of each pin of a port. Bit 0 of the memory location is for pin 0 of the port, bit 1 for pin 1 and etc.

If a pin has a zero bit in the `DDR`*x* port, the pin is configured as input. If a pin has a one bit in the `DDR`*x* port, the pin is configured as output.

Regardless whether a pin is configured for input or output, software can always read back the actual electrical potential (high or low) at the pin. This may sound useless when a pin is configured output. However, for people who know about electricity and physics, it is actually helpful to read back a pin even when it is configured for output. This allows the software spot problems.

In order to read back actual pin status of a port, the `PINx` port is read. Again, substitute `x` with `A`, `B`, `C` or `D` for the actual port. Bit 0 of this I/O memory location indicates the status of pin 0, bit 1 indicates the status of pin 1 and etc. When a pin has a high electrical potential, the corresponding bit in `PINx` reads a 1, otherwise the bit reads a 0.

When a pin is configured input using the `DDRx` I/O memory location, one can select whether the pin is pulled high or not. When a pin is configured output, one can select which way (high or low) the MCU should drive the pin. Both of these actions (pull-up and driving) are specified by the I/O memory location `PORTx`.

For an input pin, if the corresponding bit in `PORTx` is 1, an internal pull-up is enabled. Otherwise, there is no pull-up resistor and the electrical potential is allowed to "float".

For an output pin, if the corresponding bit in `PORTx` is 1, the MCU attempts to drive the pin to a high electrical potential. Otherwise, the MCU attempts to drive the pin to a low electrical potential.

Please note that each pin of each port is *independent* to all the other pins of either the same port or other ports. This means the same port can have pins of all possible configurations: input with pull-up, input without pull-up, output driven high or output driven low. This provides great flexibility for any application that requires some of each type.

10.6 Software Control

With an MCU, the configuration of each pin is software configuration at run time. Yes, software can change a pin from output driven high to input without pull-up, for example.

The `sbi` and `cbi` instructions suffice for configuration purposes. For example, the following code ensures bit 4 of port A is an output pin:

```
sbi   DDRA,4
```

"sbi" means "Set Bit I/O". It requires two pieces of information: which I/O location and which bit. In our example, `DDRA` is the I/O location to change, and `4` means bit 4. The result of `sbi DDRA,4` is that bit 4 of I/O port DDRA is set to 1. In the context of I/O location DDRA, setting a bit to 1 means turning the corresponding pin to an output pin.

Similarly, "cbi" means "Clear Bit I/O". It also requires two piece of information: which I/O location and which bit. For example, to make pin 6 of port C an input pin, we can use the following instruction:

```
cbi   DDRC,6
```

Note that `sbi` and `cbi` only affect the specified bit in the I/O location (which has 8 bits). All the other bits remain as is.

10.7 Writing Your First Program (do not turn in)

We know enough to be dangerous now. Let us write the first program that does something.

The names `PORTA`, `DDRA` and etc. are defined by a file in the system. In other words, the assembler does not understand these words natively. In order to use these predefined names, you need to insert the following line before any reference to such names:

```
.include "c:\Program Files\Atmel\AVR Studio\Appnotes\8515def.inc"
```

The `.include` directive tells the assembler to include the contents of the specified file in the assembly process. Once this file is included, you can write a simple program as follows:

```
.include "c:\Program Files\Atmel\AVR Studio\Appnotes\8515def.inc"

   sbi   DDRA,0 ; pin 0 of port A is output
   cbi   DDRA,1 ; pin 1 of port A is input
   sbi   PORTA,0 ; pin 0 drives high
   sbi   PORTA,1 ; pin 1 pulls high
   nop ; put here so you can see the result of the last useful instruction
```

This program doesn't do much. However, I want you to view the I/O locations using View — New IO View in the simulator. This command pops up a window that contains all the I/O locations for the specified MCU (the 8515 in our case). If you click on the "+" next to "Port A", it expands to "Port A Data", "Data Direction" and "Input Pins" because these three I/O locations correspond to the physical port A. Note that each I/O location is displayed as eight bits.

As you step through the program, note that some of these bits become checked. When a bit is not checked, it has a value of zero. When a bit is checked, it has a value of one. Also, note that when an I/O location is changed, it is highlighted by a red color.

The I/O View makes it easy to view the contents of I/O locations. In addition, you can also use this view to change the contents of I/O locations. Click on a bit to toggle it between 0 and 1.

In this program, observe that bit 1 of "Input Pins" does not automatically change to 1 after `sbi PORTA`. Why?

10.8 Reading the state of a push button

Although it is uncommon to connect an advanced user interface to an MCU, it is common to connect push buttons to an MCU so an end user has a simple method to interact with the MCU.

Most of the time, a normally-open, momentary, single-pole-single-throw push button is connected to between an input pin and the ground. As a single-pole-single-throw button, it is only switched to one source (in our case, the ground). As a normally-open and momentary button, it remains open when no one pushes it, and only closes for as long as someone pushes the button.

Recall from section 10.4 what pull-up means. When a button is connected between a pulled-up input pin and ground, the button can affect the voltage experienced by the input pin. When the button is not pushed, it remains open, and the pulled-up input pin experiences a high voltage through the weak connection to high voltage. However, when the button is pushed, it provides a strong connection to ground and the input pin experiences a low voltage.

In other words, when the button is pushed, the input pin reads 0. When the button is released, the input pin reads 1. Although this sounds counter intuitive, this is actually how 99% of push buttons are implemented (that 0 indicates a button is pushed).

10.9 Reading a Bit in an I/O Location and Making a Decision

Two instructions can be used to *read* a bit of an I/O location and *decide* what to do next. The first instruction, `sbic`, means "skip if bit of I/O location is cleared". I know, it is cryptic. The second instruction is `sbis`, which means "skip if bit of I/O location is set".

In this context, "set" means a value of 1, "cleared" means a value of 0. "Skip" requires a little more explanation.

Normally, instruction executes one after another consecutively. See our sample program in the previous section. However, `sbic` and `sbis` allows the processor to skip or jump over one instruction depending on the state of a bit of an I/O location.

Consider the following program snippet (a portion of a program):

```
sbic PINA,3
blah
yada
```

Of course, `blah` and `yada` are not real instructions. Nonetheless, the focus is on `sbic`. If bit 3 of I/O location `PINA` is cleared, the program *skips* the next instruction and jumps straight to continue execution at `yada`. However, if bit 3 of I/O location `PINA` is set, the program does not skip and executes `blah` instead.

The next question is: do we *have* to execute `yada` after executing `blah`? Afterall, if this were the case, `sbic` would not have been a useful instruction.

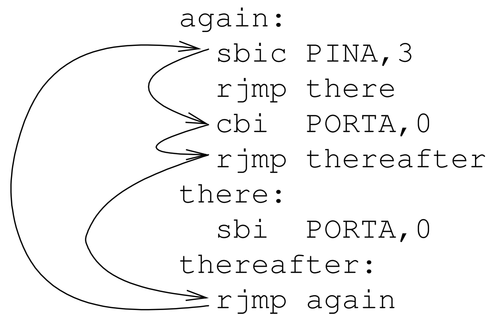


Figure 1: Illustration of instruction execution order when bit 3 of `PINA` is cleared.

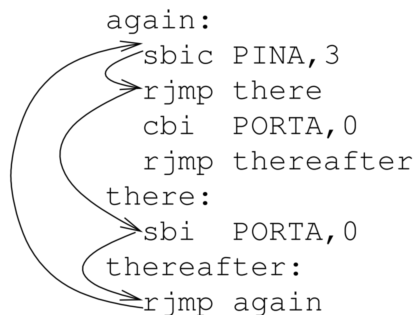


Figure 2: Illustration of instruction execution order when bit 3 of `PINA` is set.

Let us introduce another instruction, `rjmp`. `rjmp` means "relative jump", but we can ignore the "relative" part for now. `rjmp` requires just one piece of information: where to jump to. The location to jump to (to continue execution at) is often specified as a label. For example, `rjmp there` means continue execution at a label called `there`.

In order to define a label called `there`, we need one line in the program that looks like this:

```
there:
```

Whenever you put a colon after a symbol, it defines the symbol to be a label that marks a location.

Now, let us consider the following snippet:

```
again:
  sbic PINA,3
  rjmp there
  cbi  PORTA,0
  rjmp thereafter
there:
  sbi  PORTA,0
thereafter:
  rjmp again
```

What does this snippet do? You can assume pin 3 of port A was previously configured as input, and pin 0 of port A was previously configured as output.

Figure 1 illustrates how instructions are executed when bit 3 of `PINA` is cleared, where as figure 2 illustrates how instructions are executed when bit 3 of `PINA` is set.

10.10 What have We Learnt?

- `DDR`*x*: direction, input or output
- `PORT`*x*: the *desired* state of a pin
- `PIN`*x*: the *actual* state of a pin
- pull up means driving high very weakly so some external device can drive the signal low'
- circuit to connect a push button so its state (pushed or released) can be read as the state of an input pin (low or high, respectively)
- instructions may take pieces of information (operands)
- operands of an instruction are separated by (a) comma(s)
- `sbi` and `cbi` to change a bit of an I/O location
- `sbic` and `sbis` to read a bit of an I/O location make a simple decision
- `rjmp` to always continue execution at a label
- *label*`:` to define a label

11 Homework Assignment (200 points), due one week from assignment date

In this homework assignment, write a program in AVR assembly that does the following:

- configure pin 0 of port A as an input pin connected to a normally-open push button, use internal pull-up
- configure pin 1 of port A as an output pin connected to an LED, assume 0 means off, 1 means on
- initially turn off the LED
- assume the push button is released initially
- waits for the normally-open push button switch to be pressed and released twices
- turn on the LED and get into an infinite loop after the last button release

Your program should have the following elements:

- a line containing `.include` so you can use symbolic names of I/O locations
- initialization code to initialize the `PORT` and `DDR` I/O locations
- loops to detect button push, release, push and release events (see below)
- code to turn on the LED
- an infinite loop at the end

In order to detect whether a push button is pushed, assume the corresponding `PIN` bit reads 0 when the button is pushed, and reads 1 when the button is released. You can use `sbic`, `sbis` and `rjmp` to stay in a loop until there is a change of state.

Instructions that you'll need to use include the following:

- `sbic` skip if bit of I/O location is cleared
- `sbis` skip if bit of I/O location is set
- `sbi` set bit of I/O location
- `cbi` clear bit of I/O location
- `rjmp` (relative) jump
- definitions of labels

You will also need references to the following I/O location symbols:

- `DDRA` data direction for port A
- `PORTA` control for port A
- `PINA` sensed state of port A

To submit the homework assignment, follow these instructions:

- only turn in the `.asm` file, I do not need the other ones
- send an email to mailto:auyeunt@arc.losrios.edu with the following:
  - subject line should begin with “CISP317 Project1 by *your name*”
  - replace *your name* with your actual name
  - attach the `.asm` file to the message
  - carbon copy to yourself to be safe
  - request receipt if you wish (and your email interface supports it)

12 Binary Numbers and Other Bases

12.1 Why binary numbers?

People are used to *decimal* numbers. Decimal numbers consist of a variable number of digits, but each digit can only be one of 0, 1, 2, ... 9. There are 10 possibilities for each digit. The main reason for deciaml number is that (most) people have 10 fingers (digit means finger).

Obviously, a computer does not have any fingers. It is free to choose other *bases* to represent numbers. Most modern computers consist of many transistors, and each transistor can either be turned on or turned off. In other words, each transistor can represent one of two states.

If we assign the number “1” to “on” and the number “0” to “off”, each transistor can represent either 0 or a 1. In other words, a transistor can count from 0 to 1, or a digit that has two possibilities: 0 or 1. Although one transistor seems fairly useless, we can combine the states of many transistors to represent complicated data.

The *deci* in decimal means 10. When a transistor can only represent one of two number, the number is called *binary* (*bi* means two, as in bicycle–two cycles, two wheels!).

12.2 Binary numbers are just as powerful!

Just as a decimal number can have many digits, a binary number can also have many digits. A decimal number can represent any whole number, so can a binary number! In fact, any *value* that can be represented by a decimal number can also be represented by a binary number.

In addition, just as we can perform addition, subtraction, division and multiplication with decimal numbers, we can do the same with binary numbers. The rules are similar with a "spin" for binary numbers.

Before we explain how to do all kinds of operations with binary numbers, let us first examine how to read a decimal number. Yep, let us explain something that you already know!

Let us take an example, how about the number 157? The decimal number 157 can be broken down to $1 \times 100 + 5 \times 10 + 7 \times 1$. In other words, there are 1 hundred, 5 tens and 7 ones in 57. Great, everyone knows that! Now we introduce the concept of "powers".

What is 10 to the power of 2? This is the same as multiplying 10 to itself twice, which is $10 \times 10 = 100$. What is 3 to power of 4? This is the same as multiplying 3 to itself four times, which is $3 \times 3 \times 3 \times 3 = 81$. How about 5 to the power of 1? It is 5 multiplied to itself once, which is just 5 itself. Here comes the tricky one: what is 7 to the power of 0? This time, *by definition*, it is 1. In fact, any number to the power of 0 yields the result of 0.

The notation of power is superscripting the power. In other words, 10 to the power of 2 is written as 10^2.

Now, let's get back to the decimal number 157. Last time we wrote the number as $1 \times 100 + 5 \times 10 + 7 \times 1$. Let us exercise what we know about powers and rewrite it again to $1 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$. Hmmm, do you see a pattern here? The rightmost digit is multiplied to the power of 0, then each digit to the left multiplies to one more power of 10. The rightmost digit is also called the *least significant* digit.

What is the big deal of this discussion? Well, we can read a binary number the same way we read a decimal number. Yes, indeed!

To differential a binary number (base-2) from a decimal number (base-10), we add the subscript of 2 at the right hand side of a binary number. In other words, 100_2 is not 100. To be redundant, we can always add the subscript 10 for a decimal number, so 99 is the same as 99_{10}.

Let us try to interpret 110101_2 and find out what value it is representing. It turns out we can rewrite this number as $1 \times 100000_2 + 1 \times 10000_2 + 0 \times 1000_2 + 1 \times 100_2 + 0 \times 10_2 + 1 \times 1_2$. Once again, we can try to apply powers. However, because we are dealing with a binary number (instead of decimal number), the base of the powers is 2 instead of 10. Now the number becomes $1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$. Do you notice we just got rid of base 2 numbers? Now we replace the powers of 2 with the actual decimal numbers, and it becomes $1 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$. If you perform this computation, the result should be 53. In other words, $53_{10} = 110101_2$.

12.2.1 More stuff later

We'll pause the discussion of binary numbers right here so we can resume to discuss the organization of a computer. Later, we'll discuss how to perform arithmatic operations with binary numbers.

12.3 Negative binary numbers

The binary number system has an interesting method to represent negative numbers. This technique is called "two's complement". I'll use the C_2 symbol to denote this operation.

Before we talk about two's complement, let us first define "one's complement" (denoted by the C_1 symbol). It is important to note that for one's complement and two's complement, we need to know the exact number of bits used to represent a number. For example, let us assume quantities are represented by 8-bit patterns.

One's complement involves turning 0s into 1s and 1s into 0s in a binary number. For example, the one's complement of 00101110_2 is 11010001_2. We can simply say $C_1(00101110_2) = 11010001_2$.

The two's complement of a number is the sum of one's complement and 1. In other words, $C_2(X) = C_1(X) + 1$. In our previous example, $C_2(00101110_2) = C_1(00101110_2) + 1 = 11010010_2$.

Now, let me make a bold claim: if the number of digits is fixed, then two's complement is the same as negation. For example, 5 in binary is 00000101_2. It's two's complement is $C_2(00000101_2 = 11111011_2$. My claim says 11111011_2 behaves like -5 in binary operations.

Let us try a few experiments. The first experiment is whether 5+(-5)=0. In binary, we want to see if $00000101_2 + 11111011_2 = 0$. This gets a little tricky because the actual sum is 100000000_2. However, because we fixed the number of digits to 8, we only pay attention to the least significant 8 bits of the sum. This means 100000000_2 is really treated as 00000000_2.

The next experiment is whether 7+(-5)=2. In this case, we want to check if $00000111_2 + 11111011_2 = 2$. If you carry out the addition, the answer is 100000010_2. Since we only pay attention to the least significant 8 bits, the sum is, indeed, 00000010_2, which is 2 in decimal.

Note that 11111011_2 also represent 251 in decimal. We seem to have an identity problem here. Which way should be *interpret* 11111011_2? The answer is that your program is responsible to choose the proper interpretation. We will talk about this again when we talk about condition branches.

What is the most negative number that can be represented by an 8-bit number? Let's list some numbers and their two's complement:

number (decimal)	number (binary)	two's complement (binary)	-number
0	00000000	00000000	0
1	00000001	11111111	-1
2	00000010	11111110	-2
...	...	...	...
127	01111111	10000001	-127
128	10000000	10000000	-128

The problem is with decimal 128. Note that 128 and -128 both have the same binary representation! This is not acceptable because 128 and -128 are different values, they cannot have the same representation. By convention, the bit-pattern of 10000000 (for 8-bit signed numbers) is interpreted as -128. This means the most positive value an 8-bit signed number can represent is decimal 127.

Do you find any thing that is common to all negative values? Their binary representation all have a 1 for the most significant bit! For signed binary numbers, the most significant bit is also called the sign bit. If the sign bit is a 1, the value represented is negative. Otherwise, the value represented is non-negative (zero or positive).

13 Register and Memory Instructions

This section discusses simple instructions that move data among registers and memory locations.

13.1 `ldi`

The `ldi` instruction means load-immediate. This instruction is mainly used to copy an 8-bit constant to a 8-bit register. It requires two operands.

The first operand specifies the register to receive the constant. The original value of this register is overwritten by the constant. You can specify register 16 to register 31 in this operand.

The second operand specifies the constant to copy to the first operand. This constant must be an 8-bit value. Note that this is a *constant*, which means the program cannot alter the value once the program is assembled.

The following is an example:

```
ldi  r18,0x34
```

The previous instruction copies the constant of hexadecimal 34 (bit pattern 00110100) to register 18.

13.2 `mov`

The `mov` mnemonic means "move", but this is a misnomer. In the real world, we can move an object from one place to another. This means where the object was will be empty after the operation. In a computer, a location that can contain data is never "empty". Therefore, we should see this instruction as "copy" more so than "move".

The `mov` instruction copies the content of one register to another register. It requires two operands

The first operand specifies the *destination* register. This register will be overwritten by some value.

The second operand specifies the *source* register. This register provides the value to overwrite the first operand. You can use any of the 32 registers for this operand.

The following is an example of a `mov` instruction that copies the contents of register 20 to register 3. You can use any of the 32 registers for this operand.

```
mov  r3,r20
```

13.3 `lds`

The `lds` instruction means "load register from SRAM". SRAM means "static random access memory", which is more commonly known as data memory. With the `lds` instruction, the processor can load the contents of a memory location into a register.

The first operand specifies which register should be loaded. You can use any one of the 32 registers here.

The second operand specifies the address of memory location to access. Note that this address is fixed at *assembly* time, which means the program cannot change the address as it runs. In other words, this address is *static*. You can specify any number from 0x0000 to 0xffff for this operand, but you may not have 64kB of SRAM to access.

The following instruction loads the contents of location 0x014f into register 6.

```
lds  r6,0x014f
```

Please be careful not to confuse `ldi` with `lds`. `ldi` loads a *fixed* value to a register, while `lds` loads *the contents of a fixed address* to a register. This means a register always end up with the same value after a `ldi` instruction, but it may get different values after a `lds` instruction because the value of a location in memory may change as a program executes.

13.4 `ld`

The `ld` instruction means "load register from memory". Once again, it really should be "copy memory content to register". There are *many* variants of this instruction, we will only look at three common ones here.

The first operand is easy. It specifies which register should be copied into. You can specify any one of the 32 register.

The second operand is a bit more complicated. First of all, we need to discuss the register pairs `X`, `Y` and `Z`.

Register pair `X` is the concatination of registers 26 and 27. `X` is a 16-bit number where the most significant (leftmost) 8 bits come from register 27, and the least significant (rightmost) 8 bits come from register 26. This 16-bit number forms the *address* of a memory location. The processor then use this 16-bit address to specify a location (a byte) in memory. The contents of this memory location is then copied to the first operand.

Register pair `Y` is the concatenation of registers 28 and 29. `Y` is similar to `X` in how it is an address of a memory location to access.

Register pair `Z` is the concatenation of registers 30 and 31, `Z` is similar to `X` in how it is an address of a memory location to access.

This means that in order to use an `ld` instruction to access memory, you need to first load two registers with the address to access. This certainly seems more clumsy than the `lds` instruction. However, a `lds` instruction can only access a particular memory location. On the other hand, the `ld` instruction depends on the values of two registers. This means the `ld` instruction can access different locations in memory depending on the values of registers. This ability makes the `ld` instruction extremely flexible and useful.

The technique to use registers to specify an address to access is also called "indirect addressing" because the address of memory location to access is not direct (as in the `lds` instruction).

The following instruction uses the value of `Y` as an address, and copies the content at that location to register `r5`:

```
ld  r5,Y
```

13.5 sts

The `sts` instruction is like a counterpart of `lds`. It stores the value of a register into a location in memory.

The first operand specifies the location in memory to store the second operand. It can be any number from 0x0000 to 0xffff. However, the MCU may not have the entire range of memory available.

The second oeprand specifies which register to store. You can use any one of the 32 register.

The following instruction stores the value of register 23 to location 0x007e in memory.

```
  sts  0x007e,r23
```

13.6 st

The `st` instruction is like a counterpart of `ld`. It stores the value of a register to a location specified by either `X`, `Y` or `Z`. We are limiting our options to just these three for now. If you forget which `X`, `Y` and `Z` are, please refer to 13.4.

The first operand specifies which location in memory to store the value of the second operand. It can be either `X`, `Y` or `Z`. The value of `X`, `Y` or `Z` is a 16-bit quantity that is treated as the address of the location to store the second operand.

The second operand specifies which register to store into memory. You can use any one of the 32 registers.

14 Memory Handling

14.1 Data versus Code memory

Because the AVR is a Harvard architecture, we maintain two separate memory spaces. One memory space is for storing data (that changes when a program executes), and one memory space is for storing program code (that does not change when a program executes). This means there is an address zero for data space, and an *independent* address zero for code space.

When you write a program, you are using code space by default. In order to reserve RAM for variables, you need to instruct the assembler that you want to switch to data space. The following commands are used to switch between data and code space:

- `.cseg` tells the assembler to switch to code space
- `.dseg` tells the assembler to switch to data space

14.2 Labels

From time to time, you need to refer to a particular memory location. For example, you may need to read a byte from a memory location, or branch to a particular location if a condition is satisfied. Without an assembler, we need to count memory locations by hand. This means you need to find out the size of each instruction and variable and keep track of the current location manually.

Fortunately, the assembler can keep track of addresses for us. Furthermore, the assembler allows the programmer assign "names" to memory locations. These names must follow a simple set of rules, but otherwise they are only determined by the programmer. Each "name" is formally called a "label".

Label definitions are denoted by a colon (':') following the label to be defined. For example, if you need to perform a conditional branch, you need to specify the location to branch to. The following example illustrates how to skip incrementing one if a register already has a value of `0xff`:

```
    cpi   r16,0xff ; compare r16 with 0xff
    breq  dontinc  ; branch only if r16 equals 0xff
    inc   r16      ; if I get here, r16 does not equal 0xff, increment r16
dontinc:
```

Note that it does not matter what instruction is following the label `dontinc`. The `breq` instruction is simply specifying that the processor should proceed to execute code at the label `dontinc`, regardless of the actual instruction at that location.

Another common use of label is for variables. Variables are items in data memory (SRAM) that a program can read and change. For example, you may want to keep track of a counter in memory. You need the following instructions to add one to the variable `counter`.

```
    lds  r16,counter ; read counter into a register
    inc  r16         ; increment the register
    sts  counter,r16 ; store the new value to counter
```

14.3 Reserving Memory

For the previous code (incrementing variable counter) to work, you need to reserve a byte for `counter` *in SRAM*. Note that SRAM is in the data segment, and program code (instructions) is in the code segment. This means you need to define `counter` in another segment. The program should look like this:

```
.dseg
counter: .byte 1 ; reserve one byte
.cseg
    ...
    lds  r16,counter
    inc  r16
    sts  counter,r16
    ...
```

Note that `.byte` is an assembly *directive*. An assembly directive is not an instruction. This means it does not generate any machine code to be executed by the processor. Instead, it is a command intended tor the assembler to perform a particular function. In the case of `.byte`, it tells the assembler to reserve a number of bytes. The number of bytes to reserve is always supplied to the right of `.byte`. For example, to reserve 32 bytes, one can use the following directive:

```
    .byte 32
```

You do not need to define a label for each `.byte` directive.

14.4 Loading an Address to X, Y or Z

There is no instruction to load a 16-bit constant to the register pairs X, Y or Z. Instead, you need to use to ldi instructions to load an addres to a register pair. When you load an address, it is important to know which register contains the most significant byte, and which register contains the least significant byte. For X, register 26 contains the least significant (right most) byte, and register 27 contains the most significant (left most) byte.

Two built-in functions of the assembler allow you to extract just the most significant byte and the least significant byte of a 16-bit value. high is a function that returns the most significant byte of a 16-bit value, and low is a function that returns the least significant byte of a 16-bit value.

In our previous example (to add one to the variable counter), we can use the following instructions:

```
.dseg
counter: .byte 1
.cseg
   ...
   ldi   r26,low(counter) ; r26 is the least sig. byte of addr. of counter
   ldi   r27,high(counter) ; r27 is the most sig. byte of addr. of counter
   ; at this point, X is the address of counter
   ld    r16,X  ; X is the VALUE of counter
   inc   r16    ; increment the value
   st    X,r16  ; store the incremented value
   ...
```

15 Analogy for Addressing Modes

Consider a register as a post-it. The AVR has 32 of such post-its. Each post-it is an 8-bit item, it can store 8 independent bits.

Memory is a little booklet. Each page (memory location) in the booklet stores 8 bit. Furthermore, each page has a page number (an address). In the case of the AVR, a booklet can be as thick as 65336 pages. The first page has a page number of 0, while the last page has a page number of 65535.

When represented in binary, 65335 is 1111111111111111_2. This means it takes two post-its or two pages in the booklet to store an address.

15.1 ldi

The ldi instruction is a single instruction that specifies what to write to a register.

In other words, ldi r23,42 is saying "copy the *value* 42 to post-it r23".

Note that ldi does not require any memory location because it does not refer to the memory booklet at all.

15.2 lds

The lds instruction specifies a page in the booklet to read from, and copies the value to a post-it.

The instruction lds r23,4021 says "flip the booklet to page 4021, then copy the contents from that page in the booklet to post-it r23." Note that lds can specify any page of the 65336 (potential) pages in the booklet.

15.3 `ld`

The `ld` instruction pieces two post-its together, then read them as an address (recall how an address or page number requires two bytes). The instruction then reads from the location specified by this address and copies the value to a post-it.

It is more difficult to illustrate this instruction. Let us assume `r26` has a value of `0x4d`, while `r27` has a vlaue of `0x01`. This is necessary because `r26` and `r27` combine to become `X`.

The instruction `ld r14,X` does the following:

- combine `r26` and `r27` for a 16-bit value of `0x014d`.
- read from location `0x014d`
- copy the value to register `r14`

15.4 `st` and `sts`

`sts` is the counterpart of `lds`, and `st` is the counterpart of `ld`. The addressing modes are the same, only the direction is different. `lds` and `ld` copies a value from memory to a register (from a page in the booklet to a post-it), `sts` and `st` copies a value from a register (post-it) to a memory location (page in the booklet).

16 Register Arithmetics and Conditional Branches

This is going to be a long section because there are plenty of operations to operate on registers. To make this section more manageable, some of the more difficult operations are discussed in another section.

16.1 Status flags

A status flag is basically a boolean value stored in the processor. In other words, a status flag can only be true or false. A true flag is also said to be set or have a value of 1. Simiarly, a false flag is also said to be reset, cleared or have a value of 0. Most of the time, a status flag is used to indicate a particular property of the preceeding instruction.

For now, let us only be concerned with a few flags.

`Z` The `Z` flag stands for zero. This flag is set if and only if the result of the previous operation is zero. This is the same as saying the `Z` flag is set if and only if all the bits of the result of the previous instruction are zeros.

`C` The `C` flag stands for carry. This flag is set if and only if the previous instruction does not have enough bits for the result.

`N` The `N` flag stands for negative. This flag is set if and only if the previous instruction yields a negative result. A negative result always has the most significant bit (MSB) set to 1.

16.2 Adding and Subtracting

There are three instructions for adding and three instructions for subtracting.

`inc` The `inc` (increment) instruction adds one to a register and stores the sum back to the register. It requires one operand, which can be any of the 32 registers.

`add` This instruction is perhaps the simplest of all the add instructions. It adds the value of two registers and put the sum back into one of them. This instruction requires two register operands.

The first operand is also the destination. It provides one of the values to be added before the operation, and it *also* stores the sum after the operation. This operand can be any one of the 32 registers.

The second operand only supplies a value to add to the first operand. The second operand is not altered during the operation. This operand can be any of the 32 registers.

Although the `Z` and `N` flags are fairly self-explanatory, the `C` flag requires a little bit of explaining. Let us use the prefix `0b` to denote a binary number. The sum of `0b11000000` and `0b01000000` is 0b100000000. Note that the result has 9 bits instead of 8. Because a register only has 8 bits, the result is `0b00000000`. However, although the 9-th bit is not stored in a register, it is stored in the `C` flag.

`adc` This instruction is a little bit more complicated than the `add` instruction, although they almost the same. The `adc` instruction has two register operands, and the first operand also stores the sum. However, in addition to adding the two registers, the `adc` instruction also treats the `C` flag as a number and add that to the sum as well. Recall that the `C` flag can only be a 0 or a 1.

The main purpose of the `adc` instruction is to extend the width of an integer to more than one byte. In many applications, an 8-bit integer is not quite enough. With just 8 bits, an integer can contain values from -128 to 127 (when interpreted signed) or from 0 to 255 (when interpreted unsigned). If an application requires a larger range, 8-bit numbers are no longer sufficient.

The `adc` instruction allows the programmer easily implement the add operation for multibyte integers. For example, let R3:R2 denotes a 16-bit integer made from register 3 (as the most significant byte) and register 2 (as the least significant byte). Also let R5:R4 denotes another 16-bit bit integer. In order to add these two 16-bit integers and put the result into R3:R2, we can perform the following operations:

```
add R2,R4
adc R3,R5
```

The first instruction is `add` because we do not have any carry from a previous add. The first operation can set the `C` flag because the result may have 9 bits. The second operation is `adc` because we need to account for the carry from the addition of the least significant bytes. This approach is flexible, we can add additional `adc` instructions to handle even more bytes. For example, it will only take one `add` and three `adc` instructions to add two 32-bit integers (each integer stored in four registers).

Please note that we always add the least significant byte first, then move on to successively more significant bytes. This is similar to how people perform add calculations.

`adiw` The `adiw` instruction is not quite as general purposed as `add` and `adc`. The `adiw` instruction takes two operands.

The first operand is a register *pair* that provides a number to add and stores the result. The programmer should only specify the least significant register of the pair. Valid register pairs are R25:R24, R27:26, R29:R28 and R31:R30. This means `r24`, `r26`, `r28` and `r30` are the only valid choices for the first operand.

The second oeprand is a 6-bit immediate operand. This means the second operand must be a constant that is between 0 and 63.

Note that the register pair R27:R26 is `X`, R29:R28 is `Y` and R31:R30 is `Z`. The `adiw` instruction is particularly useful for incrementing addresses so that an `ld` or `st` instruction using either `X`, `Y` or `Z` can "skip" a fixed number of bytes.

`dec` The `dec` (decrement) instruction subtracts one from a register and stores the result back to the register. It expects one register operand, which can be any of the 32 registers.

`sub` The `sub` instruction is the counterpart of `add`. The second operand is subtracted from the first operand, the difference is stored in the first operand. Both operands must be registers.

`subi` The `subi` (subtract immediate) instruction is similar to `sub`, but with the following differences. The first operand must be a register from r16 to r31, and the second operand must be an 8-bit immediate constant.

`sbc` The `sbc` isntruction is the counterpart of `adc`. Again, the second oeprand is subtracted from the first operand. However, this time, the `C` flag is *also subtracted* from the first operand (recall the `C` can have a value of 0 or 1). The `sbc` operand allows a program to perform subtraction of multibyte integers, much like `adc` allows a program to perform addition of multibyte integers.

`sbci` The `sbci` instruction is similar to `sbc`, but with the following differences. The first operand must be a register from r16 to r31, and the second operand must be an 8-bit immediate constant.

`sbiw` The `sbiw` instruction is the counterpart of `adiw`. The first operand must be a register pair similar to the first operand of the `adiw` instruction. The second operand is an immediate constant much like the second operand of `adiw`. The second operand is subtracted from the first operand, the difference is stored in the first operand.

16.3 Adding an 8-bit Constant

It may appear that subtract instructions are not symmetric to add instructions. There is counterpart of add instructions to `subi` and `sbci`. This problem can be remedied by subtracting *negative* numbers.

In other words, if you need to add the constant of 5 to register r17, you can do this:

```
subi r17,-5
```

The assembler computes the two's complement form of 5 to represent -5.

16.4 Conditional Branches

A program that does not make decisions is not very useful. In other words, if all instructions of a program must execute from start to end, consecutively and only in one directioy, many complicated algorithms (methods) cannot be implemented.

The most basic decision making mechanism is called a condition branch. In this case, "branching" means continuing execution at an instruction location *possibly other than the immediately following instruction*. The location to branch to is usually the only operand a condition branch instruction requires. In order to make decisions, status flags are detected by conditional branch instructions. A condition branch instruction checks a status flag, then branches or continue execution depending on the value of the designated status flag.

This means a processor can only make very simple decisions based on whether a result is zero or not, negative or not, or whether the result carries. As we will see later, such simple decision making mechanisms can combine to very complicated decision making.

For now, we will only discuss three pairs of condition branches.

`brcc` and `brcs` `brcc` stands for "branch only if the `C` flag is cleared", and `brcs` stands for "branch only if the `C` flag is set". Both instructions require one operand to specify the instruction location to continue execution if the checked condition is true.

breq and brne brne (branch if not equal) really means "branch only if the Z flag is cleared", and breq (branch if equal) really means "branch only if the Z flag is set". Both instructions require one operand to specify the locaiton to continue execution if the checked condition is true.

brmi and brpl brpl (branch if plus) really means "branch only if the N flag is cleared", and brmi (branch if minus) really means "branch only if the N flag is set". Both instructions require one operand to specify the location to continue execution if the checked condition is true.

16.5 Comparison

Although the subtraction instructions can be used for comparison, they do modify the value of the first operand. Often, a program only wants to *test* which register is larger or whether they are equal, rather than actually taking the difference and store it into the first operand.

The compare instructions do exactly that. They compare two operands without modifying either after the comparison. However, The status flags are set *as if* the second operand is subtracted from the first.

cp The cp (compare) instruction is like the sub instruction, but cp does not store the difference back to the first operand.

cpc The cpc (compare with carry) instruction is like sbc, but cpc does not store the difference back to the first operand.

cpi The cpi (compare with immediate) instruction is like the subi instruction, but the difference is not stored back to the first operand.

17 Autoincrement

All register pairs, X, Y and Z can be used in ld and st with autoincrement. As discussed in 16.2 and 16.2, there are convenient instructions to increase or decrease the addresses stored in X, Y and Z by a constant. However, even these instructions are cumbersome in most cases because a program often only needs to increment or decrement an address by one.

The autoincrement and autodecrement feature allows the modification (either adding one or subtracting one) be performed in the same instruction. In other words, with these modifiers, the ld and st instructions can access memory (read or write) and advance the 16-bit address to the previous or next location all in one instruction.

17.1 Post Increment

If X, Y or Z is appended with a "+" symbol, the ld or st instruction increases the value of X, Y or Z by one *after* accessing memory.

For instance, the following two instruction can be replaced by ld r15,Y+.

```
ld   r15,Y
adiw r28,1
```

17.2 Pre Decrement

If `X`, `Y` or `Z` is preceded by a "-" symbol, the `ld` or `st` instruction decreases the value of `X`, `Y` or `Z` by one *before* accessing memory.

For instance the following two instruction can be replaced by `st r3,-X`.

```
sbiw r26,1
st   r3,X
```

17.3 Caution

Please understand that increment can only occur after memory access, and decrement can only occur before memory access. In other words, `ld r4,+X` is not permitted.

18 Homework Assignment (Assigned on 10/21, due in a week)

Write a program to reverse a sequence of bytes. Start with the following code:

```
.dseg
original: .byte 10
reversed: .byte 10
.cseg
```

Put a byte sequence (anything) into `original`, and `reversed` should have the reversed sequence after your code is completed. For example, if `original` has the sequence:

`01 23 45 67 89 ab cd ef 02 13`

`reversed` should have the following sequence:

`13 02 ef cd ab 89 67 45 23 01`

Study the program at http://www.drtak.org/teaches/ARC/cisp317/samples/ld_st_example.asm. Use it as a starting point.

You have to use either the `sbiw` instruction or to use the predecrement mode

Send the program to tauyeung@drtak.org. The subject of your email should be "CISP317 Project 2 by *your name*".

19 Assignment 2: String Comparison 200 points (due two weeks from 10/28)

19.1 What it does

In this assignment, you will write a program that compares two strings. A string is a sequence of bytes in which a byte represents a character. String comparison is useful in searching and sorting.

We assume strings are "null-terminated". This means the end of a string is indicated by a "null" character. Other than using the null character to mark the end of a string, a string has no other size limitation.

A string comparison subroutine indicates the result of the comparison using the `C` and `Z` flags. If the two strings are identical, the `Z` flag should be set. If the first string is "less than" the second string, the `C` flag should be set. If the first string is "greater than" the second string, the `C` and `Z` flags should both be cleared.

19.2 The Method (Pseudocode)

The following is the pseudocode for string comparison. I assume register `X` begins with the address of the first byte of the first string, and register `Y` begins with the address of the first byte of the second string.

while character at X is the same as character at Y and character at X is not null **do**
  X = X + 1
  Y = Y + 1
end while

Indeed, this is it!

19.3 The entire program

The program must contain a subroutine called `strcmp` (string compare) to perform the actual comparison. In addition, the program must initialize `X` and `Y` so they begin with some useful address in memory.

You must place the main program before the definition of the subroutine. This is because assembly programs simply execute from the first available instruction. In other words, your program should look like the following:

```
.dseg
string1: .byte 0x20 ; allocate 20 bytes for string1
string2: .byte 0x20 ; allocate 20 bytes for string2
.cseg
   ... ; code to initialize X
   ... ; code to initialize Y
   ... ; your code to compare strings
   nop ; extra instruction so we can stop here, examine C and Z here
```

There is no easy way to initialize SRAM locations with a string. What you need to do is to initialize those locations using the memory view window. Click on the byte to change and enter the hexadecimal code. Note that the memory view also has a “text” mode that displays the actual characters.

19.4 How to Turn in

Send the program (just the `asm` file) to my at tauyeung@drtak.org, with the subject indicating “CISP317 Project 3 by *your name*”.

20 The Basics of Programming Logic

As we have seen already, each individual operation that a computer can perform is very simple. Then how can computers perform operations that are complicated? By using the small building blocks, such as assembly instructions, a programmer can construct the representation of complex procedural logic.

20.1 Sequences

A sequence is the simplest procedural logic. Operations in a sequence are performed one by one in the specified order. In most programming languages, including assembly, operations follow the “natural” order of top to bottom, much like how people read English text. Some languages also allow multiple operations per line, read from left to right. For assembly programs, each line can only contain one instruction.

The key concepts of a sequence are as follows:

1. only one operation can be performed at any particular time
2. operations are specified in a top-to-bottom manner
3. operations are executed in the specified order

In assembly language, there is no particular instruction to specify sequences.

20.2 Uncondition branch

It is often necessary to repeat operations. For example, in order to surf TV channels, one must press the channel-up button on the remote control *repeatedly*. In order to repeat, a computer must be able to break out of a sequence to continue with an earlier step.

This action of breaking away from the normal (top-to-bottom, one-by-one) sequence is called branching. For example, the following snippet illustrates the assembly code to initialize multiple memory locations with the contents of register 16:

```
again:   st   X+,r16
         rjmp again
```

In this example, the `st` instruction stores the value of register 16 to where `X` points to, then it increments `X`. In addition, the `rjmp` instruction tells the processor to continue execute at the location of `again`. This leads the processor back to an earlier instruction, which in return makes the processor repeat the `st` instruction.

Of course, this snippet is flawed because there is no way to exit the loop! This code will execute forever!

20.3 Conditional branches

The previous snippet in 20.2 cannot exit the loop because there is no branch out of the loop. If we want to store the contents of register 16 to 32 consecutive locations in SRAM, we want to exit the loop when the `st` instruction has performed 32 times.

This 'want to exit when...' business can be performed by a conditional branch. A conditional branch *first* evaluates whether a condition is true, *then* it proceeds in one of two ways. If the condition is, indeed, true, the instruction proceeds to a predetermined location that the programmer can specify. If the condition turns out false, the instruction acts like a normal instruction and let the next instruction (the instruction below it) execute.

Conditions that a conditional branch can check for are mainly whether a flag is set or cleared. Such flags are usually set by another operation that occurs immediately before the conditional branch. For example, in our example, we can modify the code to the following:

```
         ldi  r17,32
again:   st   X+,r16
         dec  r17
         breq notagain
         rjmp again
notagain:
```

In this modified code, we initialize register 17 to 32. Note that this is performed *before* label `again` because we only need to set register 17 to 32 once. We also add the `dec` instruction in the loop to decrease register 17 by one every time the `st` instruction stores another byte.

The most important instruction is the `breq` instruction. This instruction checks to see if the `Z` flag is set. If the `Z` flag is set by the `dec` instruction, it means register 17 is decremented to zero. This means

we have performed 32 `st` operations. If the `Z` flag is set, control is passed to the instruction located at the `notagain` label. If the `Z` flag is cleared, the branch does not occur. Instead, the processor executes the instruction below `breq`, which is the unconditional branch back to label `again`.

20.4 Is that all to it?

Yes, and no. Although sequences, unconditional branch and conditional branches can be used to express any procedural logic, they are often considered too 'primitive'. In other words, if a programmer simply thinks in terms of unconditional and conditional branches, complicated procedures require a lot of effort to code in assembly.

Assembly instructions are basic and yet flexible. They do not impose or require any structure. Trying to build a program from assembly instructions is like trying to build a house from sand. The lack of structure makes it impossible to build anything that is useful.

In the next section, we will see how we can impose structure using pseudocode constructs. Pseudocode constructs are like frames and beams in the analogy of building a house.

21 Pseudocode and Assembly Code

Pseudocode means "not real code". The purpose of pseudocode is to enable a programmer write the *logic* of a program without worrying much about implementation details. This section introduces the four most basic pseudocode constructs, then discusses how to translate a pseudocode construct to assembly code.

21.1 hierarchical blocks

One of the most important concepts in pseudocoding is hierarchical blocks. This means a block can contain other blocks, and each of the contained blocks can contain even more blocks. There are many real-life analogies to hierarchical blocks. For example, the world is divided into countries, a country is divided into provinces, a province is divided into regions, a region has many cities, a city has different districts.

Being hierarchical, a programmer can write pseudocode using the top-down methodology. In other words, a programmer can focus on the overall behavior first, without worrying about how to do certain operations. Then each operation is broken into smaller operations. This process repeats until operations are small enough to be handled by a native programming language, such as assembly language.

Much like the File Explorer of Windows use indentation to represent the structure of folders, pseudocode also use indentation to represent how blocks are structured. Indentation is the amount of space on the left hand side. More indentation means there is more space to the left of the first non-space character. In pseudocode, if block A contains block B, the indentation of block B must be more than the indentation of block A.

21.2 if-then-else

The *conditional statement* is one of the most useful constructs, although it is a very simple concept. The sole purpose of a conditional statement is to express the following:

1. a condition to check, the result of evaluation must true or false
2. operations to perform only if the condition evaluates to true
3. operations to perform only if the condition evaluates to false

The general form of a conditional statement is as follows:

```
'if' condition 'then'
   then-block
'else'
   else-block
'endif'
```

In this notation, everything in quotes ('') should be used as is, and everything not in quotes should be replaced by actual code. For example, the logic to add two 16-bit integers may look like the following:

```
add low oder bytes
if there is a carry then
  add one to the result
else
endif
add high order bytes
```

Note that when the `else-block` is empty, you can leave out the 'else' word, resulting in a somewhat simpler pseudocode:

```
add low oder bytes
if there is a carry then
  add one to the result
endif
add high order bytes
```

21.3 as-long-as-do, aka while-do

The conditional statement does not repeat any operation. It is useful to perform some operations repeatedly for as long as a condition is satisfied. This is called a while-do statement. It consists of a condition to check for, and operations to perform while the condition is satisfied. It is important to note that the condition is checked *first*, and the operations are performed if the condition is true. After the operations are performed, a while-do statement always check the condition again to see if the operations should be performed again.

The general form of a while-do statement is as follows:

```
'while' pre-condition 'do'
    while-block
'endwhile'
```

The condition is called a 'pre-condition' because this is a condition that must be satisfied before the operation takes place. A while-do statement may look like a if-then statement, but they are different. I will illustrate with a real-life example.

Let us try to express the logic behind channel surfing. In English, one can say "as long as the current channel is boring, switch to the next one". The proper pseudocode to represent this logic is as follows.

```
while the current channel is boring do
  switch to the next channel
endwhile
```

It is not difficult for a beginner to confuse this with the following:

```
if the current channel is boring then
  switch to the next channel
endif
```

The similarity is due to both statements have a condition to check and a block to perform. However, the different is that in the case of if-then, the operation is performed only once if the condition is satisfied. In other words, it will limit the number of channel switching to at the most once. The while-do statement, however, repeats. Every time after switching a channel, the condition (whether it is boring) is reevaluated. If the new channel is also boring, we will switch channel again.

21.4 do-while

In a while-do statement, the condition is checked *before* the operations are performed. In a do-while loop, the operations are performed first, then the condition is checked. If the condition is satisfied, the program will go back and perform the operations again. Its general form is as follows:

```
'do'
    do-block
'while' condition
```

This means the do-block is performed at least once because the condition is checked afterward. Sometimes this may not be desireable. Let us reexamine the channel surfing example. Let us see what will happen if the code is rewritten as follows.

```
do
  switch to the next channel
while the current channel is boring
```

In this case, we always switch to the next channel first. This means if the current channel is already interesting right after turning on the TV, the program still switches the channel!

In general, there are more applications of while-do loops than do-while loops.

21.5 combining constructs

As mentioned earlier, we can put a block within another block. This is useful when we need to perform complex operations. Consider the following ordering procedure at a fastfood restaurant. "If a customer requests a combo meal, ask if the customer want to mega-size it. Otherwise, if a customer requests a kiddy-meal, see if a toy should be given. For a boy 10 or younger, give a Sgt. Bash'em action figure. For a girl 8 or younger, give a Pretty'n Silly Lily doll."

The logic can be written in pseudocode as follows.

```
if order is combo meal then
  ask if to mega size or not
else
  if kiddy-meal then
    if boy and 10 or younger then
      give Sgt. Bash'em action figure
    else
      if girl and 8 or younger then
        give Pretty'n Silly Lily doll
      endif
    endif
  endif
endif
```

In this example, you can see how indentation helps to indicate which portion of the program belongs to the same block. Furthermore, indentation also helps to match the 'if' keywords to the corresponding 'else' and 'endif' keywords.

22 Multiplication as an Example

The more expensive AVR MCUs do come with the multiply instructions. However, many processors, especially low-cost 8-bit types, do not have a multiply instruction. This means sometimes it is necessary to write explicit code to perform multiplication.

22.1 A quick reminder of decimal multiplication

Binary multiplication is not much different from decimal multiplication. For example, let us consider the product of 3178 and 79. Note that 3178×79 is the same as $3178 \times (70 + 9)$, which is also the same as $3178 \times 10 \times 7 + 3178 \times 9$. This is how we break up the multiplication of two multiple digit number for multiplication.

22.2 Binary multiplication

It turns out binary multiplication has easier rules when compared to decimal multiplication. The same general principle applies. For example, $1011_2 \times 110_2$ is $1011_2 \times (100_2 + 10_2 + 0_2)$, which then becomes $1011_2 \times 100_2 + 1011_2 \times 10_2$. This is hardly exciting.

What is $1011_2 \times 100_2$? This is *almost* the same as decimal. The answer is 101100_2. Because binary numbers can only have zeros and ones, there is no need to memorize any multiplication tables.

To return to our example, $1011_2 \times 110_2$ eventually becomes $101100_2 + 10110_2$, which is then simply added using binary addition.

22.3 Shifting

When we multiply 1011_2 by 100_2, it is as if we shift the original number to the left by two positions, while inserting two additional zeros to the right hand side. This operation is, indeed, called shifting. To be more specific, this is left shifting because the original number appears to be shifted to the left hand side.

As you can probably expect, $101100_2 \div 100_2$ yields 1011_2. This time, it is as if the number is shifted to the right hand side, as the least significant (right most) digits of the original number "fall off" the number. Zeros can be imagined inserted to the left hand side because inserting leading zeros do not modify the value of the number. This operation, as you probably know already, is called right shifting.

Almost all processors have left shift and right shift instructions. These instructions differ from our previous discussion a little bit. In *theory*, left shifting should never lose any digits as new zeros are added on the right hand side. On a processor, however, a register only has a fixed number of digits. As such, left shift operations actually lose the most significant (left most) digit of the original number. For example, with an 8-bit processor, left shifting 10101100_2 once yields 01011000_2 because the left most digit "falls off".

When a digit "falls off" as the result of a shift operation, it "falls" into the `C` flag. This feature is quite useful when we need to know the value of the bit that "falls off" from a shift operation.

22.4 Rotation

Rotation is related to shift, but with an important twist. In a shift operation, the "new" bit inserted to the right (for a left shift) or to the left (for a right shift) is *always* a zero. In a rotation operation, the "new" bit inserted is actually the value of the previous `C` flag. This does not stop a rotation operation to store the fallen out bit as the new value of the `C` flag.

For example, consider the 8-bit binary number 10110010_2. Let us assume the `C` flag currently has a value of 1. If we perform a left rotate operation, the value becomes 01100101_2, and the new value of the `C` is (still) 1 because the most significant bit shifted out is of value 1.

Rotation operations are useful because if we need to perform shifts on integers taking more than 1 bytes, we need to shift one bit from one byte to another.

22.5 AVR shift and rotate instructions

LSR `LSR` is logical shift right. This instruction takes a register as an operand. It performs an one-digit right shift on the value of the specified register. The original least significant bit is stored in the `C` flag. The operand can use any one of the 32 registers.

LSL `LSL` is logical shift left. It is the counterpart to `LSR`. It performs an one-digit left shift on the value of the specified register. This instruction takes any one of the 32 registers as its operand.

ROR `ROR` is rotate right. This instruction is very similar to `LSR` except for one thing. In `LSR`, the inserted bit to the most significant bit is *always* a zero. In `ROR`, the inserted bit to the most significant bit is the value of the `C` flag prior to the execution of the instruction.

ROL `ROL` is rotate left. This instruction is very similar to `LSL` except for one thing. In `LSL`, the inserted bit to the least significant bit is *always* a zero. In `ROL`, the inserted bit to the least significant bit is the value of the `C` flag prior to the execution of the instruction.

22.6 Shifting more than 8 bits

The builtin instructions of the AVR can only perform shifts on 8-bit registers. It is often necessary to perform shifts on larger quantities. With shift and rotate instructions, a program can perform shift operations on integers consuming any number of bytes.

For example, if we assume registers `r17:r16` (`r17` is the more significant byte, `r16` is the less significant byte) contain a number to be left shifted, we can use the following instructions.

```
lsl  r16
rol  r17
```

The first instruction, `lsl`, is hardly surprising. The second instruction, however, requires some explanation. Note that the first operation shifts the original most significant bit of `r16` into the `C` flag. The second operation, the `rol` instruction, rotates this bit (previously the most significant of `r16`) into the least significant bit of `r17`.

In order to perform a right shift on two 8-bit registers, we must operate on the most significant byte first. If the integer is stored in `r17:r16`, with `r17` being the most significant byte, then the following code shifts the 16-bit quantity to the right by one position.

```
lsr  r17
ror  r16
```

Note that this time we operate on the most significant byte first.

22.7 Multiply Pseudocode

In the absence of a multiply instruction, we can perform multiplication using the following logic.

```
product = 0
num1 = ...
num2 = ...
while (num1 != 0) do
```

```
  right shift num2
  if the old LSB of num2 is 1 then
    product = product + num1
  end if
  left shift num1
end while
```

22.8 The assembly code

You can go to http://www.mobots.com/ARC/CIS34/avrsamples/mult.asm for the actual assembly code to perform multiplication.

23 One Address Leads to Another...

23.1 The Stack Pointer

The AVR, unlike most other RISC computers, do not consider the stack pointer (SP) a register. This is because some of the lower end AVRs *do not have an data memory*! If the stack pointer is not a register, then what is it?

This is a good time to introduce the different types of memory. We'll come back to discuss how to initialize the stack pointer at the end of this section.

23.2 All Kinds of "Memory"

When we talk about "memory", most of the time we are referring to just memory. The AVR is capable of addressing $2^{16} = 65536$ locations in memory. *All* of these locations can be accessed via the memory access instructions `ld`, `st`, `lds` and `sts`.

Of our these 65536 locations, 96 locations are "special'. In hexadecimal, locations `0x0000` to `0x005f` are special. Within this portion, locations `0x0000` to `0x001f` belong to one group, and locations `0x0020` to `0x005f` belong to another group. The following sections describe how each group is different from the next.

23.3 `0x0000` to `0x001f`

This portion of memory is actually the registers. Location `0x0000` is register 0, location `0x0001` is register 1,... location `0x001f` is register 31.

This means `mov r13,r7` is the same as `lds r13,0x0007`, which is also the same as `sts 0x000d,r7`. All of the above instructions take the contents of register 7 (the same as location `0x0007`) and copy it to register 13 (the same as location `0x000d`).

Why do have the special instruction `mov`, then? It turns out the `mov` instruction is twice as fast but only half the size of the equivalent `sts` or `lds` instructions.

23.4 `0x0020` to `0x005f`

This portion of memory is called I/O memory. Each location in this region corresponds to some special device/hardware in the MCU. We'll discuss some of the more useful locations in later sections. However, for now, we just need to understand that the stack pointer is within this portion of memory.

Each location in this region can be accessed by `ld`, `lds`, `st` and `sts` instructions. However, just like the `mov` instruction can copy the contents of a register to another register more efficiently, there are instructions to access locations in this region more efficiently.

The `in` instruction loads the contents of a location from `0x0020` to `0x005f` to a register. Similarly, the `out` instruction stores a register to a location in the same region. *However*, it is important to note that when you use `in` and `out`, the addresses are *translated* to `0x00` to `0x3f`.

This means `lds r0,0x004b` is the same as `in r0,0x2b`, and `sts 0x0025,r16` is the same as `out 0x05,r16`. In the datasheet, the memory address (before subtracting `0x20`) is always in parantheses, where as the *I/O memory address* (after subtracting `0x20`) is *not* in parantheses.

23.5 Names of I/O Memory Locations

It is easier and more intuitive to use the symbolic names as defined in the datasheet than to use numeric addresses. However, it is a hassle to type in all the definitions. AVR Studio provides the definitions of all I/O memory locations (among other definitions) in "include files" (also called "header files").

An include file can be "included" in an assembly program file using the `.include` (the period is not a typo) directive. The following line instructs the assembler to take the contents of file `abc.inc` and "include" it in the program file *as if* it is a part of the program file:

```
.include "abc.inc"
```

You need to use double-quotes (`"`) to enclose the filename. The assemble assumes the file `abc.inc` is in the current directory in the previous example. If you want to include the definitions of all 8515 I/O memory locations, you need to let the assembler know the *absolute* location of the header files. At home, you probably want to use the following line:

```
.include "c:\Program Files\Atmel\AVR Studio\appnotes\8515def.inc"
```

In the lab (using the `I:` installation), you need to use

```
.include "i:\Auyeung T\Program Files\Atmel\AVR Studio\appnotes\8515def.inc"
```

Of course, you can always copy this file to the directory of your program, then you just need the following:

```
.include "8515def.inc"
```

23.6 Getting Back to the SP

Because the stack pointer can point to anywhere within the 65536 byte of memory, it requires 16 bits (2 bytes). The I/O memory locations are used to represent the stack pointer. The symbolic name `SPL` is the *I/O memory address* of the low byte of the stack pointer, the `SPH` is that of the high byte of the stack pointer.

Because `SPL` and `SPH` are defined (in `8515def.inc`) as I/O memory locations, you *must* use `in` and `out` instead of `ld`, `lds`, `st` and `sts`. The following code allocates 16 bytes for the stack, then initializes the stack pointer to the highest address:

```
.include "c:\Program Files\Atmel\AVR Studio\appnotes\8515def.inc"
.dseg ; switch to data segment to reserve data memory
stack: .byte 16 ; reserve 16 bytes
.cseg ; switch back to code segment to specify the program
   ldi   r16,low(stack+14) ; r16 is now the low byte of the end of stack
   out   SPL,r16 ; store it in the low byte of the stack pointer
   ldi   r16,high(stack+14) ; r16 is not the high byte of the end of stack
   out   SPH,r16 ; store it in the high byte of the stack pointer
   ; ... now we can use call and ret
```

24 The Stack

The stack is an important concept for all modern programming languages. This is a mechanism by which the `call` and `ret` instructions function. This section presents the stack and how it is used in the context of `call`ing and `ret`urning.

24.1 The Concept

The concept is simple: last-in-first-out. In other words, the very last item added to a stack is the first item to be removed. There are many real-life analogies for this. For example, consider a stack of dishes to be washed. The "top" item is the first item to be washed, whereas the "bottom" item is the last item to be washed. When you think about it, the "top" item is the very last item that you add to the stack of dishes.

24.2 The Terms

Just so that we can use technical terms from now on, let us first define them. Please note that these terms are often used outside of the context of assembly programming.

- Stack: a stack is a "container" that can contain many items. Items are added and removed using the last-in-first-out method.
- Top: the top of a stack is the next available location to add an item
- Push: to push an item is to add an item to the stack.
- Pop: to pop and item is to remove an item from the stack.

24.3 Subroutines

Consider the following code.

```
1:      nop
2:      call s1
3:      nop
4:      call s2
5:      nop
6:      ...
7: s1:  nop
8:      ret
9: s2:  nop
10:     call s1
11:     ret
```

This is what happens when this code executes:

1. 1 starts execution
2. 2 calls `s1`, it first remembers the address of the next instruction (at 3), then continues execution at `s1`
3. 7 executes
4. 8 executes, this recalls the address previously remembered, then continues execution there (3)
5. 3 executes

6. 4 executes, it first remembers the address of the next instruction (at 5), then continues execution at `s2`
7. 9 executes
8. 10 executes, this instruction first remembers the address of the next instruction (at 11), then continues execution at `s1`
9. 7 executes
10. 8 executes, this recalls the address previously remembered, then continues execution there (11)
11. 11 executes, this recalls the address previously remembered, then continues execution there (5)
12. 5 executes, this continues the program

But wait a second here, how do we sort out all the remembered addresses? In this program, we remembered 3, 5 and 11, how do we know where one to recall when we execute the `ret` instructions?

24.4 `call` and `ret` refined

Our existing explanation of `call` and `ret` is no longer sufficient to define the behavior in the previous sample program. We need to re-explain these instruction as follows.

- `call` *`label`*
  1. remembers the address of the instruction following `call`
  2. continues execution at *`label`*
- `ret`
  1. recalls *the most recently remembered* address
  2. "removes" this address from memory
  3. continues executation at the remembered address

The most important point here is "*most recently remembered*". Note that this is also the item to "remove" from memory. Doesn't this remind you of last-in-first-out? Indeed, addresses are "remembered" and "recaled" in a last-in-first-out manner. In other words, we use a stack to store and retrieve addresses for `call` and `ret`.

24.5 The Stack Pointer (SP)

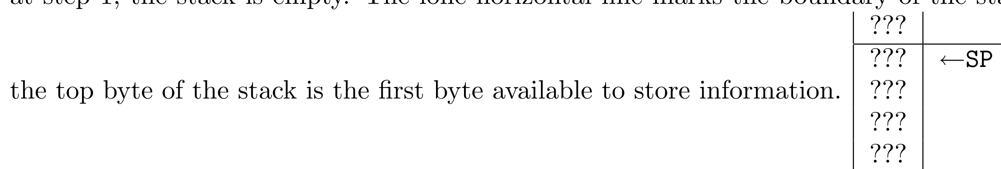
Unlike a stack of dishes, you cannot really remove memory locations. This said, how do we implement a stack? More precisely, how do we "remove" an item from a stack?

Just like you do not tear a page out when you are done reading it in a book, you do not actually remove a memory location after it is "used". You use a bookmark to mark the boundary between read and unread pages. Similarly, you use the stack pointer to mark the boundary between the portion of a stack that is in-use, and the portion that is already "used" and is no longer meaningful.

To be more specific, the stack pointer points to the *next available byte* to add an new item. This is the convention with the AVR processor family. Other architectures may have the stack pointer pointing to the *last used byte* instead.

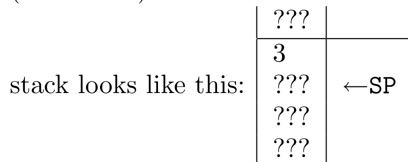
In the example discussed in section 24.3, the stack changes in the following sequence.

1. at step 1, the stack is empty. The lone horizontal line marks the boundary of the stack. Initially, the top byte of the stack is the first byte available to store information.



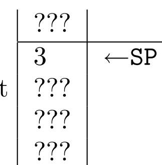
???	
???	←SP
???	
???	
???	

2. right after step 2, we "remember" the address of 3 on the stack. The stack pointer is "advanced" (downward) to remember that the original location is no longer available to put another item. The stack looks like this:



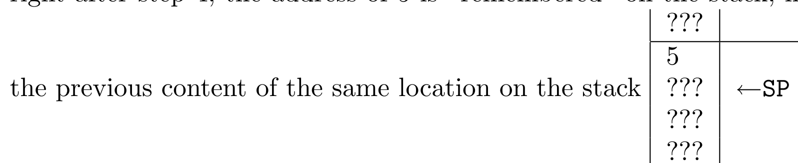
???	
3	
???	←SP
???	
???	

3. at step 3, the return address is "removed" by moving the bookmark above it



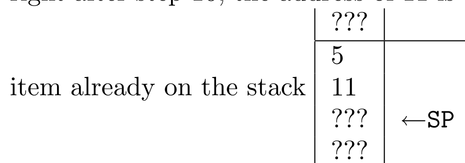
???	
3	←SP
???	
???	
???	

4. right after step 4, the address of 5 is "remembered" on the stack, note that this action *overwrites* the previous content of the same location on the stack



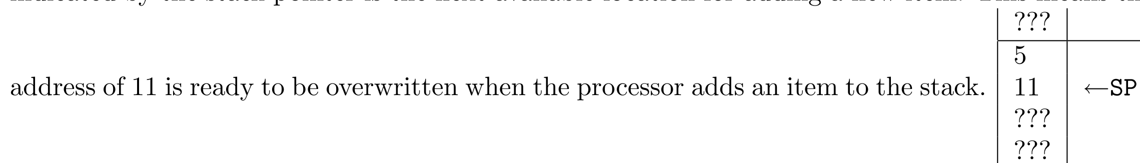
???	
5	
???	←SP
???	
???	

5. right after step 10, the address of 11 is also "remembered", but this is remembered "on top of" the item already on the stack



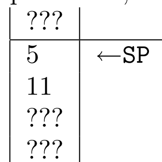
???	
5	
11	
???	←SP
???	

6. after step 8, the most recently remembered address is utilized and removed, note that the address of 11 is still on the stack, but it is now pointed to by the stack pointer. Recall that the address indicated by the stack pointer is the next available location for adding a new item. This means the address of 11 is ready to be overwritten when the processor adds an item to the stack.



???	
5	
11	←SP
???	
???	

7. after step 11, the most recently remembered address is utilized and removed. Note that the stack pointer is, once again, "behind" the boundary of the stack to indicate the stack is empty again



???	
5	←SP
11	
???	
???	

25 Subroutines

Very often, we write a portion of program that is useful in general. For example, after we write the code for division, we may need to use this code in different parts of a program. Our only option at this point is to copy and paste that code all over the program, wherever it is needed.

This section discusses a new technique to "reuse" code that is already written.

25.1 Example: 16-bit Integer Multiplication

Let us use an example for the rest of this section. In this example, we assume that we already know how to multiply using the code at http://www.mobots.com/ARC/CIS34/avrsamples/mult.asm.

Multiplying two 8-bit numbers to a 16-bit product is a very useful piece of code, especially on MCUs without a multiply instruction (such as the AT90S1200, AT90S2313 and etc.). In our example, let us try to write a program that multiplies two 16-bit number to get a 32-bit product.

Let us use `IJKL` to represent the first 16-bit number, in which each of `I`, `J`, `K` and `L` is 4 bits of the 16-bit number (with `L` being the least significant). Similarly, let `MNOP` be another 16-bit number.

Because we already know how to multiply two 8-bit numbers to get a 16-bit product, we can form groups of bytes, i.e., `IJ` and `KL` for the first number, and `MN` and `OP` for the second integer.

The product of the two numbers is $\mathtt{IJ} \times 100_{16} \times \mathtt{MN} \times 100_{16} + \mathtt{IJ} \times 100_{16} \times \mathtt{OP} + \mathtt{KL} \times \mathtt{MN} \times 100_{16} + \mathtt{KL} \times \mathtt{OP}$. This can be "simplified" to $\mathtt{IJ} \times \mathtt{MN} \times 10000_{16} + \mathtt{IJ} \times \mathtt{OP} \times 100_{16} + \mathtt{KL} \times \mathtt{MN} \times 100_{16} + \mathtt{KL} \times \mathtt{OP}$.

We need to perform four 8-bit to 8-bit multiplications in this computation. Without subroutines, we need to duplication the following code four times.

```
L1: cpi r18,0
    breq L2
    lsr  r18
    brcc L3
    add r20,r16
    adc r21,r17
L3: lsl r16
    rol r17
    rjmp L1
L2:
```

25.2 Copy and Paste, What's the Problem?

If we are to duplicate the code four times, we have at least one issue right away. The labels, `L1`, `L2` and `L3` are all visible throughout the same file. This means after we copy and paste, we need to alter the labels *and* references to the labels accordingly to the clones. If we forget to change one of the references, the program will still assembles, but the execution will be all wrong.

As you probably can guess, the copy and paste approach duplicates the code. In other words, we will have four almost identical copies of the same code stored in memory. For a small piece of code like this, it is generally not a big problem. However, what if the code to be copied and pasted is large? We'll be consuming a lot of memory to store multiple copies.

The most serious problem of this approach, however, is the difficulty to maintain this code. Imagine that I made a mistake in the multiply logic *before* copying and pasting. This means all the clones are also flawed. When I find this bug and want to fix the program, I will need to fix the same bug in all *four* clones of it. It is very easy for a programmer to forget fixing one or more of the clones.

It would be nice if we can somehow *refer* to the same piece of code at various points of our program. This way, even though the 16-bit to 16-bit multiplication needs to perform 8-bit to 8-bit multiplications four times, they can refer to exactly the same piece of code. This will not only save some code space

(because we are *referring* to a piece of code four times), it can also help debugging and maintenance. If I made a mistake in the 8-bit to 8-bit multiplication logic, I only need to fix it once.

25.3 A Subroutine

A subroutine is much like any fragment of code in your program. A subroutine always has an *entry point*, which is almost always represented by a label definition. The entry point is a name that allows the rest of the program refer to the subroutine. A subroutine also has an *exit* point which marks the end of the logic performed by the subroutine. The exit point is always marked by a special instruction called `ret` (for return).

If I want to make a subroutine for 8-bit to 8-bit multiplication, I can write it as follows.

```
mult8: ldi r17,0
L1:    cpi r18,0
       breq L2
       lsr  r18
       brcc L3
       add r20,r16
       adc r21,r17
L3:    lsl r16
       rol r17
       rjmp L1
L2:    ret       ; exit point is a ret instruction
```

25.4 Calling a Subroutine

The act to *refer* to a subroutine is technically called *calling* a subroutine. The sequence of events for calling a subroutine is as follows.

1. optionally set up before calling a subroutine
2. remember where to continue execution *after* the subroutine completes
3. transfer control to the subroutine
4. subroutine executions
5. subroutine completes
6. recall where to continue execution in 2, proceed to that point
7. continue execution

Steps 2 and 3 are often combined to one instruction, and they form the act of calling. On AVR processors, this instruction can be any one of `call`, `rcall` and `icall`. For now, we only use `call`.

Step 6 is called returning. As discussed already, it is performed by the `ret` instruction.

The most important point here is that the processor figures out and remember where to continue execution in step 2 on-the-fly when the program executes (i.e., at run-time). In addition, the address to recall for continuation in step 6 relies on the remembered address computed in step 2. This mechanism allows the program to call a subroutine from different places and still be able to return to the correct points in each case.

25.5 A Simple Example

Let us focus on a simpler example before returning to the 16-bit multiplication example. Consider the following program (line numbers added so we can refer to each line later):

```
1:      ldi  r16,32
2:      call add1
3:      mov  r0,r16
4:      call add1
5:      ...
6: add1:
7:      inc  r16
8:      ret
```

The sequence of execution is as follows.

1. line 1 initializes register 16 with the value 32
2. line 2 performs two functions, it remembers that line 3 is the next instruction, then transfer control to line 6
3. line 6 defines the entry point as a label called `add1`
4. line 7 adds 1 to register 16
5. line 8 recalls the location of the instruction that follows the `call` instruction that got us here, which is line 3 in this example
6. line 3 is where the subroutine returns to, *not* line 2! the program continues to execute here
7. line 4 calls `add1` another time, this time the processor remembers line 5 as the instruction after `call`, then transfers control to `add1`
8. line 6 again defines the entry
9. line 7 executes
10. line 8 recalls the location immediately after the most recent `call`, which is line `rest`, control is transferred there
11. line 5 marks the continuation of the program

It is important to note and understand that the `ret` instruction recalls the location to continue execution from “the most recent” `call`. This is why the program is able to return to line 3 for the first `call` and then return to line 5 for the second `call`.

25.6 16-bit Multiply

Given the `call` instruction, the 16-bit multiply logic becomes much simpler. Assume `mult8` is defined as in section 25.3.

```
; to multiply two 16-bit numbers for a 32-bit product
; assume r1:r0 is the first number, r3:r2 is the second number
; the product is stored in r7:r6:r5:r4
mov  r16,r0
mov  r18,r2
call mult8   ; compute KL * OP
```

```
mov  r5,r21  ; put result to r5:r4
mov  r4,r20
mov  r16,r0
mov  r18,r3
call mult8   ; compute KL * MN
add  r5,r20  ; add result to r6:r5
adc  r6,r21
mov  r16,r1
mov  r18,r2
call mult8   ; compute IJ * OP
add  r5,r20  ; add result to r6:r5
adc  r6,r21
mov  r16,r1
mov  r18,r3
call mult8   ; compute IJ * MN
add  r6,r20  ; add result to r7:r6
adc  r7,r21
; r7:r6:r5:r4 now contains the 32-bit product!
```

26 Pseudocode to Real Code

We have covered quite a bit of code to perform computation as well as data "pumping". Data pumping refer to the act of getting data from one source, then copy it to a destination (a sink). We have also discussed pseudocoding earlier to describe the overall logic of programs.

In this section, we explore methods to translate pseudocode to assembly code. This is very helpful to write larger programs.

26.1 Conditions

The key to all logic in programs is conditions. Conditions allow a program to make decisions. A decision, is in return, used to perform logical operations such as conditional statements and loops. As such, knowing how to code conditions is extremely important.

Simple Comparison A simple comparison is performed by a compare instruction, followed by a decision making instruction. If the numbers to be compared are stored in memory, they should be loaded into registers before using the `cp` instruction. If one side of the comparison is a constant (the value does not change as the program executes), then you may use a `cpi` instruction.

The `cp` and `cpi` instructions only compare 8-bit numbers. If you need to compare integers of more bytes, you can extend the compare with `cpc` instructions.

After the compare instructions, you can check the result. Different flags indicate different results. Here is a quick summary:

- `Z`: the `Z` flag is set if and only if the two operands are the same. This flag is set regardless of whether the numbers are interpreted signed or not.
- `C`: the `C` flag is set if and only if the first operand is less than the second operand when interpreted *unsigned*.
- `N`: the `N` flag is set if and only if the difference of the first and second operands is negative. This does not *imply* the first operand is less than the second operand when interpreted signed. For example, if the first operand is negative (`0x7f`), the second operand is negative (`0xff`), the difference is negative (`0x80`) even though the first operand is *not* less than the second operand.

- V: the V flag is set if and only if the difference of the first and second operand overflows when interpreted signed. This happens when the difference overflows. This happens when the difference of a positive number and a negative number is positive, or when the difference of a negative number and a positive number is positive. Note that the V flag does *not* imply that the first operand is less than the second operand. For example, if the first operand is positive (0x7f), the second operand is negative (0xff or -1), the difference is 0x80, which is negative. This sets *both* the V and V flags, although the first operand is greater than the second operand.

- S: as the previous two flags have indicated, the N flag and V flag do not, by themselves, reliably indicate the order of two signed numbers. The S flag is really a function of the N and V flag. The S is defined as the exclusive-or result of the N and V flags. This means in order for the S flag to be set, *one and only one* of the N and V flags is set. The exclusive-or symbol is often represented by $\oplus$. In other words, $\texttt{S} = \texttt{N} \oplus \texttt{V}$. The S flag is set if and only if the first operand is less than the second operand when interpreted signed.

Page 10 of the instruction datasheet details condition branches for use with both signed and unsigned numbers. Note that all of these instructions use the C, N, V and S flags. Also, note that some of the conditional branches utilize more than one flags. In fact, the BRLT instruction utilizes the Z, N and V flags. On this page, the symbol + is the same as "or" (disjunction), and the symbol • is the same as "and" (conjunction).

The following table illustrates the various conditions branches:

branches if and only if	mnemonic	cryptic flag condition
less than (signed)	BRLT	S=1
greater than or equal to (signed)	BRGE	S=0
equal (signed and unsigned)	BREQ	Z=1
not equal (signed and unsigned)	BRNE	Z=0
less than (unsigned)	BRCS or BRLO	C=1
greater than or equal to (unsigned)	BRCC or BRSH	C=0

26.2 Boolean and Branching

A condition is a boolean expression. In other words, the condition $X < Y$ is true if and only if X is less than Y. If X and Y are interpreted unsigned, then the C flag is set if and only if X is less than Y.

Conditions by themselves are indicators, but they do not control how a program executes. In order to make a decision based on conditions, conditional branches must be used. In our current example, BRCC or BRCS can be used to branch if the C flag is cleared or set, respectively.

The question is, then, when do we use BRCC, and when do we use BRCS? The real question is whether we want to branch only if $X < Y$ (BRCS), or branch only if $X \geq Y$ (BRCC).

The answer to this question depends on the context in which we use the condition. For now, let us classify the two branches as asserted-branch and negated-branch. In this context (X < Y), BRCS is called the asserted-branch because the branch occurs if and only if the condition is asserted (i.e., true). Naturally, BRCC in this context is called the negated-branch because the branch occurs if and only if the condition is negated (i.e., false).

From now on, we will use asserted-branch and negated-branch to refer to the type of conditional branch when a condition is evaluated. Furthermore, let us use the symbol $A(X < Y, label)$ to mean the asserted-branch to 'label' for '$X < Y$', and use the symbol $N(X < Y, label)$ to mean the negated-branch to 'label' for '$X < Y$'.

26.3 Compound Conditions

Although single conditions are useful, conditions are even more powerful when combined by conjunction (and), disjunction (or) and mutual-exclusion (exclusive or). For example, '$(A < B) and (B < C)$' is a

compound condition.

Just like we have asserted-branch and negated-branch for single conditions, we also have asserted-branch and negated-branch for compound conditions. In fact, the asserted-branch and negated-branch of a compound condition is composed of asserted-branches and negated-branches of the contained conditions.

Let us be more specific. Consider the compound condition '$(\texttt{r0} < \texttt{r1}) and (\texttt{r1} < \texttt{r2})$'. In addition, assume we want to branch to `l1` if and only if the compound condition is true. In other words, we are trying to derive $A((\texttt{r0} < \texttt{r1}) and (\texttt{r1} < \texttt{r2}), \texttt{l1})$.

There are many ways to do this. One way is to do the following:

1: $N(\texttt{r0} < \texttt{r1}, \texttt{l2})$
2: $A(\texttt{r1} < \texttt{r2}, \texttt{l1})$
3: `l2:`

In this code, we introduce a *local* label `l2`. `l2` is used only in this context and does not relate to `l1`.

Line 1 branches to label `l2` if and only if `r0` is greater than or equal to `r1`. This makes sense because if one of the components of a conjunction is not true, we can immediated continue execution at `l2` because the whole conjunction cannot be true.

Line 2 branches to label `l1` if and only if `r1` is less than `r2`. Note that if we ever get to execute here, we *already know* that `r0` is less than `r1`! As a result, as soon as we confirm that `r1` is less than `r2`, we know the whole conjunction is true, and we branch to `l1`.

Indeed, we can generalize compound conditions in abstract ways. In the following subsections, 'c1' and 'c2' are two conditions, 'l1' and 'l2' are two labels.

26.3.1 Conjunction

A(c1 and c2, l1) is
  N(c1, l2)
  A(c2, l1)
  l2:
N(c1 and c2, l1) is
  N(c1, l1)
  N(c2, l1)

26.3.2 Disjunction

A(c1 or c2, l1) is
  A(c1, l1)
  A(c2, l1)
N(c1 or c2, l1) is
  A(c1, l2)
  N(c2, l1)
  l2:

26.3.3 Mutual Exclusion

A(c1 eor c2, l1) is
  A(c1, l2)
  A(c2, l1)
  l2:
  N(c2, l1)
N(c1 eor c2, l1) is
  A(c1, l2)

N(c2, l1)
l2:
A(c2, l1)

26.4 Examples

Let us consider our favorite example, '$(\texttt{r0} < \texttt{r1}) and (\texttt{r1} < \texttt{r2})$'.

If we want to branch to label `l1` if and only if the condition is true, we need to first evaluate $A((\texttt{r0} < \texttt{r1}) and (\texttt{r1} < \texttt{r2}), \texttt{l1}$, which expands to

$N(\texttt{r0} < \texttt{r1}, \texttt{l2})$
$A(\texttt{r1} < \texttt{r2}, \texttt{l1})$
`l2:`

Then, we evaluate $N(\texttt{r0} < \texttt{r1}, \texttt{l2})$, which expands to

```
cp   r0,r1
brcc l2
```

We also evaluate $A(\texttt{r1} < \texttt{r2}, \texttt{l1})$, which expands to

```
cp   r1,r2
brcs l1
```

Now we combine everything as follows:

```
; N(r0 < r1, l2)
    cp   r0,r1
    brcc l2
; A(r1 < r2, l1)
    cp   r1,r2
    brcs l1
l2:
```

This is the final code for $A((\texttt{r0} < \texttt{r1}) and (\texttt{r1} < \texttt{r2}), \texttt{l1})$!

26.5 Long Compound Conditions

What if we need to deal with 'c1 and c2 and c3', or 'c1 or c2 or c3'? In these cases, you need to understand how we can associate the conditions. 'c1 and c2 and c3' is the same as '(c1 and c2) and c3', whereas 'c1 or c2 or c3' is the same as '(c1 or c2) or c3'.

Once you have the newer form, you work on the operations in a top-down fashion. In other words, the asserted-branch of '(c1 and c2) and c3' is

N(c1 and c2, l2)
A(c3, l1)
l2:

Then you break up 'N(c1 and c2, l2)' so that the overall code is as follows:

N(c1, l2)
N(c2, l2)
A(c3, l1)
l2:

You can try to write the code for N(c1 or c2 or c3, l1).

26.6 Pseudocode Translation

Once we understand asserted-branch and negated branch, pseudocode statements are quite easy to translate. The templates are listed in the following subsections.

26.6.1 if-then-else

if condition **then**
  body1
else
  body2
end if

Translates to

N(condition, `l1`)
body1
`rjmp l2`
`l1:`
body2
`l2:`

If there is nothing in 'body2', you can do the following:

if condition **then**
  body1
end if

Translates to

N(condition, `l1`)
body1
`l1:`

26.6.2 while-do

while condition **do**
  body
end while

Translates to

`l1:`
N(condition, `l2`)
body
`rjmp l1`
`l2:`

26.6.3 repeat-until

repeat
  body
until condition

Translates to

`l1:`
body
N(condition, `l1`)

27 Text Representation and Handling

27.1 Introduction

Although computers are famous for numerical computations (such as division, multiplication and etc.), they can also process “text” information. For example, when you order something online, the merchant’s web server needs to store and process your name and address.

What exactly is text information? In the context of a computer program, text information is something that can be represented by a chosen set of code. The most popular code is called ASCII (American Standard Code for Information Interchange). Don’t worry about its long name. What ASCII really is is a lookup table that maps *characters* to their binary code for storage purposes.

A character can be a letter, a digit, a punctuation mark, a space or a misc. symbol (such as the @ symbol). In ASCII, a character is encoded by 8 bits, or a byte.

For example, in order to represent “Tak”, we use the following bytes:

1. `01010100` for the letter T
2. `01100001` for the letter a
3. `01101011` for the letter k

Don’t worry about memorizing the ASCII table (256 entries!). A more immediate concern is: doesn’t `01101011` also represents the value 107 (base 10)? How do we know whether `01101011` is “k” or 107_{10}?

The answer is it depends on *how* this bit pattern is utilized. When you transmit this bit pattern to a printer, for example, the letter “k” is printed. In this case, `01101011` is interpreted as a letter. When you add `00010101` to `01101011`, you get `10000000` because 107+21 is 128, and 128 (unsigned) has a binary representation of `10000000`.

The bottom line is that in ASCII, a character can represent any one of the letters, digits, punctuations or special symbols as defined in the look up table. In addition, each character in ASCII is an 8-bit pattern.

27.2 Character Comparison

Just like you can order numbers in increasing or decreasing order, you can order letters in alphabetical or counter-alphabetical order. Given the numbers 185, 23, 156 and 44, the increasingly ordered sequence is 23, 44, 156 and 185. Similar, given the characters k, t, i, a and p, the alphabetically ordered sequence is a, i, k, p and t.

But, how does a computer know that “a” should precede “i”? Let’s look at the ASCII code of these two letters. “a” has a bit pattern of `01100001`, whereas “i” has a bit pattern of `01101001`. When you treat these bit patterns as unsigned numbers, `01100001` (decimal 97) is less than `01101001` (decimal 105).

This is how characters are compared! In a computer, the ordering of characters (not just letters, including digits, punctuations and other symbols) is simply *the same as the ordering of the unsigned number interpretation of the ASCII bit patterns*!

In ASCII, the space character (that produces a space on a printer) has a bit-pattern of `00100000`. This means lexigraphically, space precedes any upper case or lower case letters. Whether this makes sense or not is another question.

27.3 Strings

Although characters are useful, they are seldom useful individually. In order to spell a name, for example, requires a few characters. Such a sequence of characters is often called a “string”.

Given the start address of a string (the address of the first character in the string), how do we know where the end is? There are two ways to indicate the end of a string.

27.3.1 Length Delimited

In the case of a length delimited string, it has a *fixed length.* Regardless of the actual contents of this string, it always contains the same number of characters.

For example, for a length delimited string of 5 characters, it always uses 5 characters. How about string "Tak"? It should use up only three of the five available characters. In this case, the first three characters are used to represent "Tak". The remaining two (at the end) are "padded" with a padding value. The padding value is usually the space character (ASCII code 0x20).

27.3.2 Null Terminated

In the case of a null terminated string, it has a *fixed maximum size*, but a *variable length* that must be less than the maximum size.

The way this works is that instead of always interpreting the entire portion of memory allocated for a string, a program interprets only up to the "null" character. The "null" character has an ASCII code of 0x00.

In this case, if a string is allocated 5 bytes to represent "Tak", the first three characters will be "T", "a" and "k". The fourth character *must* be the null character. The last character, however, can be anything. This is because the program stops interpreting the string at the null character, so whatever is after it does not matter.

28 Pushing and Popping

In an earlier section, we discussed how the stack was used in subroutine calling and returning. Essentially, the stack served as a place where the processor can store (remember) addresses to return to. Although this is *one* way to utilize the stack, the stack is actually much more useful than this.

The basic nature of a stack is last-in-first-out (LIFO). This is a very useful feature when the processor just needs to "remember something and then go do something else".

28.1 An Analogy

Imagine you have a car that friends just like to borrow (like a truck among a farming community). Imagine that every time a friend borrows your truck, he/she changes all the settings: seat height, mirror angles, radio stations and etc. This upsets you a little because you have to spend a lot of time resetting all these settings.

To make the problem worse, your friend also lend the truck to his/her own friends. By the time your friend gets the truck back, settings are already changed!

To fix this problem, you include a little notebook in the car. This is what happens when a person borrows the truck:

1. flip a page in the notebook forward and move the bookmark to that page
2. erase contents on that page
3. write down all settings (seat height, radio stations, etc.)
4. change settings to his/her liking

When a person returns the truck, he/she must do the following:

1. flip to the previous page in the notebook and move the bookmark
2. restore settings as recorded on the current page (with the bookmark) of the notebook

Using this method, your friends and friends of your friends and etc. can keep lending and returning the truck. By the time it is returned to you, you are guaranteed to have all the personal settings restored to your original ones. Furthermore, *any one* who lends your truck to others also finds his/her settings restored.

Your problem is solved by, you guess right, a stack!

In this analogy, the notebook is the stack, the bookmark is the stack pointer. What is written to the stack? Typically, those are values of registers and the status flags. This is because the processor only has 32 registers. When you have subroutines calling each other, eventually all 32 registers will be used. When that happens, subroutines have no choice but to temporarily remember the "old" value of a register before using it, then return the "old" value so "no one knows" the register has been reused.

28.2 Example

Here is an example that can cause problems:

```
sub1:
    ldi   r16,0
sub1_loop1:
    cp    r16,20
    brcc  sub1_loop1_end
    call  sub2
    inc   r16
    rjmp  sub1_loop1
sub1_loop1_end:
    ret

sub2:
    ldi   r16,0
sub2_loop1:
    cp    r16,5
    brcc  sub2_loop1_end
    inc   r17
    inc   r16
    rjmp  sub2_loop1
sub2_loop1_end:
    ret
```

Both subroutines, `sub1` and `sub2`, use `r16` to count the number of times to iterate. However, since `sub2` is called from `sub1`, it destroys the value of `r16`. In other words, whenever `sub2` returns to `sub1`, the value of `r16` is *always* changed to 5. This cannot be good for `sub1` because it expects `r16` be changed from 0 to 20 incrementally.

What is the root of the problem? The problem is *not* that both subroutines want to use `r16`. Afterall, we only have 32 registers to use. At some point, a register has to be reused. The problem was that `sub2` made the invalid assumption that "no one else is using `r16`". It is wrong of `sub2` to just start to change the value `r16`. Of course, the same argument applies to `sub1` because whoever calls `sub1` *may* be using `r16` for some purpose.

To fix this problem, we need to introduce two new instructions:

- `push` *`r`* copies the value of *`r`* to where `SP` points to, then decrements `SP` by one. *`r`* can be any of the 32 registers.

- `pop r` increments the value of `SP`, then copies the content where `SP` points to to `r`. `r` can be any of the 32 registers.

In vague terms, `push r16` remembers the value of `r16` on the stack, `pop r16` recalls the value of `r16` from the stack.

In order to fix our problem, we add `push r16` at the beginning of each subroutine, then add `pop r16` at the end of the subroutine. This way, to the caller, a subroutine does not change the value of r16. The modified code is as follows:

```
sub1:
    push  r16
    ldi   r16,0
sub1_loop1:
    cp    r16,20
    brcc  sub1_loop1_end
    call  sub2
    inc   r16
    rjmp  sub1_loop1
sub1_loop1_end:
    pop   r16
    ret

sub2:
    push  r16
    ldi   r16,0
sub2_loop1:
    cp    r16,5
    brcc  sub2_loop1_end
    inc   r17
    inc   r16
    rjmp  sub2_loop1
sub2_loop1_end:
    pop   r16
    ret
```

28.3 Saving and Restoring Multiple Registers

What if we have more than one registers to save and restore? For example, if `X` is used in a subroutine, we should save it at the entry point and restore it when the subroutine returns. There is no `push X` because `push` only works for 8-bit individual registers. Okay, we need to use `push` instructions then. To save `X`, we need to use the following:

```
    push  r26
    push  r27
```

This works fine because `X` consists of `r16` and `r27`. The *order* of these two instructions is not important. This particular order is used because the AVR is a *big endian* machine, where the first (in position) byte is the most significant byte. We follow the same order so we do not get ourselves confused.

Here comes the tricky part: how do we restore the value of `X`? *Given* the order of the `push` instructions, the `pop` instructions must follow the opposite order. In other words, we need the following instructions to restore `X`:

```
pop   r27
pop   r26
```

This is because of the last-in-first-out nature of a stack. When we execute the `pop r27` instruction, `r27` is the last item placed on the stack, *not* `r26`! When `r27` is popped (removed from the stack), `r26` becomes the last item placed on the stack.

As a consequence, any subroutine that uses multiple registers must save and restore used registers in opposite sequences.

29 Arrays in Assembly

An array is a collection of objects of the same type. In assembly programming, the assembler cannot keep track of the type or size of objects. It is, therefore, the responsibility of the programmer to make sure all objects in an array are interpreted as the same type and have the same size.

Individual objects in an array are distinguished by their "indices" (index as singular). The first item has an index of 0, the second item has an index of 1 and etc.

Given that A is the address of the first byte of an array, i is the index of an object in the array, and s is the size of each object, the address of the object at index i is $A + s \times i$. For example, if `strArrayBegin` is the label that marks the beginning of an array, and each item has a size of 16 bytes, the item of index 4 is at address `strArrayBegin+4*16` or simply `strArrayBegin+64`.

29.1 Uses of Arrays

Arrays are very useful constructs, not only in assembly programming, but in all programming languages. In order to use arrays to organize a bunch of objects in memory, you need to fulfil the following requirements:

1. objects in the "bunch" must have the same type and size
2. each object must be identifiable with a non-negative integer
3. the maximum number of objects must be known ahead of time

If the "bunch" of objects satisfies these requirements, you can use an array to organize them. The following are benefits of using arrays to organize objects:

1. "random access", it is quick to access any object in the bunch
2. "sequential access", it is quick to access neighbors of an object, either the previous or the next
3. fixed size, the array has a known size based on the maximum number of objects

29.2 Implementation of Arrays

Because assembly language has no concept of type, the implementation of arrays is rather flexible. In general, if the size of each object is known, the size of the array is the product of the size of each object and the maximum number of objects.

For example, let's say a first name cannot have more than 15 characters. We can, then, reserve 16 bytes for each first name (16 because we need one more byte for the null character). Assuming a class has no more than 30 students, then the array to store first names of students in a class has a size of $30 \times 16 = 480$ bytes. The following code fragment reserves space for this array. It also defines a symbol `firstnames_end` for the address of the byte immediately past the last byte of the array.

```
.dseg ; switch to data segment
firstnames: .byte 30*16 ; reserve 480 bytes
firstnames_end:
...
```

It is, however, better to use symbols to represent the number of characters per firstname and the maximum number of students. You can use the `.equ` directive to define symbolic names. The following lines define the appropriate symbols for our array:

```
.equ maxStudents = 30
.equ firstnameSize = 16
```

With these symbolic definitions, we can change the reservation directive:

```
.dseg
firstnames: .byte maxStudents * firstnameSize
firstnames_end:
```

This is nice because in case we want to change the number of students in the class or the maximum number of characters for each first name, we simply have to change the definitions of the symbols, *not* locating all the `30` and/or `16` throughout the program.

29.3 Computing Addresses

Given the address of the first byte of the array and the index of the object that you want to access, you need to use multiplication *and* addition to computer the address of the object. This can be rather cumbersome. The following code is an example of computer the start address of an object at an index stored in `r0`, given the size of each object is defined by the symbolic name `itemsize` and the start address of the array is defined by the symbolic name `arraystart`.

```
    ldi  r24,itemsize
    mov  r25,r0
    call mult
    ldi  r16,low(arraystart)
    add  r24,r16
    ldi  r16,high(arraystart)
    adc  r25,r16
```

When this code completes, `r25:r24` is the address of the object at an index indicated by `r0`. This code assumes `mult` is a subroutine that computes the product of `r24` and `r25`, and stores the product back to `r25:r24` (as a 16-bit product). Referring to an earlier section, the multiplication code is as follows (with register saving and restoring):

```
mult:
    push r16
    push r17
    push r18
    ldi  r16,0 ; LSB of product
    ldi  r17,0 ; MSB of product
```

```
    ldi  r18,0 ; MSB of r18:r25
mult_l1:
    ; N(r24 <> 0, mult_l2)
    cpi  r24,0
    breq mult_l2
    ; r24 <> 0
    lsr  r24
    brcc mult_l3 ; skip adding if 0
    add  r16,r25
    adc  r17,r18
mult_l3:
    ; left shift r18:r25
    lsl  r25
    rol  r18
    rjmp mult_l1
mult_l2:
    mov  r25,r17
    mov  r24,r16
    pop  r18
    pop  r17
    pop  r16
    ret
```

As you can see, computing the address of an object in an array can be quite time consuming (the `mult` subroutine is long).

29.4 Neighbors

As discussed earlier, the address at index i is $A+s\times i$. Similarly, the address at index $i+1$ is $A+s\times(i+1)$. The address at index 0 is $A+s\times 0=A$. This means *given* the address at index i is A_i, the address at index $i+1$ is simply $A_{i+1}=A_i+s$. This makes sense because each object in an array is s from the next or the previous item, where s is the size of an object.

This property makes it easy and inexpensive to compute addresses of neighboring objects in an array. For example, in our example, if we need to do something with each object in the array *sequentially*, we can use the following code shell:

```
    ; initialize X
    ldi  r26,low(firstnames)
    ldi  r27,high(firstnames)
l1: cpi  r27,high(firstnames_end)
    brcs l1a ; less than, do the loop body
    brne l2  ; greater than
    ; MSB of X equals MSB of end address
    cpi  r26,low(firstnames_end)
    brcc l2 ; greater than or equal to
l1a:
    call dosomethingwithX
    adiw r26,firstnameSize
    rjmp l1
l2:
```

In this code, subroutine `dosomethingwithX` should do something with the register-pair `X`, but it should not modify `X`. Note that we don't need to use `maxStudents` because `firstnames_end` marks the end of the array, and this code uses this label to know when to exit the loop.

Whenever possible, the more general method of multiplication should be replaced by the much more efficient method of addition.

30 The Frame

Although we can use the registers to facilitate the communication between a caller and the called subroutine, registers are limited. There are only 32 registers with the AVR architectures. As a result, using registers to pass information is not sufficient in some cases.

Passing information via registers does have certain advantages. For example, registers are easy to set up and they do not require any memory space. Furthermore, access registers is easy because many instructions work with registers directly.

This section describes an alternative method to pass information between the caller and the subroutine it calls. The method is quite general and is used by most high-level programming languages.

30.1 The Stack as a Medium

The stack, as in the stack used in `call`, `ret`, `push` and `pop`, is a medium that is accessible to both the caller and the subroutine called. This is because the stack is an area of memory that both the caller and callee (the called subroutine) have access to. The stack is a good medium because although the absolute value of the stack pointer (SP) may vary when a subroutine is called, we can always use location *relative* to the stack pointer.

Let us first review what kind of communication occurs between a caller and a callee:

- the caller may have parameters to instruction the callee what to process or how to process
- the callee may have returned values as a result of some computation

We can combine these two functions so we can summarize what the stack will be used for:

- store parameters
- store return values
- store the address of the instruction following the `call` instruction
- store values of registers that will be changed in the subroutine

30.2 The `strlen` Subroutine as an Example

Let us use the `strlen` subroutine as an example. For this subroutine, the caller provides the beginning address of the string (to be counted), whereas the callee returns an integer that indicates the number of non-null characters in the string.

To begin with, the caller first specifies the beginning address of the string on the stack. Assuming the string starts at the label `str`, this is done by the following instructions:

```
ldi   r16,low(str)
push  r16
ldi   r16,high(str)
push  r16
```

The order (pushing the low byte first) is determined so it is consistent with the big-endian nature of the AVR processor. If there are more than one parameters, the *last* parameter is pushed first, and the first parameter is pushed last.

After all parameters (in this case, one parameter) are (is) pushed, the caller reserves space for the returned value. For `strlen`, the returned value is a 16-bit unsigned number. As a result, we need to reserve two bytes. Reserving two bytes on the stack is easy. We can simply push some register twice on the stack. Note that the *value* of a returned value should be specified by the callee anyway, so whatever value the caller initializes the returned value to is irrelevent.

```
push  r16
push  r16
```

At this point, the caller has done all it can in this call. The call instruction is executed next.

```
call  strlen
```

At this point, control is transferred to the callee. As usual, the first thing the callee should do is to save all the registers that will be changed later on.

```
push  r26
push  r27
push  r24
push  r25
psuh  r16
in    r16,SREG
push  r16
```

Note that `r24` and `r25` are pushed in this case. This is because we are now using the stack to return a value, not via the register pair `r25:r24`.

After the last `push r16` instruction, we are *finally* ready to execute the logic of `strlen`. Let us recount what items are now on the stack:

code	remarks	displacement
`ldi r16,low(str)` `push r16`	low order byte of the beginning address of the string to be counted	13
`ldi r16,high(str)` `push r16`	high order byte of the beginning address of the string to be counted	12
`push r16` `push r16`	just taking up two bytes on the stack, the values of these two bytes should be overwritten by the callee anyway	10
`call strlen`	calling the subroutine, saving the address (two bytes) of the following instruction	8
`push r26` `push r27`	save the current value of `X`	6
`push r24` `push r25`	save the current value of `r25:r24`	4
`push r16`	save the current value of `r16`	3
`in r16,SREG` `push r16`	save the status register	2
`push r30` `push r31`	save the frame pointer before we change it	0

31 Efficient I/O Instructions

Although most input/output activities can be handled by `in`, `out`, `and` and `or`, the AVR architecture offers much more efficient methods to handle single bits. This section describes each of these efficient instructions and illustrate how to utilize them.

31.1 `sbi`

`sbi` means "set bit in I/O memory location". This instruction requires two operands. The first operand is a number from 0 to 31 (or 0x00 to 0x1f in hexadecimal). The first operand specifies an I/O memory location (*not* general memory location). Note that there are 64 I/O memory location (from 0x00 to 0x3f). This means only half of these locations can utilize the `sbi` instruction.

The second operand is a number from 0 to 7. This operand specifies a bit position. Remember that bit 0 is the rightmost bit (also called the least significant bit, or LSb).

When the instruction executes, it accesses the I/O memory location specified by the first operand, and set only the bit position specified by the second operand. No other bit position is changed in the process.

For example, we can replace the following code

```
in    r16,PORTA
ori   r16,0b00100000
out   PORTA,r16
```

with a single instruction as follows:

```
sbi   PORTA,5
```

The `sbi` instruction is 50% faster (2 cycles versus 3) and 66% smaller (1 word versus 3 words) than the equivalent code. In addition, the `sbi` instruction does not need a temporary register for bit manipulation.

31.2 `cbi`

`cbi` is similar to `sbi` (see section 31.1). except that this instruction *clears* the bit position specified by the second operand in the I/O memory location specified by the first operand.

For example, we can replace the following code

```
in    r16,DDRA
andi  r16,0b11111011
out   DDRA,r16
```

with a single instruction:

```
cbi   DDRA,2
```

The advantages of `cbi` is similar to those of `sbi`.

31.3 `sbis`

We often have to check a single bit in byte for embedded applications. The AVR offers two instructions to handle this. `sbis` stands for "skip if bit in I/O memory location is set". This may sound a little strange, but its operation is fairly straight forward.

`sbis` requires two operands. The first operand is a number from 0x00 to 0x1f that specifies a memory location. The second operand is a number from 0 to 7 that specifies a bit position.

The instruction "skips" the following instruction if and only if the specified bit position in the specified I/O memory location is set. This means this instruction continues to execute the following instruction if and only if the specified bit position in the specified I/O memory location is cleared.

Consider the following code:

```
1:    sbis  PINA,3
2:    inc   r16
3:    ...
```

If bit 3 of PINA is indeed set, the execution path is 1 and 3. If bit 3 of PINA is cleared, the execution path is 1, 2 and 3. The equivalent code is as follows:

```
    in    r17,PINA
    andi  r17,0b00001000
    brne  l1
    inc   r16
l1: ...
```

Note that the alternative code needs to use `r17` as a temporary register. The alternative code also needs to utilize a label, whereas the `sbis` version does not need any label.

31.4 `sbic`

`sbic` is similar to `sbis` (see section 31.3). `sbic` skips the following instruction if and only if the specified bit in the specified I/O memory location is cleared.

Assignments

- 19, assigned on 10/28/2003, due on 11/11/2003
- 11, assigned on 09/04/2003, due on 9/16/2003
- 18, assigned on 10/21/2003, due on 10/28/2003