Computer Science 52

The CS52 Machine

Spring Semester, 2017

Contents

1 The CS52 Machine Architecture 2
2 CS52 Machine Programming 6
3 Assembly Language Reference 21
4 Machine Language Reference 29
5 Real Computers 35
A The CS52 Machine Simulator 42
B Binary Representation of Integers 48
C Sample Programs 51
D Historical and Contextual Notes 61
E Practice Exercises 65
F Assembly Language Quick Reference 68

The CS52 Machine is a rudimentary model of a computer, used to understand how real computers operate at a low-level. We will use it to study how data are stored, how programs are executed, and—most importantly—how recursion is implemented at the machine level.

We begin in Section Section 1 with a description of the machine and its operation. Then in Section Section 2, we describe the assembly language and see how programs are run. After reading this material, you will be able to write programs in the CS52 Machine assembly language and run them on the simulator. See Appendix A if you need help using the simulator.

Sections Section 3 and Section 4 are reference manuals for the assembly language and the underlying machine language, respectively. They may be skipped on first reading but are likely to be valuable as you gain more experience writing programs.

All of the ideas in the CS52 Machine are present in modern computers. In Section Section 5 we compare the CS52 Machine with real computers to see what is missing from our model.

The appendices contain documentation for the simulator for our model, some sample programs, historical remarks, practice exercises, and a quick reference sheet for the assembly language.

1 The CS52 Machine Architecture

The fundamental data type of the CS52 Machine is the 16-bit word, two bytes. As unsigned integers, these words take on values between 0 and $2^{16}-1$. See Appendix B for a refresher on binary representation of integers.

The CS52 Machine consists of a memory and five registers. There is a set of instructions that modify the values in memory and registers. A program is simply a sequence of instructions. Figure 1 shows a block diagram of the components of the CS52 Machine.

1.1 Memory

The *memory* is an array of bytes. Each index into the array is itself a word, interpreted as an unsigned value. An index into the memory array is called an *address* or a *location*. The number of bytes in a memory can vary, but it can never be larger than 2^{16} bytes. In our work, the memory will be much smaller.

Even though memory is indexed by bytes, the size of a transaction with memory is the word. The word at address `addr` is formed by taking the low order byte from address `addr` and the high order byte from address `addr`+1. Normally our addresses are even numbers, so we can think of memory as an array of *words* whose addresses are even. (However, we also permit odd addresses and speak of the two-byte word beginning at that address.) We use the notation `mem[addr]` to refer to the *word* at address `addr`. Consecutive words in memory have addresses that differ by two.

The word values in memory can be used and changed. We *load* a value from a memory address in order to use it, and we *store* a new value at a memory address. The word values stored in memory can be interpreted as signed integers, as unsigned addresses into memory, or as instructions to the CS52 Machine.

Memory location 0 has a special use in the CS52 Machine. Rather than storing a value, it is the conduit for input and output. By obtaining a value from `mem[0]` a program can acquire data from the user. Storing a value at address 0 is the way to create output. Values for input and output are interpreted and displayed as signed integers.

1.2 Registers

All of the actual computation takes place in the registers. A *register* is a component that holds the value of one word. There are five registers in the CS52 Machine.

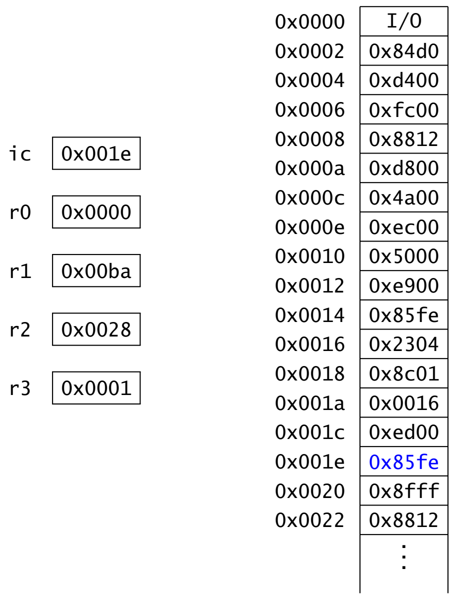


Figure 1: A block diagram of the CS52 Machine. The registers are on the left, and a portion of memory is on the right. Register `ic` has the value `0x001e`, and the word at that address—shown in blue—is the instruction that is about to be executed. All values are given in hexadecimal.

Register `ic` is the *instruction counter;* it holds the address in memory of the *next* instruction to be executed.

The other four registers hold data values on which the machine operates. Register `r0` is special in that it always contains the value zero; any attempt to change it is ignored. The other registers, `r1`, `r2`, and `r3`, are general-purpose registers whose values can be read and modified by program instructions.

The CS52 Machine itself does not reserve the data registers for any particular purpose. Later, when we encounter subprograms, we will adopt a special convention for the use of the register `r1`.

1.3 Instructions

An *instruction* to the CS52 Machine is a tiny step of a computation; it may change a value in a register or at a memory location.

From the CS52 Machine's point of view, instructions are simply sixteen-bit quantities. Later, we shall see exactly how these instructions are encoded as words in memory. As human beings, we usually write instructions in a more understandable way, using the CS52 Machine assembly language. Each instruction consists of a three-letter abbreviation followed by up to four arguments, which may be registers or numerical values. For example, the instruction

```
add r3 r2 r2
```

causes the machine to add the value in `r2` to itself and place the result in `r3`. The convention, consistent with assignment statements in other programming languages, is that the leftmost register is always the one that *receives* the new value. Similarly,

```
loa r3 r1
```

causes the machine to use the value in `r1` as an address and to place the value at that address in memory into `r3`.

The arguments in an instruction can be registers, numerical values, or labels referring to locations in the program. In specifying arguments, we use `rX` to refer to the (leftmost) destination register, and we use `rY` and `rZ` to refer to the other two register arguments. Only one of the four possible arguments can be a non-register; if it is present, it must be the last argument. We use `arg` to refer to a non-register argument.

An instruction may change the value in a register or in a memory location, but not both. Each instruction also changes the value in `ic`, usually by incrementing it by 2 to move on to the next instruction in sequence.

The full assembly language instruction set is described in detail in Section 3 and is summarized in a single page in Appendix F.

1.4 Programs

A *program* is simply a sequence of instructions stored in a block of contiguous words in the machine's memory. In executing a program, the CS52 Machine follows a simple loop:

- The machine fetches the value at `mem[ic]` for use as an instruction.
- The machine increments the value in `ic` by 2.
- The machine decodes and carries out the instruction.

Figure 2 shows one iteration of the loop as the CS52 Machine executes a program. The loop continues until a `hlt` instruction is executed or some error occurs.

Notice that, while an instruction is being executed, the value in `ic` is the address of the *next* instruction. We will see that this detail is important when we look at the branch instructions.

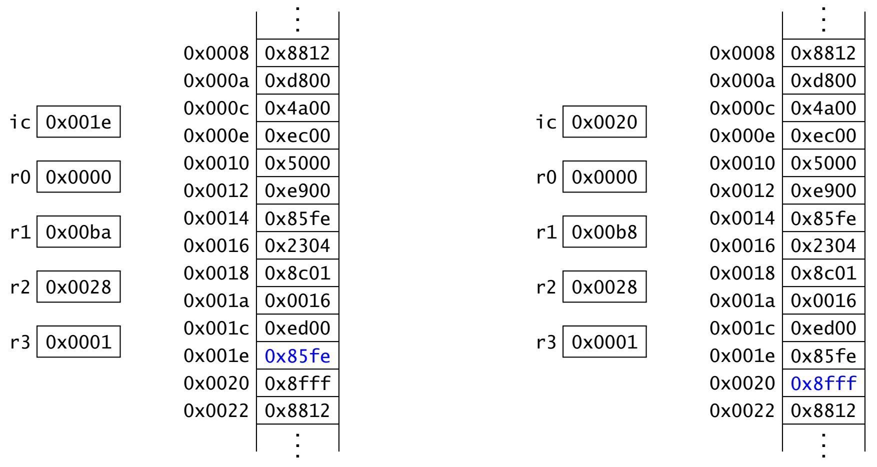


Figure 2: On the left is the configuration of the CS52 Machine shown in Figure 1. The instruction in blue is about to be executed. As an instruction, the value `0x85f3` means "change the value in `r1` by subtracting 2." On the right is the configuration after that single instruction is executed. Notice that the only changes are to the values in registers `ic` and `r1`. The next instruction will come from memory address `0x0020`.

At this low "machine" level, there is no type checking. A word is simply a 16-bit quantity that may be interpreted as a data value that is a signed integer, as an unsigned address into memory, or as an instruction to be executed.

The CS52 Machine will halt and report an error if it tries to use an address that is is too large for the actual memory. Similarly, it will halt and report an error if a word that is to be interpreted as an instruction is not a legitimate instruction. There are other errors from which the CS52 Machine offers no hardware protection. For example, there is no simple way to test for arithmetic overflow, and there is no way to prevent data from being interpreted as instructions, or *vice versa*. The programmer must therefore be very careful to insure that values are used properly.

2 CS52 Machine Programming

In the olden days, programmers computed each bit of each instruction. They entered the instructions by hand, using plug boards or toggle switches. As you can imagine, the process was tedious and error-prone. They quickly realized that computers were more reliable for that kind of work and developed programs to do the intricate work with bits.

You will be writing programs in the CS52 Machine's assembly language, using abbreviations like `add` and `loa` which we have just seen. There is a program called an *assembler* that translates the assembly language program into a sequence of 16-bit instructions that can be executed on the CS52 Machine itself. For example, the assembler will translate the assembly language instruction

```
add r1 r1 -2
```

into the machine instruction `0x85fe` that we saw in Figure 2.

In this section we give some informal descriptions of the assembly language instructions and show how they can be combined into programs. A more formal description of the assembly language can be found in the next section.

Our approach is bottom-up. We begin with basic programming constructs like assignment statements, loops, and if-then-else statements. In the first few examples we will use all three non-zero registers. Later, when we move to subprograms, we will have to adopt a convention that reserves `r1` for a specific use. Life will get a little more complicated with only two registers available for data values.

2.1 Assignment Statements

As you well know, the assignment statement

```
k = k + 1;
```

fetches the value of the variable `k`, adds 1 to it, and stores the result back in the variable `k`. The value of the variable `k` is stored in memory at a known address. Assuming that the address for `k` is in register `r2`, we may execute the following block of code to increment its value.

```
loa r3 r2       ; put the value of k in r3
add r3 r3 1     ; add 1 to it
sto r3 r2       ; store the new value back into k
```

These three instructions exactly reflect our understanding of the assignment statement. It is not hard to imagine more complicated assignments in which the values of several variables are used to update another variable.

There are a few details about the assembly language syntax that should be mentioned here. There is one instruction to a line. The semicolon is the comment character—it and everything following it is ignored by the assembler. Because assembly instructions are so cryptic, programmers comment nearly every line of code, as we have done above.

It is not obvious from the example above, but every instruction must be indented away from the left margin. There must be at least one blank space (we usually use several) between the left margin and the beginning of the actual instruction. A word that begins at the left margin is interpreted by the assembler as a *label,* not an instruction. We shall see examples of labels shortly.

2.2 Straight-line Code

A very simple example of a whole program is one that accepts two integers and prints their difference. It has only five executable instructions: read, read, subtract, write, and halt.

```
     loa r2 r0        ; get the first value
     loa r3 r0        ; get the second value
     sub r2 r2 r3     ; compute the difference
     sto r2 r0        ; write the result
     hlt              ; quit
```

Notice that we have used register `r0` as the second register for `loa` and `sto`. Remember that the value of `r0` is always zero, and that the memory at address zero is used for input and output. Therefore, loading from address zero gets a value from the user, and storing a value at address zero writes the result for the user to see.

The five lines above form a complete program. Normally, a program has a (sometimes lengthy) preamble containing comments indicating who wrote the program, when, why, and how. The complete program appears among the example programs as `subtract.a52`.

2.3 While-loops

Suppose that we want to multiply two numbers instead of subtracting them. The CS52 Machine has no multiply instruction, so we resort to repeated addition. (Warning: This is *not* how real computers multiply! The example gives us a clean and simple illustration of a loop, but no real computer—or real programmer—would ever do anything so inefficient. Later, we will see a much faster multiplication method.) Starting with non-negative numbers `a` and `b`, we want to initialize a value `result` to zero and then add `b` to it `a` times.

```
result = 0;
while (0 < a) {
   result += b;
   a--;
}
```

As you know, the while-loop starts by making its test (`0 < a`, in this case). If the test result is true, the body of the loop is executed and the program returns to the top of the loop for another test. If the test result is false, the body of the loop is skipped and the computation proceeds with the code after the loop.

We use labels to indicate positions in the code, and branch instructions to control the location of the next instruction. The instruction `brs` is the *unconditional branch;* it causes the next instruction to come from a location specified by the label. The instruction `ble` is a *conditional branch;* it makes a comparison to determine the location of the next instruction. If the comparison is true (the value in the first register being less-than-or-equal-to the value in the second, in the case of `ble`), the next instruction comes from the location specified by the label. If the comparison is false, computation simply proceeds to the instruction following the branch.

Let us use `r1` for `a`, `r2` for `b`, and `r3` for `result`. Notice how faithfully the assembly language code below follows the while-loop.

```
        mov r3 r0           ; result = 0;
loop
        ble r1 r0 endloop   ; stop if a <= 0
        add r3 r3 r2        ; result += b;
        sub r1 r1 1         ; a--;
        brs loop            ; return for another iteration
endloop
```

We could package this code with two input commands and an output command and have a multiply program just like our subtraction program, but there is a further refinement: Notice that the target of the `brs` instruction is another (conditional) branch instruction. Why branch to a conditional branch? Why not make the test in the first branch? We can change the logic slightly and eliminate one instruction from the body of the loop.

```
        mov r3 r0           ; result = 0;
        ble r1 r0 endloop   ; skip the whole loop if a <= 0
loop
        add r3 r3 r2        ; result += b;
        sub r1 r1 1         ; a--;
        blt r0 r1 loop      ; return for another iteration,
                            ;    if necessary
```

```
endloop
```

There are still five instructions, but now only three of them are in the body of the loop. As you have seen from your work with other programming languages, many programs spend most of their time in loops. If we can cut down the number of instructions in the body of a loop, we can significantly reduce the running time of the program. Here, we have reduced the time for one iteration of the loop body by one-fourth.

The loop above forms the core of the example program `multiply.a52` and provides a template to use for all while-loops.

```
        b?? r? r? endloop   ; do we enter the loop?
loop
        ...                 ; loop body
        b?? r? r? loop      ; another iteration
endloop
```

Obviously, if we have more than one loop in a program, we have to use more imaginative names than `loop` and `endloop`.

Once we have understood one kind of loop, it is easy to code other sorts of loops. For example, the for-loop `for (a=0; a<b; a++) {...}` can be translated into the while-loop below, and so our technique can accommodate for-loops as well.

```
a = 0;
while (a < b) {
   ...
   a++;
}
```

2.4 Conditional Execution

By now, we have seen many of the basic assembly language instructions: the register instructions `add`, `sub`, and `mov`; the memory instructions `sto` and `loa`; and the branch instructions like `brs` and `blt`. Those instructions, and minor variations of them, are sufficient to translate all the fundamental steps of a computer program.

The only construction left to consider is the conditional, if-then-else. The basic pattern consists of two blocks of code, only one of which is executed. If the condition is true, the computer executes the then-part and skips over the else-part. If the condition is false, the computer skips the then-part and executes the else-part. It all can be carried out with two branches and two labels.

```
        b?? r? r? else      ; branch if condition is false

        ...                 ; then-part
        brs endif           ; skip else-part
else
        ...                 ; else-part
endif
```

For a concrete example, we compute the “sign” of a number.

```
if (a < 0)
   write(-1);
else if (0 < a)
   write(+1);
else
   write(0);
```

Suppose that the value `a` is in `r2`. We have the following code. Notice that we branch to the else-part, so that the branch condition is the negation of the one in the if-statement.

```
        ble r0 r2 elseif    ; first comparison
        add r3 r0 -1        ; a < 0, place result -1 in r3
        sto r3 r0           ; write the value in r3
        brs endif           ; skip other clauses

elseif
        ble r2 r0 else      ; second comparison
        add r3 r0 1         ; 0 < a, place result +1 in r3
        sto r3 r0           ; write the value in r3
        brs endif           ; skip other clauses

else
        sto r0 r0           ; a == 0, write the value zero

endif
```

The full program appears as `sign.a52` in Section C.2.

2.5 Arrays and Nested Loops

The loop template from Section 2.3 can be applied inside itself to obtain nested loops. The Sieve of Eratosthenes, a classical method for listing prime numbers, provides an example of nested loops. The idea is to make a list of numbers from 2 up to some limit. We make several passes across the list. On each pass, the first number encountered is a prime. We record that number and then remove it and all its multiples from the list.

We can implement the sieve in Java or C++ with an array called `num`. The value `num[i]` is non-zero or zero, according to whether or not `i` is still on the list.

```
for (i = 2; i < limit; i++)
    num[i] = i;    // initialize to a non-zero value
for (i = 2; i < limit; i++)
    if (num[i] != 0) {
        write(i);
        for (j = i; j <= limit; j += i)
            num[j] = 0;
    }
```

In the CS52 Machine, we can represent the array as a contiguous sequence of words in memory. If `num` is the address of the base of the sequence, then the value `num[i]` is at address `num` + 2`i`. Suppose that `i` is in `r3`. If the base address `num` is a small value, one that can fit in a single signed byte, then we can store a value into `num[i]` with just two instructions.

```
        add r2 r3 r3
        sto r3 r2 num
```

Notice how `r2` is used as a temporary register to hold 2`i`. (How would you get the value of `num[i]` if the array `num` were *not* stored at a small address?)

The first part of our program reserves memory for `i` and `num`. We allocate two bytes for `i` and 202 bytes for `num`, thereby creating slots for the numbers 0 through 99. (Indices 0 and 1 are not used, but it is less confusing to include them.)

```
i       dat 2               ; i
num     dat 200             ; an array indexed 0 through 99;
                            ;   indices 0 and 1 are not used
```

The first loop in the sieve is easy to write. We use `r3` for `i` and `r1` for the upper bound (100, in this case).

```
        add r3 r0 2         ; i = 2;
        add r1 r0 100       ; limit = 100;

        ble r1 r3 enda      ; check if done
loopa
        sal r2 r3 1         ; double to get word offset
        sto r3 r2 num       ; num[i] := i
        add r3 r3 1         ; i++;
        blt r3 r1 loopa     ; go back for another round
enda
```

The body of this loop has four instructions and uses all three data registers. Can you find a way to reduce the number of instructions to three and use only two registers?

Let us now look carefully at the structure of the second loop.

```
for (i = 2; i < limit; i++)
    if (num[i] != 0) {
        write(i);
        for (j = i; j < limit; j += i)
            num[j] = 0;
    }
```

Call the outer loop `loopb`. The body of that loop consists of an if-statement whose then-part contains another loop, `loopc`. The structure of the labels is shown below.

```
        ; set up outer loop
        ; conditional branch to endb
loopb
        ; branch to endif when (num[i] != 0)
        ; write(i)
        ; set up inner loop
        ; conditional branch to endc
loopc
        ; body of inner loop
        ; increment j
        ; conditional branch to loopc
endc
        ; clean up after inner loop
endif
        ; increment i
        ; conditional branch to loopb
endb
```

The structure of the labels is a little complicated, in part because we have lost the nested structure of the Java-like code, but it is simpler than it might be because the else-part is empty. The outer loop has the same structure as `loopa` above, so we can copy that pattern.

Our register conventions are (almost) the same. We use `r1` for the limit and `r3` for the loop index (either `i` or `j`). We use `r2` for two purposes: to temporarily store `2j` and to hold `i` when it is used as an increment for `j`. That means we must store `i` in memory when it is not being used, and that is why we reserved space for it at the top of our program. Here us the final version of the second loop.

```
                                ;;; set up outer loop
        add r3 r0 2             ; i = 2
        add r1 r0 100           ; limit = 100
        ble r1 r3 endb          ; conditional branch to endb
loopb
                                ;;; body of outer loop
        sal r2 r3 1             ; double to get word offset
        loa r2 r2 num           ; load num[i]
        beq r2 r0 endif         ; branch to endif when num[i] == 0
        sto r3 r0               ; write(i)

                                ;;; set up inner loop
        sto r3 r0 i             ; store i temporarily
                                ;     we now use r3 for j
        ble r1 r3 endc          ; conditional branch to endc
loopc
                                ;;; body of inner loop
        sal r2 r3 1             ; get address of num[j]
        sto r0 r2 num           ; make num[j] zero
        loa r2 r0 i             ; recover i
        add r3 r3 r2            ; increment j
        blt r3 r1 loopc         ; conditional branch to loopc
endc
                                ;;; clean up after inner loop
        loa r3 r0 i             ; restore i into r3
endif
                                ;;; finish the outer loop
        add r3 r3 1             ; increment i
        blt r3 r1 loopb         ; conditional branch to loopb
endb
```

Even though none of the steps is hard, the program is intricate. Programming at this level is a delicate task that requires careful attention. All the pieces are assembled in the example program `sieve.a52`.

Read over the block of code above—as many times as are necessary to become comfortable with it. As you do, you may see opportunities for improvement. For example, by reversing the roles of `r2` and `r3` in the inner loop, we can eliminate at least one instruction. Similarly, we can avoid doubling the indices by incrementing them by two each time. (We did not make either of those changes here because they would have made the loop even more enigmatic than it already is. It is always more important for a program to be correct than to be fast.)

2.6 The Organization of Memory

Until now, we have been focussed on the careful ordering of instructions, and we have been rather vague (or even sloppy!) about where in memory variables are stored. We have only three registers, and most programs will have more than three variables. It is the programmer's responsibility to decide where in memory the variables are stored.

Memory in a modern computer is arranged into four basic areas:

- The *code* consists of the block of memory where the instructions are stored. On the CS52 Machine, the assembler insists that all the instructions reside in one contiguous block of memory.
- The *data region* is where the global data are stored. For most of our programs on the CS52 Machine, this area will be rather small or non-existent. It may appear either before or after the code.
- The *stack* is a region devoted to managing subprogram calls. Subprograms are also called methods, functions, or procedures. The next few sections focus on the use of the stack.
- The *heap* is where dynamic objects are stored. When you use the keyword `new` in Java, you are allocating memory on the heap for a new object. The object is "dynamic" in the sense that memory is allocated as the program runs and is in use for an indeterminate amount of time. The management of the heap is rather complicated, and we will not discuss heaps in this course.

The boundaries between the various regions are maintained by the programmer. The CS52 Machine knows nothing about them. It is possible for a badly written program to try to execute a word that was intended as data or to use an instruction word as data. The part of the stack that is in use grows and shrinks as a program executes, and it is possible that it will outgrow the space allocated for the stack and overwrite part of the program. You will no doubt encounter such situations as you write recursive programs.

Just as we use `ic` to provide a "current location" in the code region, we will adopt a convention that `r1` always contains an address in the stack region. It will be our "anchor" for data. Our programs will allocate a space in memory for the stack just after the code region. The stack will grow upward, from high addresses to low.

2.7 Stacks

In the Sieve of Eratosthenes program, we allocated space to save the variable `i` when it could not be held in a register. That method for saving and restoring the value of a variable is `ad hoc` and difficult to generalize. Most computer systems save temporary values on a stack. It is a mechanism that can be generalized to support subprogram calls.

The Stack as an Abstract Data Type A *stack* is an data structure that maintains a collection of elements. For our purposes, the elements may be taken to be words on the CS52 Machine, but in other contexts, any kind of element could be used.

The name "stack" comes from an analogy with a stack of dishes or cafeteria trays. Dishes may be placed on the top of the stack or removed from the top of the stack, but we may not—without disastrous consequence—move dishes that are in the middle of the stack. All activity occurs at the top of the stack. The object removed is the one that most recently was added, leading to a discipline that is called LIFO, an abbreviation for "last-in-first-out."

A stack is *empty* when it contains no elements. If a stack is not empty, we may *pop* the top element off the stack. The pop operation is considered to be a function whose value is the element that is removed. In some situations it is convenient to have a *top* operation that returns the value of the top element without removing it. An attempt to pop an element from an empty stack causes an error known as *stack underflow*.

Conversely, a *push* operation places an element on the top of the stack. Usually, an implementation of a stack will have a bounded capacity, and a push operation that would cause the capacity to be exceeded is an error known as *stack overflow*.

As general data structures, stacks can be implemented using arrays or linked lists, but most processors have some direct hardware assistance for the stacks that support the execution of subprograms.

The Hardware Stack A stack is at the heart of a running program. A *hardware stack* consists of a region of memory reserved for the elements and a designated register to hold an address. The register has a different name in every architecture; we will call it `sp`, for *stack pointer*. The unit of data for our hardware stack is the 16-bit word.

Paradoxically, most hardware stacks grow from high memory addresses to low, so that a push operation *decreases* the stack pointer. A push operation

1. copies the value from a register to `mem[sp]`, and then
2. decrements `sp` by the word size (in our case, by 2).

Analogously, a pop operation

1. increments `sp`, and then
2. copies the value from `mem[sp]` to a register.

Here, we have adopted the convention that `sp` is the location *just below* the top of the stack. It is the location where the next element is to go. We could have equally well set things up so that `sp` pointed directly to the top of the stack. (How?)

Stack overflow, in the case of the hardware stack, means that the stack grows beyond the region of memory that was reserved for it. There is no direct protection against corrupting data or programs in adjoining memory locations. Similarly, there is no protection against stack underflow, in which erroneous values will be returned. A correct program will prevent underflow by invoking the operations in push-pop pairs.

One use of stacks is to hold intermediate values in arithmetic calculations. For example, in computing

$$(a + b) * (c + d),$$

the value $a + b$ is computed and pushed onto the stack. Then $c + d$ is computed, and the value $a + b$ is popped for multiplication.

Another, similar, use is to temporarily store the value from a register. We could have used that device in the Sieve of Eratosthenes when we had more variables than registers.

One cost of using a stack is that the stack pointer cannot be used for any other purpose. We adopt the convention that `r1` is the stack pointer, leaving us with only two general-purpose registers.

Stack Frames One of the most important uses of the hardware stack is to manage subprograms. When a subprogram is invoked, there are two agents: the *caller* of the subprogram, which is suspended while the subprogram is executing, and the subprogram itself, the *callee*. When the callee completes its task, the caller resumes where it left off.

During the course of its execution, a subprogram may call another subprogram. The calling mechanism obeys a stack discipline. The active subprogram is on top of the stack, and all the suspended callers are further down. When a subprogram is finished, it is popped and the new top-of-stack, its caller, becomes active. The LIFO stack discipline is exactly what we need to keep track of subprograms.

A *stack frame* is a block of data on the stack that is used for communication between the caller and callee. It includes space for the arguments that are passed from the caller to the callee; the function result, if any, that is passed back from the callee to the caller; and the address in the caller's code to which execution should return. The frame on the top of the stack belongs to the currently-executing subprogram. A stack frame is also known as an *activation record*.

There are many variations for maintaining the information in a stack frame. Different subprograms may have stack frames of different sizes. It does not matter much what conventions one adopts, but once the decisions are made, strict adherence to them is vital. We adopt the following conventions for using registers and the stack on the CS52 Machine.

- `r1` is the stack pointer. A subprogram call should return it with the same value. (A rare exception occurs when a subprogram returns more than one value. In that case, there are additional values on the stack.)
- `r2` is used for the return address. The caller cannot expect any data in `r2` to be preserved.
- `r3` is used to pass the first argument to the callee and to return a value to the caller. Again, the caller cannot expect its value to be preserved.
- Any arguments beyond the first are pushed onto the stack by the caller, who is also responsible for removing them from the stack.

Typically, the caller will save the values in `r2` and `r3`, if necessary, by pushing them onto the stack. It will then put the first argument into `r3` and push any additional arguments onto the stack. It will then place the address of the subprogram in `r2` and execute

```
        cal r2 r2
```

to invoke the subprogram. One effect of the `cal` instruction is to save the return address in `r2`. The first act of the callee will be to save the return address by pushing it from `r2` onto the stack. When it is finished, the subprogram will put the value to be returned into `r3`, pop the return address back into `r2`, and jump back to the caller.

```
subprog
        psh r2              ; save the return address
        ...                 ; compute, and
        ...                 ;     leave the result in r3
        pop r2              ; restore the return address
        jmp r2              ; return
```

If the subprogram has any local variables, space is allocated for them after the return address is pushed, and they are discarded by the callee just before the return address is popped.

Initializing the Stack A program must reserve an area in memory for the stack, and its very first act must be to initialize the stack pointer `r1`. Remember that a stack grows from high locations toward lower ones, so the base of the stack is at the high end of the stack region. By convention, we put global data region *before* the executable code and the stack area *after* it.

```
data
        dat ??              ; data area, if required

                            ; beginning of executable code
        lcw r1 stack        ; initial stack pointer

        ...                 ; rest of executable code

        dat 100             ; stack area, 50 words
stack
        end
```

2.8 Subprograms

The facility to use subprograms is an essential part of any modern programming language. Subprograms might be called functions, procedures, or subroutines, but at the hardware level, they all behave in about the same way.

Now that we have a stack, we have the ability to call subprograms. There is, of course, no reason that the caller and callee have to be different subprograms, and we illustrate subprogram calls by writing a recursive version of the factorial function. Following convention, the argument and the result are both passed in `r3`. If we forget about the base case of the recursion (only for a moment!), the code is astonishingly simple.

```
fact
        psh r2              ; save return address
        psh r3              ; save argument
        sub r3 r3 1         ; decrement argument

        lcw r2 fact         ; make recursive call
        cal r2 r2           ;

        lcw r2 mult         ; call mult
        cal r2 r2           ;     one argument is in r3
                            ;     and the other is on
                            ;     the stack
```

```
pop r0                  ; discard saved argument
pop r2                  ; restore return address
jmp r2                  ; return
```

Here we are assuming that there is a subprogram `mult` which does multiplication. When it is called, one argument to `mult`—the result of the recursive call—is in `r3`, and the other argument is on the top of the stack. When it returns, the result of the factorial function is already in `r3`. No movement of data is necessary.

Returning to the base case of the factorial function, we must make the result 1 when the argument is less than 1. It is an easy insertion of an if-then-else construction, and the completed factorial function appears in context in the sample program `factorial.a52`, listed in Section C.4.

Notice that stack operations come in push-pop pairs. That discipline is the key to using a stack. *Always* follow the template and be sure that your subprogram balances pushes and pops.

2.9 Digging More Deeply into the Stack

Often, we are interested in a value that is near, but not at, the top of the stack. We can gain access to such a value using the optional third argument to the `loa` instruction.

```
loa r2 r1 offset
```

After the return address has been pushed, an offset of 2 would recover the return address itself, and an offset of 4 would recover the next word down on the stack, presumably an argument.

The subprogram `mult`, used in the factorial example, takes two arguments and uses them to initialize two local variables. It starts and ends like this:

```
psh r2                  ; save the return address
loa r2 r1 4             ; get argument b
psh r3                  ; make a local copy of a
psh r2                  ; make a local copy of b

...                     ; compute, and place the
                        ;    result in r3

pop r0                  ; discard local b
pop r0                  ; discard local a
```

```
        pop r2              ; restore return address
        jmp r2              ; return
```

When this subprogram has completed, the argument `b` is on the top of the stack; the caller is responsible for removing it.

Within the heart of `mult`, the product is computed using repeated addition, just as it was in our earlier example. However, we now have only two registers available because `r1` is reserved for the stack. We keep the result in `r3`, and we alternate the use of `r2` between `a` and `b`. Here is the loop.

```
loopm
        loa r2 r1 2         ; recover b
        add r3 r3 r2        ; product += b;
        loa r2 r1 4         ; recover a
        sub r2 r2 1         ; a--;
        sto r2 r1 4         ; store a
        blt r0 r2 loopm     ; go back for another roundÄ
```

The sequence involving the decrement of `a` is common: put the value in a register, modify it, and then put it back in memory. We do not have to store `b` because its value is not changed. The complete function `mult` appears in the example program `factorial.a52`, listed in Section C.4. A much better multiplication function appears in the library `mullib.a52`, listed in Section C.5.

3 Assembly Language Reference

After reading the examples and discussion of assembly language programs in the previous section, you are no doubt familiar with many of the instructions and their use in programs. This section provides, for reference, an unambiguous description of the CS52 Machine's assembly language.

Informally, a CS52 Machine assembly language program is a sequence of lines. Each line may contain up to three optional components, in order:

- A label. Each label is a string of characters that begins on the left margin, right at the beginning of the line.
- Whitespace followed by an instruction or directive. Each instruction or directive consists of a three-letter abbreviation followed by up to four arguments. There are 35 different instructions and two directives. The number and nature of the arguments are dependent on the particular instruction and are specified in Section 3.2.
- A semicolon followed by a comment. The semicolon and everything following it are ignored by the assembler.

The assembler translates the instructions into a sequence of 16-bit words that can be executed by the CS52 Machine. See Section 4 for details on the native machine instructions and how the assembly language instructions are translated.

3.1 Assembly Language Syntax

The formal syntax for the CS52 Machine's assembly language is presented in Table 1 using Extended Bachus Naur Form (EBNF). An EBNF specification is a formal method of specifying a substitution closely related to a context-free grammar. We will have more to say about EBNF later in the course.

For now, it is enough to say that the symbols in brackets, like ⟨line⟩, are intended to be the targets of substitution. One may substitute a symbol on the left side of ::= with the construction on the right side. On the right,

- | means "or,"
- [...] means "optional," and {...} means "repeated zero or more times."

A few of the substitutions have been omitted. For example, ⟨eoln⟩ is the end-of-line symbol, and ⟨whitespace⟩ is any non-empty sequence of blanks spaces or tab characters. All the commands in the language are listed in Table 2.

The EBNF syntax tells us quite a bit about the assembly language, but some details are left out. In particular, the syntax does not tell us how labels are used or exactly what kinds of arguments a particular command takes.

⟨program⟩	::=	{⟨line⟩⟨eoln⟩}
⟨line⟩	::=	[⟨label⟩] [⟨instruction⟩] [⟨whitespace⟩] [; ⟨comment⟩]
⟨instruction⟩	::=	⟨whitespace⟩⟨command⟩ {⟨whitespace⟩⟨arguments⟩}
⟨command⟩	::=	`nop` | `hlt` | `cal` | ...
⟨arguments⟩	::=	[⟨register⟩ [⟨register⟩ [⟨register⟩]]] [⟨argument⟩]
⟨register⟩	::=	`r0` | `r1` | `r2` | `r3`
⟨argument⟩	::=	⟨label⟩ | ⟨decimal number⟩ | ⟨hex number⟩
⟨label⟩	::=	⟨letter⟩ {⟨letter⟩ | ⟨digit⟩ | ⟨underscore⟩}
⟨decimal number⟩	::=	[`-`] ⟨digit⟩ {⟨digit⟩}
⟨digit⟩	::=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨hex number⟩	::=	[`-`] `0x` ⟨hex digit⟩ {⟨hex digit⟩}
⟨hex digit⟩	::=	⟨digit⟩ | `a` | `b` | `c` | `d` | `e` | `f`
⟨whitespace⟩	::=	⟨blank⟩ {⟨blank⟩}

Table 1: The EBNF specification for the CS52 Machine’s assembly language.

⟨command⟩	::=	⟨action⟩ | ⟨directive⟩
⟨action⟩	::=	`adc` | `add` | `anc` | `and` | `ari` | `beq` | `bge` | `bgt` | `ble` | `blt`
		`bne` | `brs` | `cal` | `hlt` | `jmp` | `lch` | `lcw` | `loa` | `mov` | `neg`
		`nop` | `not` | `orc` | `orr` | `pop` | `psh` | `sal` | `sar` | `sft` | `sll`
		`slr` | `sto` | `sub` | `xoc` | `xor`
⟨directive⟩	::=	`dat` | `inc`

Table 2: A listing of all the commands in the CS52 Machine’s assembly language.

Labels The label, if it exists on a line, begins in the left column. Although there is no entry for labels in the EBNF of Table 1, one is not hard to write. A label must begin with a letter and contain only letters, digits, and underscore characters. The assembly language is not case-sensitive.

A label that begins in the first column is called the *declaration* of the label. A label may be declared only once. A label is an anchor, an abbreviation for the location in the program where it is declared.

A label may be *used* as a byte or word argument anywhere in the program. It is not necessary that declaration come before use. Labels are frequently used with the branch instructions, in which case the assembler computes the offset from the current instruction to the location of the label and inserts that offset into the branch instruction. Another use of a label is as an argument to the `lcw` instruction: the location specified by the label is loaded into a register.

The branch instructions and `lcw` are nearly always used with labels. Other instructions normally have numerical values for their byte arguments.

Commands and Arguments A command is a three-letter assembly language instruction or one of the *directives* `dat` or `inc`. An assembly language command must be followed by arguments appropriate for that instruction. For example, `cal` must be followed by exactly two registers chosen from among `r0`, `r1`, `r2`, and `r3`; and `add` must be followed either by three registers or by two registers and a byte argument. Byte or word arguments can usually be numeric values or labels, but there are a few cases in which labels are not allowed. See the next section for details on the actual arguments taken by each of the commands.

The directive `dat` reserves space for data; no instructions appear in a data area. The argument to `dat` is an integer indicating the number of bytes to reserve. A data area may be composed of any number of consecutive `dat` directives, but the data areas must lie before or after the executable code. A data area may not appear between actual CS52 Machine instructions.

Comments A semicolon starts a comment that lasts to the end of the line. The semicolon and the remainder of the line are ignored by the assembler.

3.2 Assembly Instructions

Each CS52 Machine command takes between zero and four arguments, the type of which are specified by single letters, decoded as follows:

`R`	register
`S`	signed byte
`U`	unsigned byte
`W`	full word constant
`B`	signed byte, interpreted as an offset from the current position
`A`	6-bit constant
`F`	string

The numerical quantities may be specified by decimal or hexadecimal constants, or (usually) by labels denoting locations in the program.

Arithmetic and bitwise logical instructions These instructions are present:

$$\left.\begin{array}{l}\texttt{add}\\ \texttt{sub}\\ \texttt{and}\\ \texttt{orr}\\ \texttt{xor}\end{array}\right\}\ \texttt{RRR}\ \text{or}\ \texttt{RRS}$$

The arguments can be three registers, or two registers and a signed byte argument. The assembler distinguishes between the two cases and generates the correct machine instructions. The byte argument, if present, must be a single byte; it can be specified as a number or a label. It is sign-extended to a full word at execution time. The operation is done on the second and third arguments, and then stored in the first argument.

There are three two-register instructions:

$$\left.\begin{array}{l}\texttt{mov}\\ \texttt{neg}\\ \texttt{not}\end{array}\right\}\ \texttt{RR}$$

The `mov` instruction simply copies a value from the second register into the first register. The `neg` instruction takes a value from the second register and places its two's-complement negation in the first register. The `not` instruction places the bitwise complement of the value from the second register into the first register.

Shift instructions The CS52 Machine is capable of arithmetic and logical shifts—either left or right:

$$\left.\begin{array}{l}\texttt{sll}\\ \texttt{slr}\\ \texttt{sal}\\ \texttt{sar}\end{array}\right\}\ \texttt{RRR}\ \text{or}\ \texttt{RRS}$$

As in the case of the arithmetic instructions, the shift instructions can take three registers, or two registers and a byte argument. The third register or byte argument specifies the shift distance.

The byte argument, if present, must be a number between 0 and 16, inclusive. It cannot be a label.

If the shift distance comes from a register, any value is allowed. Negative distances result in shifts in the opposite direction. When the absolute value of the shift distance is greater than 15, the result will be a word with all zeroes or all ones.

Memory and data-moving instructions These instructions move values back and forth between registers and memory:

$$\left.\begin{array}{l}\texttt{sto}\\\texttt{loa}\end{array}\right\}\texttt{RRS}$$

The instructions take two registers and an optional byte argument. The byte argument defaults to zero when it is not present. An address is computed as the sum of the second register and the sign-extended byte argument. In the case of `sto`, the value in the first register is stored at the address in memory. In the case of `loa`, the value at the address in memory is copied into the first register.

The byte argument to `sto` or `loa` can be a number or a label; most frequently it is a number.

There is an instruction that loads a full-word constant into a register:

```
lcw RW
```

It takes a single register and a constant. The constant can be a number or a label. The assembler (usually) expands the instruction into two machine instructions.

We also have the familiar instructions for manipulating the stack:

$$\left.\begin{array}{l}\texttt{psh}\\\texttt{pop}\end{array}\right\}\texttt{R}$$

The assembler adheres to the CS52 Machine convention of using register `r1` as the stack pointer. The instructions take a single register, the one containing the value to be pushed or the one receiving the value to be popped. The assembler expands each of these instructions into two machine instruction.

Control instructions The CS52 Machine contains these familiar short branch instructions:

```
brs  B
```

$$\left.\begin{array}{l}\texttt{beq}\\ \texttt{bne}\\ \texttt{blt}\\ \texttt{bge}\\ \texttt{bgt}\\ \texttt{ble}\end{array}\right\}\ \texttt{RRB}$$

The unconditional branch instruction `brs` takes no register arguments. The others take two registers whose values are compared to decide whether or not to take the branch.

All the branch instructions take a signed byte argument. At the machine level, the argument is an offset from the address of the instruction immediately following the branch. In assembly language, the argument can be a number or a label. If it is a number, it is taken as the literal offset. If it is a label, the assembler computes the offset from the next instruction to the location of the label in the program.

Beware of using numbers as arguments to branch instructions. There are cases in which the assembler may emit more than one native instruction for a single assembly instruction, so making an accurate count of instructions is difficult. Further, the count may change as you develop your program.

Branches are intended to move short distances in the program. The signed byte argument takes values between -128 and 127, the offset must be even, and each instruction takes two bytes. Hence it is possible to move only 64 instructions backward and 62 bytes forward.

There are two instructions that support subprograms:

```
cal RR
jmp R
```

The `cal` instruction takes two registers: The first is the destination that receives the "return address," the location of the instruction immediately following the `cal`. The second register is the target of the call; it becomes the location of the next instruction to be executed by the machine.

The CS52 Machine follows the convention of using register `r2` for calling subprograms. Thus, `cal r2 r2` simply interchanges the values in `r2` and `ic`.

The `jmp` instruction takes one register and transfers control to the instruction at the address specified in the register. It does not save a return address.

There are a few familiar instructions for doing nothing or stopping the computation:

```
nop
hlt A
```

The instruction `nop` takes no arguments and does nothing except advance the instruction counter. The instruction `hlt` with no argument or a zero argument causes the CS52 Machine to halt in a normal error-free state. With a non-zero argument, `hlt` causes the CS52 Machine to halt in an error state specified by the argument. That argument must be a number between 1 and 63 inclusive. See page 30 for a list of the current conventions for interpreting the error codes.

Machine instructions These instructions are part of the CS52 Machine's instruction set:

```
ari RRRA
adc RRS
anc RRS
orc RRS
xoc RRS
sft RRS
lch RU
```

The assembler makes good use of them, but it is unlikely that a human will write them. See the next section for a description of the CS52 Machine's native language.

Directives There are two directives:

```
dat W
inc F
```

The `dat` directive is used to make space for data. It creates a blank space either before or after the image of the program in memory. It must have one positive numerical argument.

The assembler for the CS52 Machine insists that all executable code be in one continuous block, so a `dat` directive may not appear in the middle of a program.

The `inc` directive is used to load instructions from an external file. It allows one to load routines for operations like multiplication and division from libraries. The argument to `inc` must be a string containing only lowercase letters. The assembler adds the extension `.a52` and looks for a file with that name in the same directory as the original source file.

Libraries may be nested, up to four deep. The number of open files, including the original source file, can be no greater than four. This limit is a simple, if inelegant, way of preventing infinite “include loops.” It is hard to imagine a situation in which anyone would want to nest files even that deeply. There is no limit—apart from good judgment—on the number of libraries that can be included *sequentially.*

4 Machine Language Reference

A machine-level instruction is a word whose bits encode an atomic action of the machine. An instruction encodes the specific operation and up to four arguments. All the arguments, except perhaps the last, are registers. The registers `r0` through `r3` are specified with two bits each. There are two instruction formats:

	4	2	2	2	6
Format I:	`opcode`	`rX`	`rY`	`rZ`	`auxcode`

	4	2	2	8
Format II:	`opcode`	`rX`	`rY`	`argument`

`high-order bits` ⟵ ⟶ `low order bits`

The type of instruction, called the opcode, is encoded in the four high-order bits of the word. There are sixteen possible opcodes: one is unused, and the rest correspond to some of the assembly instructions we have already seen. The remaining assembly instructions are translated into machine instructions by the assembler. Not every opcode uses all its fields.

4.1 The Native Instruction Set

Here is a complete list, in opcode order, of all the instructions. It includes the format of each instruction, the fields used by the instruction, and the restrictions on the values in the fields.

The branch instructions

`beq`	Format II	`0x0`	`rX`	`rY`	`displacement`
`bne`	Format II	`0x1`	`rX`	`rY`	`displacement`
`blt`	Format II	`0x2`	`rX`	`rY`	`displacement`
`bge`	Format II	`0x3`	`rX`	`rY`	`displacement`

Each branch instruction compares the values in the two specified registers. If the condition (equal, unequal, less-than, greater-or-equal) is satisfied, the displacement argument is added, as a sign-extended byte, to the instruction counter. Remember that the instruction counter is the address of the instruction *immediately after* the branch instruction.

Because the value in the instruction counter must be even, the displacements are limited to even values.

The call instruction

`cal`	Format I	0x4	rX	rY	0	0

The call instruction specifies two registers, `rX` and `rY`. The current value of the instruction counter—the return address—is copied into `rX`, and the value in `rY` is copied into the instruction counter. The two registers may be the same, so the copies must happen simultaneously.

The third register field and the auxcode field must both be zero.

The halt instruction

`hlt`	Format I	0x5	0	0	0	auxcode

The halt instruction stops the computation. All three register fields must be zero. The auxcode is viewed as a 6-bit unsigned integer, taking values from zero through 63. The CS52 Machine assigns meaning to some of the possible auxcodes:

`0x00`	normal error-free termination
`0x01`	unimplemented instruction
`0x02`	instruction format is illegal
`0x03`	instruction counter is not even
`0x04`	memory address is out of bounds
`0x05`	memory size request is out of bounds
`0x06`	user input is not a signed number
`0x07`	machine code file is incorrect
`0x20`	attempt to divide by zero

All but the last of these errors are raised automatically by the CS52 Machine when it encounters a problem. The programmer may raise a "divide by zero" error when a division routine encounters a zero divisor. The programmer is free to use other auxcode values to signal other kinds of errors.

The arithmetic instruction

`ari`	Format I	0x6	rX	rY	rZ	auxcode

This is the classic three-register instruction that carries out arithmetic and bitwise logical operations. The values in `rY` and `rZ` are combined using the operation specified by the auxcode. The result is placed in `rX`. These are the legal auxcode values:

`0x0`	addition
`0x1`	subtraction
`0x4`	bitwise and
`0x5`	bitwise or
`0x6`	bitwise xor
`0x8`	shift logical left
`0x9`	shift logical right
`0xa`	shift arithmetic left
`0xb`	shift arithmetic right

The shift instruction

`sft`	Format I	`0x7`	`rX`	`rY`	`0`	`auxcode`

The shift instruction combines two kinds of shifts, in two directions. The value in `rY` is shifted and placed in `rX`. The two most significant bits of the auxcode specify the kind of shift:

`0b00`	logical left
`0b01`	logical right
`0b10`	arithmetic left
`0b11`	arithmetic right

The other four bits of the auxcode determine the shift distance; it is $d + 1$, where d is the unsigned value of the four least significant bits of the auxcode. There is no way to specify a shift distance of zero; the assembler translates a shift of zero into a `mov` instruction.

The third register field must be zero.

The constant-argument instructions

`adc`	Format II	`0x8`	`rX`	`rY`	`argument`
`anc`	Format II	`0x9`	`rX`	`rY`	`argument`
`orc`	Format II	`0xa`	`rX`	`rY`	`argument`
`xoc`	Format II	`0xb`	`rX`	`rY`	`argument`

These instructions specify two registers and a signed byte. A value is created by applying an operation (addition, bitwise and, bitwise or, or bitwise xor) to the value from `rY` and the sign-extended byte. That value is placed into `rX`.

An illegal instruction

0xc	anything

The instruction whose opcode field is `0xc` is unused. An attempt to execute it will result in an "illegal instruction" error.

The high-byte instruction

`lch`	Format II	0xd	rX	0	argument

This instruction replaces the most significant byte of `rX` with the instruction's byte argument. The least significant byte of `rX` is left unchanged. The unused register `rY` must be specified as zero. The assembler uses `lch` in conjunction with `adc` to create full word constants.

The second register field must be zero.

The memory instructions

`sto`	Format II	0xe	rX	rY	argument
`loa`	Format II	0xf	rX	rY	argument

These are the instructions that interact with memory. The byte argument is sign-extended and added to the value in `rY` to create an address. In the case of `sto`, the value in `rX` is copied to memory at that address. In the case of `loa`, the value from memory at that address is placed in `rX`.

4.2 Translation of Assembly Instructions

The assembler translates its instructions into the native machine language. Here is a description of the translation, in the same order that the assembly instructions were described in Section 3.

Arithmetic and bitwise logical instructions The three-register versions of the instructions are translated as follows.

`add rX rY rZ` is translated as `ari rX rY rZ 0x0`.

`sub rX rY rZ` is translated as `ari rX rY rZ 0x1`.

`and rX rY rZ` is translated as `ari rX rY rZ 0x4`.

`orr rX rY rZ` is translated as `ari rX rY rZ 0x5`.

`xor rX rY rZ` is translated as `ari rX rY rZ 0x6`.

The assembler determines whether it is in the three-register or two-register case. The two-register-and-constant versions of the instructions are translated as follows.

`add rX rY arg` is translated as `adc rX rY arg`.

`sub rX rY arg` is translated as `adc rX rY -arg`.

`and rX rY arg` is translated as `anc rX rY arg`.

`orr rX rY arg` is translated as `orc rX rY arg`.

`xor rX rY arg` is translated as `xoc rX rY arg`.

The two-register-without-constant instructions are translated as follows.

`mov rX rY` is translated as `ari rX r0 rY 0x0`.

`neg rX rY` is translated as `ari rX r0 rY 0x1`.

`not rX rY` is translated as `xoc rX rY 0xff`.

Shift instructions The shift instructions can also come in two forms.

`sll rX rY rZ` is translated as `ari rX rY rZ 0x8`.

`slr rX rY rZ` is translated as `ari rX rY rZ 0x9`.

`sal rX rY rZ` is translated as `ari rX rY rZ 0xa`.

`sar rX rY rZ` is translated as `ari rX rY rZ 0xb`.

Remember that the constant argument to a shift operation must be a value between 0 and 16 inclusive. A shift by 0 bits does not change a value, so in all four cases it is translated into a `mov` operation.

`s??  rX rY 0` is translated as `ari rX r0 rY 0x0`.

When the argument is non-zero, we let `0bdddd` be the binary representation of `arg` − 1.

`sll rX rY arg` is translated as `sft rX rY 0b00dddd`.

`slr rX rY arg` is translated as `sft rX rY 0b01dddd`.

`sal rX rY arg` is translated as `sft rX rY 0b10dddd`.

`sar rX rY arg` is translated as `sft rX rY 0b11dddd`.

Memory and data-moving instructions The `sto` and `loa` instructions are part of the native instruction set; they are left unchanged, except that the assembler supplies the default value of zero when the argument is not present.

The push and pop instructions assume that register `r1` is the stack pointer. They are *synthetic instructions,* in the sense that each one expands into two native instructions.

`psh rX` expands to `sto rX r1 0` followed by `adc r1 r1 0xfe`.

`pop rX` expands to `adc r1 r1 2` followed by `loa rX r1 0`.

The `lcw` instruction usually expands into two instructions, one to load the low byte and then one to load the high byte. The assembler does the computation to split the argument `arg` into its low and high bytes. Let `low` = `arg`&`0xff` and `high` = `arg`>>> 8. By >>> we mean the logical right shift operation.

`lcw rX arg` expands to `adc rX r0 low` followed by `lch rX high`.

When the argument is a constant value in the range of a signed byte, the second instruction is unnecessary, and the assembler omits it.

Control Instructions When the argument to a branch instruction is a constant, that constant is taken literally. When it is a label, the signed distance to the label from the next instruction (the one following the branch) is computed, and that value is taken as the argument. In the following specifications, we assume that the argument has already been adjusted, if necessary.

The branch instructions `beq`, `bne`, `blt`, and `bge` are part of the native instruction set and are left unchanged.

`brs arg` is translated as `beq r0 r0 arg`.

`ble rX rY arg` is translated as `bge rY rX arg`.

`bgt rX rY arg` is translated as `blt rY rX arg`.

The `cal` instruction is part of the native instruction set and is not translated.

`jmp rX` is translated as `cal r0 rX`.

`nop` is translated as `beq r0 r0 0`.

Machine instructions The remaining assembly instructions are native instructions and are assembled directly, without translation.

5 Real Computers

The CS52 Machine contains many of the features of a real computer, at least in primitive form. In this section, we explore how the CS52 Machine concepts are extended in real computers.

5.1 Memory and Registers

As in the CS52 Machine, memory in a real computer is indexed by bytes, but the word size is larger. The 32-bit word was the standard for quite a while but is being replaced with 64-bit words.

A real computer has more registers, seldom less than eight and often 32 or more. Sometimes, as in the Intel x86 architecture, every register has a special purpose.

5.2 Instructions

The instruction set on a real computer includes, of course, many more arithmetic operations. There are also more varied and flexible addressing methods for the store and load operations. There are instructions to manipulate a wider range of data types, including characters and floating point numbers. Finally, there are specialized instructions that are used by the operating system; see Section 5.5.

Designing an instruction set requires careful thought. Instructions should be chosen to give the programmer maximum flexibility and expressiveness. The particular instructions should be the ones most frequently used. They should be grouped according to form and function for easy decoding by the processor.

The CS52 Machine instruction set is quite uniform: Every instruction is exactly two bytes long. The first four bits determine the instruction, and there are only two instruction formats, as described in Section 4. (Many instructions disregard some, or all, of their arguments, however. In those cases, the bits encoding the arguments are ignored.)

An alternative to fixed-length instructions is a variable-length instruction set. The Intel x86 instruction set, currently the most popular, contains instructions with lengths from one byte up to fifteen bytes. An advantage of variable length instructions is that they make the code stream very compact; few bits are wasted. The most commonly-used instructions can be made short.

Variable-length instruction sets were chosen in the 1970's when the x86 processors were first designed because they increased the speed of the computer. Frequently-used combinations of instructions (like `pop r2`; `jmp r2` on the CS52 Machine) could be combined into a single short instruction and executed as a single instruction on the hardware. Further, programs expressed in variable-length instructions tend to be more compact—a consideration that was important when memory was expensive.

Later, in the 1980's, new processors used uniform, fixed-length instructions and took advantage of the speed at which they could be decoded. The tension between CISC (complex instruction set computers) and RISC (reduced instruction set computers) was high, and people argued the superiority of both approaches. The capabilities of today's processors make arguments about instruction sets moot. Complex instructions can be quickly decoded and executed, which is probably why we still use a descendant of the original x86 instruction set.

Figure 3 contains the Sieve of Eratosthenes in Intel x86 assembly language. The code was generated by a compiler, and the branches are a little different from our example. Deciphering the code is an interesting exercise. To get you started, we have annotated the first loop. We will refer back to this code in subsequent sections.

5.3 Conditional Branches

The conditional branch instructions on the CS52 Machine make *signed* comparisons. Unsigned comparisons can be different. For example, `0xffff` is less than zero as a signed word, but not as an unsigned word. There are cases in which we want to make unsigned comparisons.

One solution would be to emulate the comparisons in software; see Exercise 6. Such solutions involve complicated case-by-case analyses and would give unacceptable performance on any real computer. Another solution would be to add unsigned comparison instructions, but that greatly increases the complexity of the instruction set.

A common solution is to use subtraction and add four one-bit registers, called *flags*. The flags are modified on every arithmetic operation, and the conditional branch instructions examine the flags, not the contents of registers. A branch instruction like `ble rX fY target` would then, in three steps, compute `rX` − `rY`, discard the result of the computation, and then determine the branch based on the flags.

A flag with the value 0 is said to be clear or "false," and one with value 1 is set or "true."

- The *carry flag* is set when an addition causes a carry bit to be discarded, or when a subtraction requires a borrow bit.
- The *zero flag* is set if the result of the arithmetic operation is zero.
- The *sign flag* is set if the result, interpreted as a signed value, is negative. The sign flag is simply the sign bit of the result.

```
_sieve:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $440, %esp
        movl    $2, -16(%ebp)           ; i = 2;
        jmp     L2                      ; jump to loop test
L3:
        movl    -16(%ebp), %eax         ; eax = i;
        movl    $0, -416(%ebp,%eax,4)   ; num[i] = 0;
        leal    -16(%ebp), %eax         ; eax = address(i);
        incl    (%eax)                  ; i++;
L2:
        cmpl    $100, -16(%ebp)         ; compare i and 100
        jle     L3                      ; branch back if i <= 100
        movl    $2, -16(%ebp)
        jmp     L5
L6:
        movl    -16(%ebp), %eax
        movl    -416(%ebp,%eax,4), %eax
        testl   %eax, %eax
        jne     L7
        movl    -16(%ebp), %eax
        movl    %eax, (%esp)
        call    L_write$stub
        movl    -16(%ebp), %eax
        movl    %eax, -12(%ebp)
        jmp     L9
L10:
        movl    -12(%ebp), %eax
        movl    $1, -416(%ebp,%eax,4)
        movl    -16(%ebp), %edx
        leal    -12(%ebp), %eax
        addl    %edx, (%eax)
L9:
        cmpl    $100, -12(%ebp)
        jle     L10
L7:
        leal    -16(%ebp), %eax
        incl    (%eax)
L5:
        cmpl    $100, -16(%ebp)
        jle     L6
        leave
        ret
```

Figure 3: The Sieve of Eratosthenes in Intel x86 Assembly Language. This function was generated by the `gcc` compiler. The array and indices are all local variables.

- The *overflow flag* is set when the result, interpreted as signed, is out of range. For addition that means that the overflow flag is set when two operands having the same sign produce a result with a different sign. For subtraction it means that the operands have different signs and the minuend[1] and result also have different signs.

Combinations of these flags will give all possible comparisons. For example, with unsigned values, `a` < `b` is true when the carry flag is set after computing `a` − `b`. With signed values, `a` < `b` is true when the sign flag is different from the overflow flag. See the course document *Logic, Words, and Integers* for further specifications.

Look again at Figure 3 and notice the branches. The sequence `cmpl; jle` corresponds to CS52 Machine's `ble`.

5.4 Stack Frames

The top of the stack may move during the execution of a subprogram (to store intermediate results of a calculation, for example). Most real computers designated another register, called `fp`, for *frame pointer*, to retain the location of the current stack frame. The addresses of elements in a stack frame are computed using an offset from the frame pointer. The frame pointer is called the *base pointer* in some architectures.

Here is the sequence of operations that takes place when a subprogram is called:

1. The arguments are pushed onto the stack, one at a time, by the caller.
2. The call is made. Either the call instruction pushes the return address automatically or the callee pushes it from a designated register.
3. The value of `fp` is pushed by the callee.
4. The value of `sp` is copied into `fp`.
5. The top of stack is adjusted to make space for the local variables.

There are many variations on this theme. The order of operations may be slightly different, there may be a different division of labor between the caller and the callee, and other values may appear in the frame. The crucial features are that the caller's values for `fp`, `sp`, and `ic` are saved (in steps 2, 3, and 4) for restoration later.

As shown in Figure 4, the frame pointer is an address in the middle of the frame. In our convention, `fp` is the address of the first local variable. The return address is at address `fp` + 4 (assuming two-byte words). The subroutine has complete access to the arguments, as long as it knows how many there are.

Upon completion of the subprogram, the following steps are executed:

[1]The *minuend* is the top number in a subtraction problem. The *subtrahend* is subtracted from the minuend.

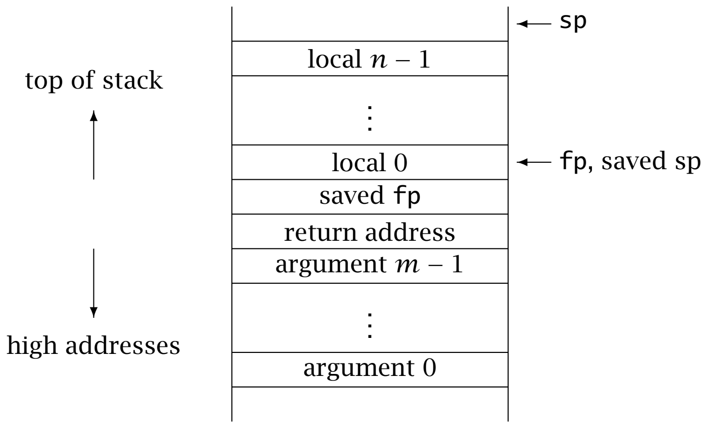


Figure 4: A typical stack frame for a subprogram call with m arguments and n local variables. As shown, the stack grows from higher addresses to lower ones.

1. The value of `fp` is copied to `sp` by the callee, effectively discarding the lo cal variables.
2. The saved value of `fp` is restored with a pop by the callee.
3. The return address is popped by the callee, and execution returns to the caller.
4. The caller increments `sp` to discard the arguments.

Again, variations are possible. However the details are carried out, the essen tial parts are to restore `fp` and `sp` to their values before the call and to set `ic` so that the calling program continues on its way.

If the subprogram is a function, it returns its value either in a register or on the stack. If a value is returned on the stack, space is allocated for it before the arguments are pushed, so that the returned value is on the top of the stack after caller has discarded the arguments.

Look, once again, at Figure 3 and notice the beginning and end. The Intel stack pointer is `%esp` and the frame pointer is `%ebp`. The first few lines match our description exactly. It is not necessary to push the return address because the Intel `call` instruction automatically pushes it. The exit code of the function is not as clear, because all the operations are packed into the complex instructions `leave` and textttret. It is a very good example of how a complex instruction set can package frequently-done operations.

5.5 Operating System Support

Just as an assembler makes programming more efficient, an operating system makes using a computer easier. In this section, we touch on a few services provided by an operating system and show how those features are supported by the hardware.

Loader One bit of "magic" in our CS52 Machine simulator is the *loader*, which reads an external program file and copies its contents into memory. In the CS52 application, the loader is an external program; on a real computer, the loader is itself a program. When a user starts a program, the operating system reserves space for it, the loader is invoked to copy the file into memory, and then the program is started. When the program is finished, the operating system is called to reclaim the memory. In a sense, a program is simply a subprogram of the loader.

One could conceivably write a loader for the CS52 Machine, but it would be awkward to translate a program into a sequence of signed integers and enter it through the CS52 Machine's only input channel.

Memory Management On a real computer, several programs run at the same time, each one having the illusion that it has the whole computer to itself. It is the operating system's job to parcel out blocks of memory and to keep the programs from interfering with one another. One aspect of the task is *virtual memory.* Each running program has its own virtual memory space, as if it occupied the whole computer. There is special hardware that translates virtual addresses into addresses in the actual memory. It is the operating system's responsibility to maintain the different translation tables in a way that each program has its own private region of memory.

A real computer has special registers and a collection of privileged commands that the operating system uses to manage memory. It would be virtually impossible to implement memory management on the CS52 Machine.

Multiprogramming Even though it appears that several programs are running on a real computer, only one can actually be executing at a given instant. (Or, in the case of multi-core and multi-processor computers, only a few can be executing at one time.) The illusion of many programs running simultaneously is accomplished through time-sharing. Each program gets a short amount of time on the processor, and then is replaced by the next program. The switch between programs happens many times a second, so that it appears that each program is executing continuously.

The operating system maintains a queue of programs that are ready to run, and it moves from one to the next sequentially. The "trigger" to shift from one program to another is a *timer tick.* The timer is an external device that sends a signal, an interrupt, to the processor at fixed intervals. Typically, a tick occurs a few hundred times a second.

The processor must support interrupts by having a few special registers for use by the operating system only. When an interrupt occurs, the processor replaces the user program's stack pointer with another register which serves as the operating system's stack pointer. A program in the operating system is then given control. It saves the *state* of the currently-executing program—including all the registers, flags, and virtual memory translation data. It then restores the state of another program and allows it to resume execution.

The processor has two states: user state in which ordinary programs execute and supervisor state in which the operating system executes. In supervisor state, the operating system has access to the special registers and instructions to manage memory and programs. Obviously, it would be very difficult to add interrupts and a supervisor state to the CS52 Machine.

Input and Output Real computers have many different channels to get data in and out of a computer. They include displays, keyboards, mice, tapes, disk drives, and network interfaces. *Memory mapping*, which we have used in a primitive way with `mem[0]`, is the most common method for communicating with these devices.

The idea is that each device is assigned a set of addresses, and the computer communicates with the device by loading and storing to those addresses. Most programmers will never actually use those addresses, because there is a set of routines, called a driver, for each device that hides the low-level details. High level programming languages provide a collection of facilities for communicating with the input and output system.

A computer does not know when input has arrived, so a device (a network interface, for example) sends an interrupt when it needs attention. Just as in the case of a timer tick, the operating system takes over, transfers data from the device to memory, and informs the relevant program that it has input. The "relevant program" may actually be another part of the operating system. For example, when a new electronic mail message arrives, the operating system passes it to the system-wide mail manager, which in turn deposits it in the recipient's mailbox.

A The CS52 Machine Simulator

We will work with a Java application that assembles and runs CS52 Machine programs. The code for the application is contained in a Java jar file, found at `/common/cs/cs052/cs52-machine/cs52-machine.jar`. In the laboratories, you may start it by double clicking on the jar file or by executing a command in a terminal window. See Appendix A.3 for the command line options. You may also download the jar file from the course Resources page to your own computer and execute it there.

A.1 File Names

The application uses the following extensions for files:

`.a52`	a source program in assembly language
`.m52`	the object program, executable by the CS52 Machine

The program expects to be able to write to the directory in which it finds the files. If you are running example programs, first copy them to a location in your own home directory.

A.2 The Visual Application

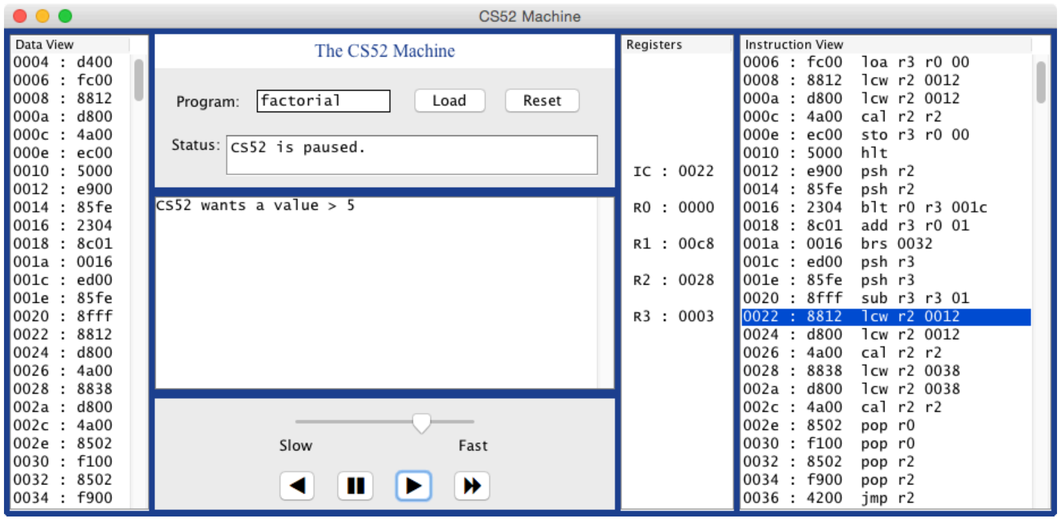
On most platforms, you can start the application by clicking on the jar file's icon.

The exact appearance of the window will differ according to platform, but it will be organized as shown in Figure 5. There are two views of memory: On the far left is the Data View in which the memory values are interpreted as data, and on the far right is the Instruction View in which the memory values are interpreted as instructions. The instructions are translated into assembly language, with the default view being the "Assembly View" showing the instructions that the programmer actually wrote. The alternative is the "Machine View" which shows the actual native machine instructions that the CS52 Machine executes.

To the left of the Instruction View is a panel that shows the registers and their current values.

The large panel in the center is where the input and output takes place. Above and below the input/output window are status messages and controls.

Controls on the main window The buttons have been designed to work in an intuitive manner. The Load button opens a file selector window from which you can



```
Data View
0004 : d400
0006 : fc00
0008 : 8812
000a : d800
000c : 4a00
000e : ec00
0010 : 5000
0012 : e900
0014 : 85fe
0016 : 2304
0018 : 8c01
001a : 0016
001c : ed00
001e : 85fe
0020 : 8fff
0022 : 8812
0024 : d800
0026 : 4a00
0028 : 8838
002a : d800
002c : 4a00
002e : 8502
0030 : f100
0032 : 8502
0034 : f900
```

```
Program: factorial
Status: CS52 is paused.
CS52 wants a value > 5
```

```
Registers
IC : 0022
R0 : 0000
R1 : 00c8
R2 : 0028
R3 : 0003
```

```
Instruction View
0006 : fc00  loa r3 r0 00
0008 : 8812  lcw r2 0012
000a : d800  lcw r2 0012
000c : 4a00  cal r2 r2
000e : ec00  sto r3 r0 00
0010 : 5000  hlt
0012 : e900  psh r2
0014 : 85fe  psh r2
0016 : 2304  blt r0 r3 001c
0018 : 8c01  add r3 r0 01
001a : 0016  brs 0032
001c : ed00  psh r3
001e : 85fe  psh r3
0020 : 8fff  sub r3 r3 01
0022 : 8812  lcw r2 0012
0024 : d800  lcw r2 0012
0026 : 4a00  cal r2 r2
0028 : 8838  lcw r2 0038
002a : d800  lcw r2 0038
002c : 4a00  cal r2 r2
002e : 8502  pop r0
0030 : f100  pop r0
0032 : 8502  pop r2
0034 : f900  pop r2
0036 : 4200  jmp r2
```

Figure 5: The appearance of the CS52 Machine simulator.

choose the file to run. The Reset button takes the machine back to the state at which the program was originally loaded.

The four buttons at the bottom control the execution of the program.

◀ The Back button causes the machine to go one step backwards. It is useful when you want to study carefully a few steps of the CS52 Machine.
The CS52 Machine remembers on a limited number of steps, so there may be cases when it is not possible to go backwards. In those cases, the Back button will do nothing. The button will also be ignored when there is no execution history or the machine is in a unrecoverable error state.

❚❚ The Stop button stops execution when the CS52 Machine is running. In other cases it does nothing.

▶ The Step button causes the CS52 Machine to execute one instruction. It does nothing when there is no “next” instruction.

⏩ The Run button lets the CS52 Machine run to completion—or until one of the other buttons is pressed.

If the machine has halted normally, pressing the Step or Run button will reset the machine and start the program anew. If the machine is in an error state, it must be reset explicitly by pressing the Reset button.

The slider above the four buttons controls the speed at which the CS52 Machine runs.

The top-level menus The menus appear at the top of the screen on a Macintosh and on top of the simulator window on other platforms. Under File there are clones of the Load and Reset buttons, a Clear button to erase all programs and data from memory, and a Force Re-assembly button.

The Load and Reset buttons are programmed to call the assembler aggressively. A program will be assembled prior to loading if the source file exists and the object file either does not exist or it is older than the source file. An object file will be reloaded if it is newer than the previously loaded file. In rare cases, the assembler will not activate when it should. Use the Force Re-assembly selection in those situations.

Under the top-level View menu, you will find controls to switch between "Assembly View" and "Machine View" in the Instruction memory window.

Under Memory there are selections to set the memory size. Notice that the memory is cleared when its size is changed.

Stopping the simulator Exit the application by closing the window, selecting Quit from the top-level File menu, or by pressing the appropriate key-combination for your platform (usually command-Q or control-Q).

Breakpoints Perhaps the only non-obvious feature of the simulator is the facility for setting breakpoints. Double click on a line in the instruction memory window. The colon will change to a B indicating that the breakpoint is set. The simulator will pause, as if you had pressed the Stop button, when the `ic` reaches the address of the breakpoint. Use the Step or Run button to resume execution. Double click in the instruction window again to remove the breakpoint.

A.3 Command Line Invocation

By invoking the program from the command line, one can obtain a wider range of behaviors. The basic command is

```
java -jar <path-to-jar-file> <arguments>
```

The path is the location of the file `cs52-machine.jar`. Examples of paths are `~/cs052/cs52-machine.jar` and `/common/cs/cs052/cs52-machine/cs52-machine.jar`. The arguments cause the the program to fulfill various functions.

Assembler The arguments for the stand-alone assembler are

```
-a [ sourceFile ] [ -f ]
```

When the `sourceFile` is specified, it must have the extension `.a52`, and the output goes to the file with the same name and extension `.m52`. When the `sourceFile` is not specified, the input is taken from the standard input, and the output goes to the standard output.

The `-f` flag indicates "filtered input" in which only lines between the tokens `BEGIN` and `END` are passed to the assembler. This is a device that allows the assembler to take input from a program that may spew other output.

Executor To run a program, give a version of the following arguments:

```
-r [ objectFile ] [ -m memorySize ] [ -u inputScript ]
```

When the `objectFile` is specified, it must have the extension `.m52`. If the `objectFile` is not specified, the program is expected to come from the standard input.

If a `memorySize` is specified, it must be a decimal or hexadecimal integer specifying the number of bytes, or else a decimal or hex integer followed by the letter `K`, specifying the number of kilobytes. The smallest permissible memory size is 64 bytes. The largest is 2^{16} or $65,536$ bytes, or 64K.

If an input script is specified, it must exist as a text file with one decimal integer to a line. It is not possible to take both program and user response from the standard input.

Pipe The pipe mode is like the executor, except that the input is a source file that is passed to the assembler before execution. No object file is created.

```
-p [ sourceFile ] [ -m memorySize ] [ -u inputScript ] [ -f ]
```

As above, the program comes from a specified file or the standard input. The other flags have the same meaning as before.

Visual Application One can also start the visual application from the command line, with the advantage of optionally specifying a non-standard memory size.

```
[ -m memorySize ]
```

A.4 Error Messages

Here is a list of error messages generated by the application. The majority of the messages correspond to errors found by the assembler, before the program runs. Most of the messages are self-explanatory.

Source file not found.

Source file could not be opened.

Object file could not be created.

Illegal character . . . The alphabet of the assembly language consists only of letters, plus, minus, underscore and semicolon.

Register name used as argument at line . . .

Extra arguments for . . . at line . . .

Incorrect number of registers for . . . at line . . .

Argument needed for . . . at line . . .

Argument must be numeric for . . . at line . . .

Single string argument is required by . . . at line . . .

Auxiliary code must be a constant for . . . at line . . .

Illegal opcode . . . at line . . . The assembler found a string that is not one of the accepted opcodes. Check your spelling.

Constant out of range at line . . . The numeric argument to instructions like `lcw` must be 16-bit quantities.

Argument out of range for . . . at line . . . The numeric argument to instructions like `add` and `loa` must fit into a single byte.

This error can occur in branch instructions as well. A branch, when taken, adds an eight-bit signed quantity to the IC. The error is telling you that the target is too far away from the branch instruction. See page 26.

Illegal label at line . . . A label must begin in the first column and start with a letter.

Data in executable block at line . . . A `dat` directive may not appear between instructions; see page 23.

Bad argument to dat directive at line . . .

No executable code in source file.

No library file specified at line . . .

Library file . . . at line . . . not found.

Library depth exceeded. Library files included by the `inc` instruction may be nested up to a depth of four.

Duplicate label: You have declared the same label twice in your program.

Label not found: You have used a label in an instruction but have not declared it. The label does not refer to a location in the program.

Another, smaller, class of errors are those encountered at runtime.

Unimplemented instruction at address . . .

Illegal instruction format at address . . .

IC is odd. This error is usually a sign of a runaway program, often due to corrupted memory.

Memory access out of bounds at address . . .

Input is not a number or out of range. The user has typed input that is not a number or is out of range for a sixteen-bit signed quantity.

Input is unavailable, possible overuse of stdin. In the command line operation of the simulator, it is not possible to take both a program and user responses from the standard input.

Incorrect memory size. The memory size specified on the command line was out of range.

Error in machine code file. The CS52 Machine encountered an error when loading a program.

Program too large.

File not found

Division by zero at address . . .

There are a few error messages that describe internal inconsistencies in either the assembler or the runtime system. It is unlikely that you will see them, but please report them if you do. They all begin with the same phrase.

Twilight zone: . . .

A.5 Legal Stuff

B Binary Representation of Integers

The basic unit of data on the CS52 Machine is the 16-bit word, consisting of two consecutive bytes. A word can represent an unsigned address into memory, a signed integer data value, or a machine instruction.

If $b_{15}b_{14}\ldots b_2b_1b_0$ is the sequence of bits in a word, we interpret it as the *unsigned integer* having the value

$$b_{15}2^{15} + b_{14}2^{14} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0.$$

The unsigned values range from 0 through $2^{16} - 1$. Memory addresses are unsigned integers.

The same bit pattern can be interpreted as the *signed integer* whose value is

$$-b_{15}2^{15} + b_{14}2^{14} + \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0.$$

The only difference is the sign on the leading term. Signed values range from -2^{15} through $2^{15} - 1$. The high-order bit b_{15} is the *sign bit;* it is 1 when the value is negative. Notice that "signed" does not mean "negative," rather it means "potentially negative."

A signed value can be negated by complementing all the bits (switching them between 0 and 1) and then adding 1. When negated, the word

    00000000 00000010

becomes

    11111111 11111101 + 00000000 00000001 = 11111111 11111110.

We have just transformed 2 into -2 in binary.

One can also view single bytes as unsigned or signed. The unsigned values range from 0 through $2^8 - 1$. The signed values range from -2^7 through $2^7 - 1$. An unsigned byte can be converted to an unsigned word by adding 8 zeroes on the left. Interestingly, a signed byte can be converted to a signed word by padding it on the left with 8 copies of the sign bit. This process is called *sign extension* and is used frequently in the CS52 Machine.

Addition and subtraction can be carried out on unsigned or signed values; the algorithm is exactly the same. In both cases, we simply discard the carry-out from the high-order bit. It is possible to compute a result that is not expressible in 16 bits. Whether or not a result is erroneous depends on the interpretation we impose. For example,

    01000000 00000000 + 01000000 00000000 = 10000000 00000000, which is correct in the unsigned case. In the signed case, it is incorrect because we add two positive numbers and get a negative result. On the other hand,

11000000 00000000 + 01000000 00000000 = 00000000 00000000,

is incorrect in the unsigned case because we add two positive numbers and get zero. It is correct in the signed case, because adding $-2^{15} + 2^{14}$ and 2^{14} really does give a result of zero.

It is cumbersome to write out sequences of 16 bits, and we often resort to *hexadecimal notation.* We divide the bits into 4-bit blocks and take the hexadecimal (base-16) value of each block. We use the ordinary digits `0` through `9` for their usual values and then add the letters `a` through `f` for the values 10 through 15. The prefix `0x` is often used to distinguish a hexadecimal value from a decimal one. The hexadecimal expression `0x3c04` corresponds to the bit pattern

0011 1100 0000 0100.

The CS52 Machine supports operations besides addition and subtraction. It can *shift* bits in a word left or right. When it shifts a word three bits to the left, it discards three bits on the left and adds three zeroes on the right. Shifting one bit to the left is the equivalent of multiplying by 2. Shifting three bits left is the equivalent of multiplying by 8.

Right shifts are the equivalent of dividing by powers of two. the *logical* right shift adds zero bits on the left. It is the equivalent of dividing an unsigned number by a power of two. The *arightmetic* right shift adds copies of the sign bit of the original word. In that way, it preserves the sign. The arithmetic right shift is the equivalent of dividing a signed number by a power of two.

We often talk about logical and arithmetic *left* shifts as well, but they are the same operation.

There are three *boolean* operations on bits:

- **and** gives the minimum of the two bits. The result is 1 only when both operands are 1.
- **or** gives the maximum of the two bits. The result is 1 except when both operands are 0.
- **xor** is the *exclusive-or* operation. The result is 0 when the two operands are the same, and 1 when they differ.

The CS52 Machine is capable of *bitwise* boolean operations. The boolean operation is applied in parallel to the bits in a word. The bitwise-and of a word with

00000000 11111111

sets the high-order byte to zero and leaves the low-order byte unchanged. The bitwise-or of a word with that value leaves the high-order byte unchanged and sets the low-order byte to all 1's. The bitwise-xor of a word with that value leaves the high-order byte unchanged and complements all the bits in the low-order byte.

C Sample Programs

Here are the listings of a few programs, including those developed in the text. Others examples can be found online in the course resources.

C.1 Subtract

```
;
; subtract.a52
;
; A simple CS52 Machine program that subtracts
; two numbers. A first example.
;
; Rett Bull
; Pomona College
; Written for CS41B, July 17, 2009
; Adapted to the CS52 Machine, February 12, 2016
;
        loa r2 r0           ; get first value
        loa r3 r0           ; get second value
        sub r2 r2 r3        ; subtract them
        sto r2 r0           ; print result
        hlt                 ; quit
```

C.2 Sign

See Exercise 4 in Appendix E for a branch-free way to compute the same value.

```
;
; sign.a52
;
; A CS52 Machine program that illustrates conditional
; execution by determining the sign (-1, 0, or +1) of
; an integer.
;
; Rett Bull
; Pomona College
; Written for CS41B, August 7, 2009
; Adapted to the CS52 Machine, February 12, 2016
;
;
;       if (a < 0)
;          write(-1);
;       else if (0 < a)
;          write(1);
;       else
;          write(0);
;
;
        loa r2 r0           ; get a value for a

        ble r0 r2 elseif    ; first comparison
        add r3 r0 -1        ; place result -1 in r3
        brs endif           ; skip other clauses

elseif
        ble r2 r0 else      ; second comparison
        add r3 r0 1         ; place result +1 in r3
        brs endif           ; skip other clauses

else
        mov r3 r0           ; place result 0 in r3

endif
        sto r3 r0           ; write the value in r3
        hlt                 ; halt
```

C.3 Sieve of Eratosthenes

```
;
; sieve.a52
;
; A CS52 Machine program that executes the Sieve of Eratosthenes
; to illustrate arrays, conditional execution, and nested
; loops. We use all three general registers; there is no
; stack.
;
; Rett Bull
; Pomona College
; Originally written for CS41A in the Fall of 1989
; Adapted to CS41B on July 28, 2009
; Modified on August 7, 2009
; Adapted to the CS52 Machine on February 12, 2016
;
;
;       int i;
;       for (i = 2; i < 100; i++)
;           num[i] = i;   // any non-zero value will do; i is convenient
;       for (i = 2; i < 100; i++)
;           if (num[i] != 0) {
;              write(i);
;              j = i;
;              while (j < 100) {
;                  num[j] = 0;
;                  j += i;
;              }
;           }
;
;
;
;



;
; Data area: We set aside a location to store i and a sequence
; of words for the array num.
;
i       dat 2               ; i, at address 2, the first "real" memory cell
num     dat  200            ; an array indexed 0 through 99;
```

```
;
; First loop:
;
; We use the fact that num is a small address, and use it as an
; offset to the sto instruction. If num were not a small address,
; we would have to load it into a register using a stored offset.
;
        add r3 r0 2          ; i = 2;
        add r1 r0 100        ; limit = 100;

        ble r1 r3 enda       ; check if done
loopa
        sal r2 r3 1          ; double to get word offset
        sto r3 r2 num        ; num[i] := i
        add r3 r3 1          ; i++;
        blt r3 r1 loopa      ; go back for another round
enda
```

```
;
; Second (nested) loops: In the inner loop, we have more
; quantities than registers. We have to store i temporarily
; and reload the constants when they are needed.
;
; The code is "semi-optimized" in the sense that we have
; eliminated redundant labels and branches. For clarity,
; we retain the instructions that restore the constant
; value 100 into r1; only one of the three instances of
; that instruction is necessary.
;
                           ;;; set up outer loop
        add r3 r0 2        ; i = 2
        add r1 r0 100      ; limit = 100
        ble r1 r3 endb     ; conditional branch to endb
loopb
                           ;;; body of outer loop
        sal r2 r3 1        ; double to get word offset
        loa r2 r2 num      ; load num[i]
        beq r2 r0 endif    ; branch to endif when num[i] == 0
        sto r3 r0          ; write(i)

                           ;;; set up inner loop
        sto r3 r0 i        ; store i temporarily
                           ;     we now use r3 for j
        ble r1 r3 endc     ; conditional branch to endc
loopc
                           ;;; body of inner loop
        sal r2 r3 1        ; get address of num[j]
        sto r0 r2 num      ; make num[j] zero
        loa r2 r0 i        ; recover i
        add r3 r3 r2       ; increment j
        blt r3 r1 loopc    ; conditional branch to loopc
endc
                           ;;; clean up after inner loop
        loa r3 r0 i        ; restore i into r3
endif
                           ;;; finish the outer loop
        add r3 r3 1        ; increment i
        blt r3 r1 loopb    ; conditional branch to loopb
endb
        hlt                ; we're done!
```

C.4 Factorial

```
;
; factorial.a52
;
; A CS52 Machine program that computes factorials. It
; illustrates subprogram calls and the use of the
; stack for local variables.
;
; Rett Bull
; Pomona College
; CS41B version, August 7, 2009
; CS52 Machine version, June 22, 2016
;
;       int factorial(int x) {
;           if (x <= 0)
;              return 1;
;           else
;              return mult(f(x-1), x);
;       }
;
; We use the CS52 Machine's multiplication library and
; adopt the CS52 Machine conventions for the use of the
; stack and registers.
;



;
; main routine: establish the stack, then read
; and write.
;
        lcw r1 stack        ; set up stack
        loa r3 r0           ; get variable

        lcw r2 fact         ; call fact
        cal r2 r2           ;

        sto r3 r0           ; write result,
        hlt                 ;    and halt
```

```
;
; fact subprogram: One argument, in r3.
;
fact
        psh r2              ; save return address
        blt r0 r3 recursion ;

        add r3 r0 1         ; base case; result is 1
        brs done            ;

recursion
        psh r3              ; save argument
        sub r3 r3 1         ; decrement argument

        lcw r2 fact         ; make recursive call
        cal r2 r2           ;

        lcw r2 mlmul        ; call the multiplication routine
        cal r2 r2           ;     one argument is in r3
                            ;     and the other is on
                            ;     the stack

        pop r0              ; discard saved argument
                            ; result is already in r3

done
        pop r2              ; restore return address
        jmp r2              ; return

;
; include the multiplication library
;
        inc mullib

;
; stack area: 50 words
;
        dat 100
stack
```

C.5 Multiplication Library

A library is not a program, rather it is included with the directive `inc` in another program. Here is an example library that provides a multiplication function. The sample programs include the libraries `divrlib.a52` and `divilib.a52`, which provide recursive and iterative division functions.

```
;
; mullib.a52
;
; A multiplication library for the CS52 Machine.
;
; Rett Bull
; Pomona College
; February 3, 2016
;
;
; SEE ALSO: divrlib.a52 and divilib.a52 for recursive
; and iterative division routines.
;
; USAGE: Word values are interpreted as signed integers.
;
; One argument (call it X) is on the stack and another
; argument (Y) is in r3.
;
; The function mlmul leaves X*Y in r3, and X remains
; unchanged on the stack. More precisely, the routine
; leaves the low-order word of X*Y in r3.
;
;
; LABEL CONVENTION: The entry point is mlmul. To avoid
; conflicts with other programs, all local labels start
; with libm.
;
;
```

```
; MULTIPLICATION: Essentially, we are doing elementary
; school multiplication in binary. It requires that
; the multiplier Y is non-negative. We will end up with
; the low 16 bits of X*Y.
;
;    mlmul(X,Y) {
;       if (Y < 0) {
;          X = -X;
;          Y = -Y;
;       }
;       P = 0;
;       while (Y != 0) {
;          if (Y & 1 == 1)
;             P += X;
;          X = X << 1;
;          Y = Y >> 1;
;       }
;       return P;
;    }
;
;    X is always in r2
;    Y and P share r3
;    Temporary storage is allocated on the stack, with Y
;    on top and P below it.
;
;
```

```
mlmul                       ;;; prolog
        psh r2              ; push the return address
        psh r0              ; push initial value of P
        loa r2 r1 6         ; fetch initial value of X

                            ;;; make sure the signs are correct
        bge r3 r0 libmpos   ; test the sign of Y
        neg r2 r2           ; change sign of X
        neg r3 r3           ; change sign of Y

libmpos                     ;;; get ready for the loop
        beq r3 r0 libmdone  ; if Y = 0, we are finished
        psh r3              ; push Y

libmloop                    ;;; the body of the loop
        and r3 r3 1         ; test low bit of Y,
        beq r3 r0 libmnadd  ;   and skip adding if it is 0
        loa r3 r1 4         ; fetch P
        add r3 r3 r2        ; add X to P
        sto r3 r1 4         ; save P
libmnadd
        loa r3 r1 2         ; recover Y
        slr r3 r3 1         ; shift Y right (logical shift, why?)
        sto r3 r1 2         ; store Y back again
        sal r2 r2 1         ; shift X left
        bne r3 r0 libmloop  ; go back for more if Y != 0

        pop r0              ; discard temporary Y

libmdone                    ;;; postscript
        pop r3              ; get the result P
        pop r2              ; recover the return address
        jmp r2              ; return to the caller
```

D Historical and Contextual Notes

D.1 History

The tradition of the CS52 Machine goes back to the 1980's when a rudimentary computer and the assignments surrounding it were inspired by Professor Richard Lorentz, then at Harvey Mudd College. In its original incarnation, the computer had only one data register and used decimal numbers.

The CS41A machine was designed in 1989 to look a little more like a real computer and to illustrate more concepts. In particular, it introduced an explicit instruction encoding. The original intention was to have a sequence of machines, CS41B, CS41C, and so on, of increasing complexity. It quickly became evident, however, that the CS41A architecture was not an adequate platform for expansion. Stack operations were to be an enhancement of the CS41B Machine, but maintaining a stack with only one register was too cumbersome. It became clear that CS41B would have to be an entirely new machine and could not be an extension of CS41A.

The CS41A machine stabilized in 1989 and was not changed thereafter. It was used until 1997, when it was abandoned in favor of another simulator, called ISC, that offered more registers and more flexibility.

In the spring of 2009, alumni from the late 1990's returned to campus and reported that they had been amusing themselves rewriting CS41A assignments from a decade earlier. Their report prompted another look at CS41A and its window-based simulator. The simulator was a real advantage, and in the summer of 2009 the CS41B Machine was designed and implemented. As expected, it has multiple data registers and the facilities for calling subprograms. The simulator was written and the examples and assignments were translated into the new instruction set.

The CS52 Machine was written in the spring of 2016. It is an incremental improvement over the CS41B machine, based on years of experience with CS41B. The new machine retains the same hardware architecture of its predecessor, but its native instruction set is new and has additional facilities for bitwise logical operations and shifts. The assembly language is richer and easier to use. The simulator has added features that make it easier to observe a program's execution.

D.2 Design Decisions

The CS52 Machine provides a simple yet realistic model of the register-transfer level of a computer. It introduces students to assembly-language programming, gives them an general idea of what a compiler actually does, and shows them how recursion is implemented.

The basic architecture follows the original RISC design: uniform length instructions, three-register operations, and straightforward encoding of instructions into bits. The 16-bit word is large enough to accommodate a reasonable instruction set and suffices for moderately sized programs.

Four data registers (constant zero, stack pointer, and two data registers) are just enough to support stack-based execution. It would have been possible to design an instruction set with eight registers, but that would have limited the types of operations and, more importantly, would have made programming too easy. With only two general-purpose registers, students have to confront significant resource constraints.

The CS52 Machine has only one datatype, the 16-bit signed integer. It would have been possible to add, say, a single byte type, but it would have been difficult to squeeze in an adequate number of instructions to support the type. The same can be said, even more strongly, about characters, unsigned numbers, and multi-word structures. Only simple numerical applications are possible, but those are sufficient to illustrate assembly programs, translation, and recursion.

I was tempted to include some form of operating system support, but in the end I decided against it. It would take the machine too far away from its intended purpose. Certainly, adding interrupts would have made the machine much too complicated. On the other hand, it would be feasible to add a few system services—the `cal` instruction has some extra bits that could be used to encode them, but the extra overhead of maintaining call vectors and inserting systems code would only make it harder to understand the operation of simple programs. As a result, the assembler, the loader, and the input-output system all operate externally to the CS52 machine.

The use of memory address 0 for input and output is perhaps over-simplified, but it does illustrate memory-mapped I/O. It is not hard to generalize from that trivial instance how keyboards, network interfaces, mice, and even video displays are connected to a processor and memory.

The CS52 Machine's assembly language is rich enough to be natural, but not so rich as to remove the programmer very far from the registers and memory. Instructions like `psh`, `pop`, and `mov` are natural and convenient. When students learn that the natural operations are actually built on the more primitive machine language, they come to appreciate what assemblers and compilers do for them.

I made little attempt to explain how the CS52 Machine operates below the register-transfer level, but it is not difficult to imagine how it might be done. The bit-level encoding of instructions was chosen to make it possible to construct a CS52 Machine out of gates. Perhaps some enterprising student will use a circuit simulator to do just that.

D.3 Comparison with the CS41B Machine

This section, which is slated to disappear in 2017, is intended for programmers with experience with the older CS41B Machine; others should ignore it.

Filenames The file extensions are now `.a52` and `.m52`.

- ► To adapt a CS41B Machine program, begin by making a copy with the extension `.a52`.

The `sto` Instruction The order of registers passed to the `sto` instruction has changed. The instruction

```
sto rX rY arg
```

now computes the address `rY` + `arg` and places the value in `rX` in memory at that address.

The change was made to bring symmetry to the `sto` and `loa` commands. In the CS52 Machine, the first register contains the value that is traveling between a register and memory. The address is computed from the value in the second register and the optional sign-extended argument. The circuitry on an actual computer is simpler when the address is always computed from registers in the same position in the binary instruction.

- ► To adapt a CS41B Machine program, reverse the order of registers in every `sto` command.

The `sbc` Instruction The `sbc` instruction has been eliminated. Programmers may use `sub` instead.

- ► To adapt a CS41B Machine program, replace each `sbc` instruction with `sub`.

The `pau` Instruction The `pau` instruction has been eliminated. It has been replaced with a breakpoint facility.

- ► To adapt a CS41B Machine program, take out the `pau` instructions.

The `end` Directive The `end` directive has been eliminated. There was no real need for it.

- ► To adapt a CS41B Machine program, take out the `end` directive.

The `lcl` Instruction The `lcl` command ("load constant byte low") has been eliminated. It was used only to construct a constant word, a process that is now done in a different way.

- ► To adapt a CS41B Machine program, do nothing. It is highly unlikely that any human programmer would have used the `lcl` command.

Idiom While the `adc` instruction is still present in the CS52 Machine's assembly language, its use by programmers is discouraged. One can write

```
add rX rY arg
```

as well as

```
add rX rY rX
```

so that `adc` can be replaced by `add`. It is more natural to use the same mnemonic for both operations, even though they are different at the machine level. The assembler distinguishes between the two cases.

E Practice Exercises

Here are some exercises to hone your skills with the CS52 Machine and its instruction set. Variations of some of the exercises may appear on assignments.

1. Write a sequence of CS52 Machine instructions that interchanges the values in two registers without changing the data in any other register or memory location. Hint: Consider the following sequence of assignment statements.

```
x = x + y;
y = x - y;
x = x - y;
```

2. Write a one-instruction infinite loop in the CS52 Machine assembly language.

3. a. It is sometimes useful to separate the two bytes of a word. Write a sequence of CS52 Machine instructions that replaces the word in `r3` with its low-order byte and leaves the other registers unchanged. (Hint: Think about shifts.)

b. Do the same for the high-order byte.

4. Explain why the following CS52 Machine program has the same behavior as the sign program given in Appendix C.2.

```
loa r2 r0
neg r3 r2
sar r2 r2 15
sar r3 r3 15
sub r3 r2 r3
sto r3 r0
hlt
```

5. There is one low-level detail about the CS52 Machine that has not been explicitly specified. It is the question of “endian-ness.” We know that a word at address `adr` occupies the two bytes at addresses `adr` and `adr+1`. If the value of the word is `0x0123`, then one byte has value `0x01` and the other has value `0x23`. But which is which?

A computer's representation of integers is *little endian* if the less significant byte, `0x23` in our example, is at the lower address. It is *big endian* if the less significant byte is at the higher address.

a. Write a short program to determine the "endian-ness" of the CS52 Machine.

b. Why do you have to write a program to discover the "endian-ness?" What is wrong with just looking at the values in the Data View panel?

c. Do some research to discover which famous literary work originated the terms "big endian" and "little endian."

6. In some computer architectures, engineers have adopted the convention that the words "less" and "greater" refer to signed integer comparisons, and the words "below" and "above" refer to unsigned integer comparisons.

a. List the conditions, in terms of signed comparisons of values in registers, under which "`rX` is below or equal to `rY`" is true.

b. Suppose that you wanted to add a new synthetic instruction `bbe`, branch on below or equal, to the CS52 Machine's assembly language. Give a sequence of native instructions that corresponds to

```
    bbe rX rY target
```

Assume that `target` is a label. You may use that label in the instruction sequence, but you may not introduce new labels. (However, it may be helpful to write a preliminary version using labels.)

7. When multiplying two 16-bit words, we may need as many as 32 bits to express the product. The function in the multiplication library of Appendix C.5 returns only the 16 low-order bits of the product. Write a new multiplication function that returns the full product in two words.

You may choose to implement either signed or unsigned multiplication. Make a choice and stick with it.

8. This exercise explores an interesting boundary case of the multiplication library from Appendix C.5. It illustrates the care one must take when writing low-level routines.

Near the end of the code for `mlmul`, we see the line

```
            slr r3 r3 1      ; shift Y right (logical shift, why?)
```

a. When applied to non-negative signed values, the operators `sar` and `slr` do the same thing. For what value of `Y` is it necessary to use the logical shift? What would happen if we used the arithmetic shift instead?

b. Explain why the routine gives the correct answer for the 16 low-order bits in all cases.

F Assembly Language Quick Reference

Here is a listing of all the CS52 Machine assembly language commands, along with their possible arguments. See Section 3 for details.

Legend:

`R` register	`B` signed byte offset	`F` string
`S` signed byte	`W` full word	`[ ]` optional
`U` unsigned byte	`A` 6-bit constant	

Arithmetic, bitwise logical, and shift instructions

`add`, `sub`, `and`, `orr`, `xor` } `RRR` or `RRS`	`mov`, `neg`, `not` } `RR`	`sll`, `slr`, `sal`, `sar` } `RRR` or `RRS`

Memory and data-moving instructions

`sto`, `loa` } `RR[S]`	`psh`, `pop` } `R`	`lcw RW`

Control instructions

`beq`, `bne`, `blt`, `bge`, `bgt`, `ble` } `RRB`	`brs B`	`nop`
	`cal RR`	`hlt [A]`
	`jmp R`	

Directives

`dat W`	`inc F`

Machine instructions, unlikely to be used by human programmers

`ari RRRA`	`orc RRS`	
`adc RRS`	`xoc RRS`	`lch RU`
`anc RRS`	`sft RRS`	