PROJECT BIOGEN

Symantec 2006 Science Buddies Assembly Language Programming Project

Table of Contents

Project BioGen

The year is 2020 and all is not well. The world's accelerating consumption of oil during the 2000's and 2010's has largely depleted the world's reserves of oil, leading to mass shortages. Scientists and engineers, the world round, are looking for environmentally safe and cheap alternatives to oil. The problem has become so large, and the opportunity so immense, that Symantec has decided to direct its Nano-technologies division to try to tackle the energy problem.

You have been hired by Symantec Nano-technologies Division to work on a new programmable nano-organism (NANORG) that is capable of converting industrial sludge into a renewable energy source. Symantec has already developed nano-organisms with the proper hardware to extract energy from the sludge, but needs you to write the control logic for the 16-bit NANORG CPU to make it as efficient as possible at harvesting energy. Of course, for your organism to function, it must use a portion of the energy it generates for its own operation, so it needs to balance its own energy needs with its goal of outputting as much energy as possible.

Once you have written the control program you can test how your NANORGs perform in a virtual world. In this simulation, fifty NANORG clones containing your logic will be dropped into a large tank of sludge and must consume the sludge and release energy into special collection points in the tank. Other entrants will be dropped into similar isolated vats of sludge and evaluated as well; your NANORGs will never interact with other entrants' NANORGs. Obviously, the NANORG that produces the most energy in the allotted time will be commercialized by Symantec, so it's your job in this project to produce the most efficient organism possible.

Unfortunately, in addition to converting sludge into electricity, your organism must also confront several other challenges. First, about 20% of the sludge is contaminated with radioactive chemicals; eating this sludge will cause your NANORG to undergo small mutations to its programming logic. Second, and more concerning, your organism must deal with an earlier generation of malicious NANORGs that are also present within the sludge tanks. These organisms, called "drones," were written by a spy to sabotage Symantec's Nano-technologies Division and are based on the same exact hardware and machine language instruction set as your NANORGs. However, their programming logic was intentionally designed to be harmful. While they also consume sludge, they produce no energy output. Moreover, if they come into direct contact with a foreign organism (such as yours), they may attempt to spread their logic by copying it into the adjacent organism (mutating your logic until you become one of them!). Your NANORG's logic must cope with both of these challenges.

Project Details

You will write an assembly language program using Symantec's NANORG assembly language to control an individual NANORG. To write your program, you should use a

standard text editor like Notepad, Crimson Editor, Emacs, etc. You can then use the accompanying CONTEST06 executable file to assemble and test your NANORG logic.

When you run the contest executable, it will compile your NANORG source code into NANORG machine language and then create 50 NANORG clones, each with a copy of your logic. Each of these NANORGs function as individual entities, and each has its own copy of your program, its own memory, registers, etc. This colony of 50 NANORG clones, each running your program logic, will be placed in the virtual tank and must harvest energy.

Your colony of NANORGs will have a total of 1 million timer ticks to generate as much energy as possible from the sludge in their tank.

At the start of the program, your 50 clones will be randomly placed into a 70 wide by 40 high tank of sludge with 20 clones of the malicious, previous generation of NANORG drones. Each NANORG, including drones, starts out with 10,000 units of energy. The tank's 70x40 locations are numbered from 0-69 and from 0-39, with the position 0,0 being in the north-west corner of the tank.

In a given tank, there will be hundreds of chunks of sludge. Every time a NANORG eats a chunk of sludge, it receives 2,000 units of energy. Each chunk of sludge is a member of a specific "sludge type" which is identified by a non-zero integer value (e.g. 5, or 17). Your NANORGs can sense the type of each chunk before eating the chunk.

Some types of sludge are entirely safe for a NANORG to eat and will simply provide it with energy, while other types of sludge are toxic (but still provide it with energy). For example, there might be 200 total chunks of sludge of 5 different types in the tank: 40 of type 1, 60 of type 2, 50 of type 3, 30 of type 4, and 20 of type 5. All 40 chunks of sludge type 1, 60 chunks of type 2, 50 chunks of type 3, and 30 chunks of type 4 might be safe to eat, while all 20 of type 5 might be toxic to your NANORG and cause mutations to its memory.

The number of different sludge types, and the quantity of each type of sludge, varies from tank to tank. In one tank there might be just 5 different types of sludge (with IDs 1-5), while in other tanks there may be up to 32 different types of sludge (with IDs 1-32). If there are N different types of sludge in a particular tank, then approximately **20%** of the N different types of sludge will be toxic to your NANORGs. For example, if there are 10 different types of sludge (with IDs 1-10), then exactly 2 different types of sludge will be toxic (e.g. ID #7 and #9 will be toxic, while IDs 1-6, 8, and 10 are safe to eat). All food of a given type is guaranteed to either be toxic or safe to eat (it will never be the case that some food of type 5 is safe while other food of type 5 is toxic). Which specific types of sludge are toxic in a given tank is determined randomly and varies from tank to tank.

If your organism eats a chunk of sludge which is of the toxic type, it will cause *one word (16 bits)* of your organism's memory to mutate. The earlier generation of NANORGs are completely immune to the mutagenic effects of the sludge and do not become mutated when eating toxic types of sludge.

You can determine the ID number of a given chunk by having your organism use the SENSE instruction while in the same square as a chunk of sludge. Your organism can learn which types of sludge are toxic and avoid them if you like.

Each time one of your NANORGs or one of the earlier, malicious generation of NANORGs eats a chunk of sludge, a new chunk of the same type of sludge (i.e. with the same type and therefore of the same toxicity) will be dropped in a random location of the tank automatically, creating a perpetual source of energy.

To gain points, each of your 50 NANORGs must eat sludge to gain energy and then release this energy at designated collection points using the RELEASE instruction. Your final score will be equal to *the total of amount of energy released from the tank* at the end of 1 million ticks. Each organism may not exceed 65535 energy units at a time, limiting how much each organism can eat before burning off the energy by executing instructions or using the RELEASE instruction to release energy or the CHARGE instruction to charge a neighboring NANORG.

To release energy and *gain points*, a NANORG must first travel to a *collection point* in the tank and then use the RELEASE instruction. 10 different collection points are randomly distributed within the tank. Your organism can sense when it is on top of a collection point by using the SENSE instruction, and these collection points are displayed on-screen with a $ character. Be careful not to release too much energy at a time or your NANORG may run out of energy it needs to operate before it can find more food! Attempting to release energy when not on a collection point will drain your organism's energy without increasing your score, so be careful!

If a NANORG runs out of energy, it doesn't die but instead hibernates. An active NANORG can reawaken a hibernating NANORG by sharing energy with it using the CHARGE instruction.

You will be programming your NANORG using a simple assembly language, which will be compiled into NANORG machine language for execution (the drones were also written in this assembly language and use the same machine language as your NANORGS). The executable Symantec provided for the project has a simple assembler (as well as a disassembler) for this language. Your NANORG program can be up to 3600, 16-bit words long, and should never terminate; it should continuously hunt for food and produce energy until it is finally terminated at the end of the simulation.

Each NANORG (including each drone) gets to execute a single machine language instruction per tick of the simulation. Each instruction consumes either 1 unit of energy (for computation) or 10 units of energy (to travel to an adjacent square in the tank).

In addition to traditional assembly language instructions, like MOV (to move data), and JMP (e.g. GOTO), the NANORG assembly language also includes other instructions to control the NANORG. Your organism can use the TRAVEL instruction to travel through the tank,

the EAT instruction to eat sludge, the SENSE instruction to determine if your organism is standing on sludge or on top of a collection point (each type of sludge has a different ID number between 1 and 32, and collection points have an ID of 65535), and the RELEASE instruction to release energy into a collection point. Furthermore, organisms have special instructions to interact with adjacent organisms. The PEEK and POKE instructions can be used to read/write to/from the memory of an adjacent organism (including both your NANORGs and drones). The CHARGE instruction can be used by a NANORG to transfer a portion of its energy to an adjacent organism.

The NANORG virtual machine used by both your NANORGs and the drones has exactly 3600, 16-bit words of memory, numbered 0 through 3599. The NANORG assembly/machine language instructions are only capable of reading/writing a word at a time, and cannot read and write individual bytes. Attempts to access memory outside the 3600-word range will return a value of 0 but will not cause your NANORG program to terminate. Since your organism's data and logic are stored within the 3600 words of memory, your program may use self-modifying techniques (i.e. a NANORG can read/write/modify its own instructions). The CPU will happily execute data as if it were an instruction, so long as it is properly encoded. The NANORG machine only supports UNSIGNED arithmetic; the processor does not have any support for signed/negative numbers.

All instructions in the machine language are exactly 3 memory words long, and all instructions *must* be aligned on 3-word boundaries (i.e. the CPU will restrict the instruction pointer to multiples of 3). The first memory word of each instruction specifies the opcode and the addressing modes used for the operands. The second and third words specify the operands to the instruction. The designers of the NANORG hardware chose this encoding (as opposed to a variable-length instruction encoding) to reduce the impact of logic mutations.

In order to simplify the process of writing self-modifying NANORGs, *all jump and call instructions in NANORG programs that use immediate operands use* ***relative-addressing*** *for their target offset, enabling you to write relocatable code if desired (see Operand Details, bullet #5).*

The NANORG CPU has 14 registers, numbered R0-R13 that can be used in all instructions. All registers can be used interchangeably.

Each NANORG also has a flags register (which can be referred to as FLAGS) and the instruction pointer (i.e. program counter)). *Unlike typical processors, arithmetic instructions DO NOT set the CPU flags! See the instruction documentation for information on which instructions set flags.*

The CPU has a stack pointer called SP whose value starts at 3600 at the start of execution. When pushing values onto the stack, the stack pointer is pre-decremented and then the value is stored at the specified location in the NANORG memory. When popping values off the stack, the top value on the stack is fetched first and then the stack pointer is post-incremented. If the stack pointer is invalid (i.e. too large) during a PUSH, POP, CALL or RET instruction,

it will reset to 3600 before the instruction runs. You can directly refer to the stack pointer in all instructions, by using SP instead of normal registers like R0 or R09.

If at any point a NANORG CPU encounters an invalid instruction, it will attempt to interpret and run the instruction if at all possible. Here are the possible cases:

1. An instruction with an invalid opcode will be skipped.
2. An instruction with an invalid operand will have the value of that operand treated as if it were 0. E.g. “mov r5, [9999]” attempts to access slot # 9999 of memory, which does not exist, so r5 will be set to 0 instead. Alternatively, “mov r999, 52” will be treated as a NOP, since there is no such register number 999.
3. The CPU will only execute instructions on 3-word boundaries since all instructions are 3 words long. If the instruction pointer is set to a non-multiple of 3, it will be reset to the next multiple of 3 before executing the next instruction.

Organism Source File Layout

An organism source file has two *required* components:

1. The first line of your organism source file ***must*** have an information line that is used to identify your organism and yourself. It’s form is:

   info: robotname, your full name

   For example:

   info: SampleBot, John Doe

2. Your organism source file must have one or more assembly language instructions on the lines following your information line.

   You can have *standard instructions*, like these:

   ```
   mov r0, r5 // r0 = r5 // can be used for comments add r0, 17 ; r0 = r0 +
   17 ; semicolons comment too
   ```

   You can have *labels* which specify locations in your program, like these:

   ```
   startProgram: // a label call timeToEatFood // call to a label
           jmp startProgram // jump to a label

   timeToEatFood:                  // a label
           mov r0, 17
           eat
           ret
   ```

You can define data regions anywhere in your program, like this:

topOfProgram: mov r0, 0x01 mov r1, [myData+r0] // r1 = memory[myData+r0]
mov [myData], 5 // memory[myData] = 5 jmp overHere

myData:
data { 1 2 3 4 } // defines 4 words of data equal to {1,2,3,4} data { 0xdead }
// defines 1 word of data equal to {0xdead}

overHere:
sub r0, 1
jmp topOfProgram

otherData:
data { 99 127 127 55 myData 42 42 42 overHere }

Sample Organism Source

Here is an example organism file (its pretty poor, but it works):

info: SampleBot, John Doe

```
main:

// select a random direction and distance to move
  rand [dir], 4
  rand [count], 10
  add [count], 1


loop:

// check if I'm top of food and eat if so
  sense r2
  jns noFood
  eat


noFood:


// see if we're over a collection point and

// release some energy
  energy r0
  cmp r0, 2000
  jl notEnufEnergy
```

```
  sense r5
  cmp r5, 0xFFFF // are we on a collection point?
  jne notEnufEnergy
  release 100 // drain my energy by 100, but get 100 points,

// if we're releasing on a collection point


notEnufEnergy:

// move me
  cmp [count], 0 // moved enough in this dir; try new one
  je newDir
  travel [dir]
  jns newDir // bumped into another org or the wall
  sub [count], 1


jmp loop


newDir:
         rand   [dir], 4       // select a new direction
         rand   [count], 10    // select a new count between 0 and 9
         jmp    loop

dir:
        data { 0 } // our initial direction


count: // our initial count of how far to move in the cur dir
        data { 0 }
```

Using the Contest Program

The contest program (provided for Windows, Red Hat Linux, and Apple OS X) can be used to test and debug your NANORG program. Here's how to get started:

1. Unzip all of the files into a new directory (i.e., folder) on your hard drive.
2. Open a command shell (e.g. CMD.EXE under Windows) and make sure that the command shell window is at least 80 columns wide by 50 rows high $^{+++}$.
3. You can start by editing the sample NANORG that we provided: **samplebot.asm** using your favorite source editor, e.g. VI or NOTEPAD. You can modify the programming logic of our sample NANORG to improve it. Alternatively, you can create your own NANORG from scratch.
4. Once you have updated/created your NANORG source file, you can test it by following the instructions in the **"Assembling and Testing Your NANORGs"** section below. If your NANORG programming logic contains any syntax errors or other errors that prevent it from compiling, the contest program will inform you of these errors and you

can fix them.

5. You can iteratively update your NANORG's programming logic and re-test it until you are happy with it.

+++ You should run the contest program in a command shell (e.g. via CMD.EXE on Windows XP), since the contest program requires command-line arguments. Furthermore, make sure that your command shell window is sized to at least 80 characters wide by 50 characters high before running the contest06 executable. If you do not do this, the contest display will be difficult to understand.

Running the Contest Program on Apple or Linux Computers

The contest .ZIP file contains Windows, Linux and Apple versions of the contest program. The contest program file for Red Hat Linux is called **contest06-linux**. The contest program file for Apple OSX is called **contest06-apple**.

First, you should extract all of the files into a directory of your choice on your Apple or Linux computer. Next, open a command shell (e.g. csh) and switch to the directory that holds the contest files.

In order to run the contest executable on Linux and/or Apple OS X, you will need to change the permissions of the program file first. After extracting all of the files from the official contest .ZIP file into a directory on your hard drive, run the **chmod** command as shown below from within your command shell, in the same directory as the contest files:

For Apple, type the following on the command line:

chmod +rwx contest06-apple

and then hit enter.

For Linux, type the following on the command line:

chmod +rwx contest06-linux

and then hit enter.

You will now have a runnable executable file for either Apple OS X or Red Hat Linux.

To run the executable in the Apple OS X command shell, switch to the directory where you unzipped the contest files and type:

./contest06-apple

and then hit enter.

To run the executable in the Linux command shell, switch to the directory where you unzipped the contest files and type:

./contest06-linux

and then hit enter.

The sections below describe how to use the Windows version of the contest program: **contest06.exe**. If you are running the contest program on a Linux machine, then simply substitute "./**contest06-linux**" for the "contest06.exe" in the documentation below. If you are running the contest program on an Apple machine, then simply substitute "./**contest06-apple**" for the "contest06.exe" in the documentation below.

(Note the dot followed by a slash, preceding the filename)

Assembling and Testing Your NANORGs

To assemble and run your organism source file and see how it works, use this command line (or its equivalent for Linux or OSX as detailed in the section above):

C:\CONTEST06> CONTEST06.EXE –p:yourPlayerFile.asm

Where yourPlayerFile.asm is whatever you name your NANORG's assembly file.

This will assemble your assembly source and report any syntax errors it finds. If your source is error-free, then the program will bring up a simple ASCII-based user interface where you can watch your organism run in simulation. You can hit ctrl-c at any time to terminate the simulation. Note the line at the bottom of the screen:

```
Score: 24800, Ticks: 1100 of 1000000 (Seed=11275142)
```

This indicates that your current score is 24,800 (that's how much energy all of the NANORGs have generated so far), and that 1,100 of 1 million ticks have elapsed. The seed number specifies the random number seed used to run this simulation. If you run the contest program with the same seed value S and the same NANORG assembly file, it will have the same result every time (simplifying the debugging process).

There are two different types of organisms shown on the screen: your NANORGs and the malicious drone NANORGs. Your organisms are represented by upper and lower-case letters. The drones are represented using @ signs. Food is represented by an asterisk (*). Collection points are represented by a dollar sign ($). The entrant's hibernating NANORGs (those with 0 energy) are represented as a period (.), and hibernating drone NANORGs are represented by a comma (,).

When the simulation completes all 1 million time-ticks (or all organisms, including drones, perish), then a summary of the simulation is printed out:

```
Entrant: SAMPLEBOT, JOHN DOE, YOUREMAIL@DOMAIN.COM, 310 555-1212, UCLA
Your score: 1352226
Live organisms: 35, Live drones: 0, Final tick #: 1000000, Seed: 2
```

Debugging Your NANORGs

If you want to debug your NANORG logic, there are two mechanisms to do so. First, you can turn the systems' logging mode on. This will automatically log the logic of all of your NANORGs to a text file of your choosing:

C:\CONTEST06> CONTEST06.EXE –p:yourPlayerFile.asm –L:*file.txt*

Here's what the log entries look like in *file.txt*:

```
(SAMPL A) x=48, y=32, energy=10000, IP=0, SP=3600, flags=
R00= 0 R01= 0 R02= 0 R03= 0 R04= 0 R05= 0 R06= 0
R07= 0 R08= 0 R09= 0 R10= 0 R11= 0 R12= 0 R13= 0
0000 rand [57], 4 // ([57] = 0)


(SAMPL B) x=67, y=0, energy=10000, IP=0, SP=3600, flags=
R00= 0 R01= 0 R02= 0 R03= 0 R04= 0 R05= 0 R06= 0
R07= 0 R08= 0 R09= 0 R10= 0 R11= 0 R12= 0 R13= 0
0000 rand [57], 4 // ([57] = 0)


(SAMPL C) x=16, y=23, energy=10000, IP=0, SP=3600, flags=
R00= 0 R01= 0 R02= 0 R03= 0 R04= 0 R05= 0 R06= 0
R07= 0 R08= 0 R09= 0 R10= 0 R11= 0 R12= 0 R13= 0
0000 rand [57], 4 // ([57] = 0)
```

In front of each line is the shortened name of your NANORG followed by the identifying letter of the NANORG (as displayed on the screen during the simulation). You can see each NANORG's x,y location, its energy, its instruction pointer, stack pointer, and flags. You can also see all of its registers and the current instruction it's about to execute. Be careful, this file grows *very* quickly, and logging slows down the simulation, so use this sparingly.

The second method of debugging is an interactive, single-step debugger. Before you debug, you must decide which of your NANORGs you want to debug. Select one of the alphabetical letters on the screen (a-z or A-Z) and then run the contest program like this:

C:\CONTEST06> CONTEST06.EXE –p:yourPlayerFile.asm –g:C

For example, the above line will allow you to debug and trace through the logic of clone C as it runs through the simulation. Here is what you'll see at the bottom of the screen:

```
(SAMPL C) x=67, y=17, energy=10000, IP=0, SP=3600, flags=
R00= 0 R01= 0 R02= 0 R03= 0 R04= 0 R05= 0 R06= 0
R07= 0 R08= 0 R09= 0 R10= 0 R11= 0 R12= 0 R13= 0
0000 rand [57], 4 // ([57] = 0)
(u)nasm,(g)o,(s)ilentGo,(d)mp,(e)dt,(r)eg,(i)p,(q)uit,##, or [Enter]:
```

Now you can single-step debug organism C. You will be given single-step control over your specified organism as long as the organism has at least 1 unit of energy. If the organism you are single step debugging goes into hibernation, then you will no longer be able to debug the organism unless it is CHARGEd by another organism.

Here are the commands you can use in the single-step debugger:

(U)nasm

If you type “u” and hit enter, the debugger will show a disassembly of the next set of instructions starting with the current instruction pointer.

If you type “u ####” and hit enter, the debugger will show a disassembly of the set of instructions starting at offset ####. *#### must be a decimal number.*

(G)o

If you type “g” and hit enter, the debugger will be turned off and the simulation will run to completion.

If you type “g ####” and hit enter, the debugger will run until the instruction pointer reaches offset #### and then allow you to single step again. This is useful if you want your NANORG to run until it reaches an error condition or subroutine you’d like to test. *#### must be a decimal number.*

(S)ilentGo

The SilentGo command *is the same as the (G)o command*, except the simulation will not display any output to the screen until the target instruction pointer is reached (if the user uses the “s ####” syntax), or until the simulation comes to an end (if the user uses the “s” syntax). You can use this command to speed up the debugging process by eliminating screen output.

(D)ump

If you type “d” and hit enter, the debugger will show a memory dump of the memory in both decimal and hexadecimal starting at the instruction pointer.

If you type “d ####” and hit enter, the debugger will show a dump of the memory starting at offset ####. *#### must be a decimal number.*

(E)dit

If you type “e offset value” and hit enter, the debugger will store the specified *value* in the NANORG’s memory at the specified *offset*. ***You must specify decimal values for both the offset and the value.***

(R)eg

If you type “r reg# value” and hit enter, the debugger will store the specified value into the specified register. For example, “r 3 27” will set r3=27. ***You must specify decimal values for both the register number and the value.***

(I)p

If you type “i value” and hit enter, the debugger will change the program’s instruction pointer to the specified value. ***You must specify decimal values for the IP address***.

(Q)uit

If you type “q” and hit enter, the contest program will exit.

##

If you type an arbitrary number and then hit enter, the contest program will run the specified number of instructions (without displaying single-step information) and then continue single-stepping once this has been completed.

[Enter]

If you hit enter without typing anything else, then the current instruction will be executed and the single-step debugger will stop and prompt you for the next instruction.

Using the Seed Command

Occasionally when debugging, you will notice one of your NANORGs behaving erratically. You may wish to run the simulation again in single-step debugging mode and determine what went wrong. To do so, you’ll want to run the simulation again using the same seed as was used during the last run; this ensures that the exact same set of circumstances will be reproduced. To do so, write down the seed value during your first run. During the second run, you can then invoke the contest program as follows:

C:\CONTEST06> CONTEST06.EXE –p:myorg.asm –g:C –*s:####*

Where you specify the letter of the misbehaving clone that you wish to debug after “–g:” and you specify the proper seed value after “–s”, for example: “*-s:112314895*”. This will ensure that (assuming you make no changes to your organism’s source code in between debugging sessions) that you will exactly recreate the previous simulation, only in debug mode.

Producing a Disassembly of Your Organism

You may wish to produce a disassembly of your organism for debugging purposes (e.g., to look at the *machine code* for your organism). To do so, use the following command:

C:\CONTEST06> CONTEST06.EXE –z:myorg.asm

Running the Contest Program in Quiet Mode

To speed things up drastically (after all, our UI is quite slow), you may wish to run simulation in quiet mode. This will ensure that the simulation runs without any output; the result of the simulation will simply be displayed on-screen upon completion. Note: you cannot use the single-step debugger (option –g) while the game is in quiet mode:

C:\CONTEST06> CONTEST06.EXE –p:myorg.asm –q

Instruction Details

This section describes each of the instructions in the NANORG assembly language. Unless otherwise stated, each instruction uses up 1 unit of energy from your organism:

Instruction	Opcode Value (decimal)	Operands	Examples	Behavior
NOP	0	None	NOP	Does nothing

MOV	1	dest, src	mov r0, 5
			mov [r2], r1
			mov SP, 3500
				Moves the value of the source operand into the destination operand. Flags: No effect on flags.

PUSH	2	src	push r12 push [r3+3]	Pushes the value in the src operand onto the stack (pre-decrementing the stack pointer, SP, first). Flags: No effect on flags.
POP	3	dest	pop r1 pop [100]	Pops the top value from the stack into the destination operand (postincrementing the stack pointer, SP). Flags: No effect on flags.
CALL	4	label	call foo call [10] foo: …	Pushes the address of the next instruction on the stack and transfers control to label. Flags: No effect on flags.
RET	5	None	ret	Sets the instruction pointer to the top address on the stack and pops this value off the stack. Flags: No effect on flags.
JMP	6	label	jmp bar jmp 1005 bar: jmp [data+r2]	Unconditionally jumps to the specified address in the program. Flags: No effect on flags.
JL	7	label	jl foo	Jumps to the specified address in the program if the LESS flag is set. Otherwise continue execution at the instruction following the JL. Flags: No effect on flags.
JLE	8	label	jle bletch … bletch:	Jumps to the specified address in the program if the LESS or EQUAL flags are set. Otherwise continue execution at the instruction following the JLE. Flags: No effect on flags.

Instruction	Opcode Value (decimal)	Operands	Examples	Behavior
JG	9	label	foo: … jg foo	Jumps to the specified address in the program if the GREATER flag is set. Otherwise continue execution at the instruction following the JG. Flags: No effect on flags.
JGE	10	label	jge blah … blah:	Jumps to the specified address in the program if the GREATER or EQUAL flags are set. Otherwise continue execution at the instruction following the JGE. Flags: No effect on flags.
JE	11	label	je oops … oops:	Jumps to the specified address in the program if the EQUAL flag is set. Otherwise continue execution at the instruction following the JE. Flags: No effect on flags.
JNE	12	label	jne oops … oops:	Jumps to the specified address in the program if the EQUAL flag is NOT set. Otherwise continue execution at the instruction following the JNE. Flags: No effect on flags.
JS	13	label	js foundFood … foundFood: …	Jumps to the specified address in the program if the SUCCESS flag is set. Otherwise continue execution at the instruction following the JS. Flags: No effect on flags.
JNS	14	label	jns foundFood	Jumps to the specified address in the program if the SUCCESS flag is NOT set. Otherwise continue execution at the instruction following the JNS. Flags: No effect on flags.
ADD	15	dest, src	add r0, 5 add SP, 17	dest = dest + src Flags: No effect on flags.
SUB	16	dest, src	sub [r0], r1 sub [r4], label5	dest = dest – src Flags: No effect on flags.
MULT	17	dest, src	mult r7, r9 mult [r0], [label]	dest = dest * src Flags: No effect on flags.
DIV	18	dest, src	div r1, r2 div r5, 4	dest = dest / src Dividing by zero will cause this instruction to operate like a NOP Flags: No effect on flags.
MOD	19	dest, src	mod r9, 10	dest = dest modulo src Modding by zero will cause this instruction to operate like a NOP Flags: No effect on flags.

Instruction	Opcode Value (decimal)	Operands	Examples	Behavior
AND	20	dest, src	and r0, 0xFF05	dest = dest AND src Performs a bitwise AND. Flags: No effect on flags.
OR	21	dest, src	or r9, r8 or r8, 1234 or [r1], [r2]	dest = dest OR src Performs a bitwise OR operation. Flags: No effect on flags.
XOR	22	dest, src	xor [r1], [r2]	dest = dest XOR src Performs a bitwise exclusive OR operation. Flags: No effect on flags.

CMP	23	op1, op2	cmp r10, 20
			cmp SP, 3600
			cmp 55, [r3]
			cmp label1, r0
				1. Clear *all* flags 2. Compares op1 with op2 and set: LESS flag if op1<op2 GREATER flag if op1>op2 EQUAL flag if op1 == op2

TEST	24	op1, op2	test r0, 0xff00 test r0, r2 je someLabel	**1. Clear *all* flags 2. temp = op1 AND op2 3. If temp is equal to zero then set EQUAL flag. 4. Discard the value of temp**
GETXY	25	destx,desty	getxy r0, r1 getxy [99], [100] getxy [r1], [r3]	Places the X and Y coordinates of the organism in the specified destination registers or memory locations. Flags: No effect on flags.
ENERGY	26	dest	energy r0 energy [label5] … label5: data { 0 }	Places the organism's current energy value into the specified dest register or memory location Flags: No effect on flags.
TRAVEL	27	direction	travel 0 // north travel 1 // south travel 2 // east travel 3 // west travel r0	Moves the organism one slot in the specified direction assuming the space is no occupied by another organism or outside the sludge tank. This instruction costs 10 energy points if successful; otherwise it costs 1 energy point. When an organism moves: North: their y coord lessens by 1 South: their y coord increases by 1 West: their x coord lessens by 1 East: their x coord increases by 1 **Flags: If the organism successfully moves, the SUCCESS flag is set. Otherwise the SUCCESS flag is cleared.**

Instruction	Opcode Value (decimal)	Operands	Examples	Behavior
SHL	28	dest, count	shl r0, 3	Does a bitwise shift on dest, shifting it by count bits to the left. If count > 16, then the result is undefined. Flags: No effect on flags.
SHR	29	dest, count	shr r0, 3	Does a bitwise shift on dest, shifting it by count bits to the right. If count > 16, then the result is undefined. Flags: No effect on flags.
SENSE	30	dest	sense r0	If the organism is on a square that contains sludge or a collection point, then dest is set to the ID type number of the sludge, or a value of 65535 if the organism is on a collection point. Dest will be set to 0 otherwise. **Flags: If the organism is on a square that contains food or a collection point, the SUCCESS flag is set, otherwise the SUCCESS flag is cleared.**
EAT	31	None	eat	If the organism is on a square that contains sludge/food then the organism eats the food and the food disappears from the current square. The organism then receives 2000 energy units. If eating the sludge will push the organism over 65535 energy units, then the organism will fail to eat it. **Flags: If the organism successfully eats food, the SUCCESS flag is set, otherwise the SUCCESS flag is cleared.**
RAND	32	dest, max	rand r5, 10 rand r6, 22 rand r7, label	Computes a random number between *0* and *max-1* inclusive and stores this random number into dest. Flags: No effect on flags.
RELEASE	33	energyAmt	release 1000 release r0	Attempts to release energy from the organism. This instruction fails if the organism attempts to release more energy than it has. If the organism is on a collection point, then its energy is reduced by energyAmt and its score increases by the same amount. If the organism isn't on a collection point, the organism simply loses the specified amount of energy with no change to its score. **Flags: If the release is successful, the SUCCESS flag is set, otherwise it is cleared.**

Instruction	Opcode Value (decimal)	Operands	Examples	Behavior

CHARGE	34	dir, energy	charge 0, 1000
			// charge north
			charge 3, 50
			// charge west
				Attempts to share energy with an organism on an adjacent square by deducting the specified energy from the current organism and crediting it to the adjacent organism. Fails if there is no adjacent organism in the specified direction, if the initiating organism doesn't have enough energy, of if the target organism will exceed 65535 units of energy after the transfer. **Flags: If the charging is successful, the SUCCESS flag is set, otherwise it is cleared.**

POKE	35	dir, offset (R0 = value)	mov r0, 15 poke 0, 1000	If there is an organism in the specified direction, then this instruction sets that organism's memory[offset] = R0. In the example to the left, this will stick a value of 15 into slot 1000 of an organism to the NORTH (dir=0). **Flags: If the poking is successful, the SUCCESS flag is set, otherwise it is cleared.**

PEEK	36	dest, offset	mov r1, 0
			peek r1, 999
			// after completion, r1 will be set to the northern organism's memory[999]
				If there is an organism in the specified direction (specified by the *dest* operand) then this instruction will retrieve that organism's memory[slot#] and store it back into *dest*. **Flags: If the peeking is successful, the SUCCESS flag is set, otherwise it is cleared.**

CKSUM	37	start, end	mov r1, 100
			cksum r1, 200
			// after completion, r1 will be set to the checksum of memory slots 100 to 199
				This instruction computes a checksum of the specified range of words start,end-1 inclusive from the current organism's memory by adding their values together. Upon completion, *start* is set to the checksum value. **Flags: No effect on flags.**

Operand Details

Each operand can have several different forms (see appendix I for machine language encoding details):

1. You can specify a register as an operand:

```
MOV r0, r1          ; copy the value in register r1 into register r0
MOV SP, r3          ; copy the value in register r3 into the stack pointer
```

2. You can specify an immediate value as an operand:

```
ADD r0, 1234        ;r0 =r0+1234
ADD r5, 0x5678      ; r5 = r5 + 0x5678
Label4:
MOV  r3, Label4     ; r3 gets the offset of Label4. if our program were
                    ; built from just these 3 instructions, then r3 would
                    ; get a value of 6 (the offset of the 3rd instruction)
```

3. You can directly index memory using an optional register and/or optional offset:

```
MOV r0, [r1]        ; fetch the r1'th word of memory and store into r0
MOV [r2+5], 123     ; store the value of 123 into memory slot [r2+5]
                    ; e.g. if r2 is equal to 7, then mem[12] = 123
MOV [r2-3], [r3]    ; mem[r2-3] = mem[r3]
MOV [5], r3         ; mem[5] = r3
MOV [SP], r4        ; store the value of r4 on the top of the stack
```

4. You can index memory relative to a user-defined label:

```
lb1:
        mov   r0, [lb2]         ; get the value in memory at location lb2 and
                                ; store this value into r0. I.e. r0 = 0xdead
        mov   [lb1+ r2], r3     ; store the value of r3 into the memory word that
                                is
                                ; r2 slots down from lb1
lb2:
         data { 0xdead 0xbeee 1 2 3 }      ; defines 5 words of memory
```

5. All jump and call instructions use relative addressing for immediate operands and fixed addressing for other types of operands. For example (see the machine language encoding of the JMP instruction below):

```
start:
    mov r0, r1 ; 3 words long
    xor r0, r2 ; 3 words long
    jmp label1 ; 8006 0009 0000 --label1 is 9 words from start of this jmp instr
    add r3, [r4+1] ; 3 words long
    call label7 ; 3 words long

label1:
     jmp start ; 8006 FFF1 0000 --start: is 15 words up (FFF1 is -15)
```

However, when using other types of operands, fixed addressing is used:

```
mov r0, 999
jmp r0          ; will set instruction pointer to 999, not to curIP+999

jmp [r0+5]      ; fetches mem[r0+5] and sets the instruction pointer to that value
                ; adjusting the instruction pointer to be a multiple of 3, as required.
```

The following addressing modes are NOT SUPPORTED:

```
mov r0, [label+5]      ; illegal to specify a label and constant modifier
mov r0, label+5        ; illegal to specify a label and constant modifier
mov r0, [label-r1]     ; can't have a negative register offset from label
mov r3, r0+5           ; can't add registers and constants in this way
```

Details of Defining Data

You may define data in your program using the data command. For instance, the data statement below reserves 10, 16-bit words of memory and ensures that these memory locations are set to the specified values.

```
data { 1 2 3 4 10 0x55 0xFF 99 32 aLabelName }
```

You may specify any reasonable number of data values within the bracers, including decimal numbers, hexadecimal numbers, and label names defined elsewhere in your program. If you specify a label name in a data statement, then the offset of that label in the program will be the value stored in your data area.

Typically, when you define data you specify a label before the data, so you can then easily refer to this data from your program:

```
        getxy   [myXCoord], [myYCoord]
        cmp     [myXCoord], 10
        je      doSomething
        add     [myXCoord], 5
        …


myXCoord:
        data { 0 }


myYCoord:
        data { 0 }
```

You may intersperse data statements in between your instructions. If you do this, the assembler guarantees that all instructions are aligned on 3 word boundaries:

```
topOfProgram:
        mov   r0, 1          // instruction is at offset 0 of memory
        add   [r0], 5        // instruction is at offset 3 of memory
        jmp   overHere       // instruction is at offset 6 of memory
someData:
        data  { 0xdead }     // a word of data is at offset 9 in memory
        data  { 0xbeee }     // a word of data is at offset 10 in memory
                             // a value of zero will be placed here automatically
overHere:
        cmp   [r5], r7       // instruction is at offset 12 in memory
                             // even though two only data words were defined above
```

Appendix I: Instruction Encoding

Each instruction is comprised of three 16-bit words, with the following format:

[opcode and operand addressing modes] [operand 1][operand2] ; instr #1
[opcode and operand addressing modes] [operand 1][operand2] ; instr #2
[opcode and operand addressing modes] [operand 1][operand2] ; instr #3
…

The least significant byte of the opcode word specifies the opcode number of the instruction. The most significant byte of the opcode word specifies the addressing modes for both operands.

The most significant two bits of the opcode word (bits 15 and 14) represent the addressing mode for the first operand. The next two most significant bits of the opcode word (bits 13 and 12) represent the addressing mode for the second operand. The next two bits (bits 11 and 10) are only used in the register-indexed addressing mode (see below). Here are the details:

Here are the encodings for various addressing modes (for bits 15+14 or bits 13+12):

msb	lsb	meaning
0	0	direct addressing mode (i.e. [1234])
0	1	register mode (i.e. r0)
1	0	immediate value mode (i.e. 1234)
1	1	register-indexed direct addressing mode (i.e. [r0+100] or [label1+r0]) where the constant value (e.g. 100) is encoded as a 13-bit, signed number.

If the register-indexed direct addressing mode is used for an operand, the high-order nibble of such an *operand* specifies the register number (0-15) and the low-order 12 bits of the operand specify the lower 12-bits of the 13-bit index/offset. If the first operand of the instruction is a register-indexed operand, then the 13th and most significant bit of the index/offset is stored in bit 11 of the opcode word. If the second operand of the instruction is a register-indexed operand, then the 13th and most significant bit of the index/offset is stored in 10 of the opcode word (where the least significant bit is bit 0). In this mode, the 13-bit offset is a signed number which is sign-extended during execution to compute the final address (see the *ed examples below):

Here are some examples:

Instruction	Machine language
	Opcd Op1 Op2
mov r5, 0xDEAD	6001 0005 DEAD
mov r5, 0xDEAD	6001 0005 DEAD
mov [5], 0xDEAD	2001 0005 DEAD
mov [r5+255], 0xDEAD	E001 50FF DEAD*
mov [r5-9], 0xDEAD	E801 5FF7 DEAD*
mov [r9+12], 0xDEAD	E001 900C DEAD*
mov [12], 0xDEAD	2001 000C DEAD
mov r3, r4	5001 0003 0004
mov [r4+7], [r4+7]	F001 4007 4007*
mov [r4-7], [r4-7]	FC01 4FF9 4FF9*
sub r6, [r4-7]	7410 0006 4FF9*
add r11, [5]	400F 000B 0005

Where the MOV instruction has an opcode encoding of 0x01, the SUB instruction has an opcode encoding of 0x10, and the ADD instruction has an opcode encoding of 0x0F.

Appendix II: What is a Checksum?

A checksum is a numeric value that characterizes a set of data. It is computed by summing up the elements of that data. For instance, one can compute the checksum of an array of integers by summing up all of the elements of the array.

Checksums are typically used to detect corruptions/errors in data transmission. For example, before sending a packet of data over the network, we can compute its checksum by adding all of its bytes together. This checksum value is then sent along with the packet to its destination. The destination computer can then recompute the checksum for the packet's data and compare it to the original checksum. If the two checksums don't match, then the packet was likely corrupted in-transit, and the receiving computer and request that the data be resent.

In this project, checksums can be useful to detect mutations of your NANORG's logic.