Computer Science 52

The CS41B Machine

Spring Semester, 2015

Contents

The CS41B 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 how recursion is implemented at the machine level.

We begin in Section 1 with a description of the machine and its operation. Then in Section 2, we describe the assembly language and see how programs are run. In Sections 3, we look at how the instructions are encoded as binary words, and in Section 4 we compare the CS41B Machine with real computers. The appendices contain documentation for the application that simulates our model, some sample programs, exercises, and a quick reference sheet for the assembly language.

1 The Machine

The fundamental data type of the CS41B Machine is the 16-bit word, two bytes. As unsigned integers, these words take on values between 0 and $2^{16} - 1$.

The CS41B 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.

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. This scheme is used even in the unusual case in which `addr` is odd. 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 CS41B Machine.

Memory location 0 has a special use in the CS41B 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 CS41B Machine. 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 CS41B Machine itself does not reserve the data registers for any particular purpose. Later, when we encounter subprograms, we will adopt conventions for the use of the registers.

1.3 Instructions

An *instruction* to the CS41B Machine is a tiny step of a computation; it may change a value in a register or at a memory location.

From the CS41B 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 CS41B Assembly Language. Each instruction consists of a three-letter abbreviation followed by up to three 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 is being changed. Similarly,

```
loa r3 r1
```

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

abbreviation	arguments	action

Register Instructions

`mov`	RR-	`dest` = `src0`
`neg`	RR-	`dest` = −`src0`
`add`	RRR	`dest` = `src0` + `src1`
`sub`	RRR	`dest` = `src0` − `src1`
`adc`	RRS	`dest` = `src0` + `arg`
`sbc`	RRS	`dest` = `src0` − `arg`
`lcw`	R-W	`dest` = `arg`
`lcl`	R-U	`lowbyte(dest)` = `arg`
`lch`	R-U	`highbyte(dest)` = `arg`

Table 1: The CS41B Assembly Language instructions that affect data registers. The first argument is always a register. The second one, if it appears, is also a register. If there is an argument in the third position, it can be a register, a signed byte, an unsigned byte, or a full word.

The arguments in an instruction can be registers, numerical values, or labels referring to locations in the program. In specifying arguments, we use `dest` to refer to the (leftmost) destination register, and we use `src0` and `src1` to refer to the other two register arguments. Only one of the three 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 that replaces `src1`.

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.

Table 1 through Table 4 list all the CS41B Assembly Language instructions. That information also appears in a handy one-page summary in Appendix E at the end of this document.

Table 1 contains the instructions that operate on data values in registers. The register instructions carry out arithmetic (or other) operations and place the result in the destination register.

Table 2 contains the instructions that interact with memory. The instructions `sto` and `loa` transfer data between a register and a memory location. We use `sto` to store a value from a register into memory and `loa` to load a value from memory into a register. There is an apparent asymmetry in the way that `sto` and `loa` are

abbreviation	arguments	action

Memory Instructions

`sto`	`RR[S]`	`mem[dest + arg] = src0`
`loa`	`RR[S]`	`dest = mem[src0 + arg]`
`psh`	`R--`	push the value in `dest`
`pop`	`R--`	pop the top of stack into `dest`

Table 2: The CS41B Assembly Language instructions that affect memory. The third argument, a signed byte, to the `sto` and `loa` instructions is optional.

written; it may be confusing at first. Just remember that the destination—whether it is a register or a memory location—always corresponds to the register on the left.

The byte argument to `sto` and `loa` is optional; it is assumed to be zero if it is not present.

Do not worry about the details of the stack instructions `psh` and `pop` for now; we will explore them later.

Table 3 contains the instructions that control the sequence of instructions being executed. The instructions `jmp` and `cal` are used to invoke, and return from, subprograms. (Subprograms are also called functions or methods or procedures.)

Table 4 contains *branch* instructions that adjust the instruction counter by the amount specified in the signed argument. They are used in conditional execution (if-then-else statements) and loops. For example, `brs` with a negative argument can be used to go back to the top of a loop for another iteration.

All of the branch instructions except `brs` are *conditional* branch instructions which take their branch if some relationship between `dest` and `src0` is satisfied. If the relationship is not satisfied, control simply proceeds to the next instruction in the sequence.

A pathological example is the instruction `blt r0 r1 2`. It "branches" to the next instruction regardless of the relationship between the values in `r0` and `r1`. Another pathology is `brs 0`; it is the world's shortest infinite loop.

The instruction `pau` exists for convenience in debugging programs and is not logically part of the CS41B structure. See Appendix A for details on the behavior of `pau`.

abbreviation	arguments	action

Control Instructions

`nop`	`---`	do nothing
`hlt`	`---`	stop the machine
`pau`	`---`	pause the machine
`jmp`	`R--`	`ic` = `dest`
`cal`	`RR-`	`dest` = `ic` and `ic` = `src0`

Table 3: The CS41B Assembly Language instructions that control the behavior of the machine. These instructions may change the `ic`, but they do not change any values in data registers or memory.

abbreviation	arguments	action

Branch Instructions

`brs`	`--S`	`ic` = `loc` + `arg`
`beq`	`RRS`	if `dest` = `src0`, `ic` = `loc` + `arg`
`bne`	`RRS`	if `dest` $\neq$ `src0`, `ic` = `loc` + `arg`
`blt`	`RRS`	if `dest` $<$ `src0`, `ic` = `loc` + `arg`
`ble`	`RRS`	if `dest` $\leq$ `src0`, `ic` = `loc` + `arg`
`bgt`	`RRS`	if `dest` $>$ `src0`, `ic` = `loc` + `arg`
`bge`	`RRS`	if `dest` $\geq$ `src0`, `ic` = `loc` + `arg`

Table 4: The CS41B Assembly Language branch instructions control the sequence of execution. These instructions change the instruction counter by (conditionally) adding a signed value to `ic`. The address of the currently-executing instruction is `loc`.

1.4 Programs

A *program* is simply a sequence of instructions stored in the machine's memory. In executing a program, the CS41B Machine follows a simple loop: The word at location `ic` is retrieved and interpreted as an instruction, and that instruction is carried out. Then the process repeats with the next instruction, until the `hlt` instruction is encountered.

More specifically, the CS41B Machine cycles through the following steps.

- 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.

Notice that, while an instruction is being executed, the value in `ic` is the address of the *next* instruction. This detail is important in implementing branch instructions.

1.5 Errors

There are two situations which will cause the CS41B Machine to report an error and halt.

1. A word that is to be interpreted as an address is too large for the actual memory.
2. A word that is to be interpreted as an instruction is not a legitimate instruction.

There are other errors from which the CS41B 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*.

2 Assembly Language and Programs

This section describes the assembly language that we use to write programs for the CS41B Machine.

2.1 Assembly Language Structure

An assembly language program is a sequence of lines, with each line being composed of a label, a command, arguments, and a comment. All the components are optional. The EBNF syntax for CS41B Assembly Language is given in Table 5.

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

Table 5: The EBNF specification for the CS41B 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 5, 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.

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.

A line is terminated by a system dependent end-of-line symbol. The nonterminals ⟨eoln⟩, ⟨blank⟩, ⟨label⟩, and ⟨comment⟩ are left undefined in Table 5. Usually, blanks are defined to include spaces and tab characters. The reader is invited to supply specifications for them.

Commands and Arguments A command is a three-letter assembly language instruction or one of the *directives* `dat` or `end`. An assembly language instruction

must be followed by arguments appropriate for that instruction. For example, `add` must be followed by three registers chosen from among `r0`, `r1`, `r2`, and `r3`. Byte or word arguments can be numeric values or labels.

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 CS41B instructions.

The directive `end` signals the end of the instructions and directives in the source file. Anything after an `end` directive is ignored. The `end` directive takes no arguments and is required.

2.2 Developing Programs

The following sections contains several examples of CS41B programs that illustrate the fundamental features of a programming language. We begin with a simple, linear, five instruction program and proceed to look at the structure of loops, conditional execution, and subprograms.

Assembly languages are often cryptic, and programs are difficult to follow. Programming requires careful attention to detail. Most assembly programmers adopt a high standard of documentation: virtually every line has a comment explaining what is happening.

Full listings of the programs developed in this chapter appear in Appendix B. Files containing the sample programs can be found in the on-line course resource materials.

2.3 Straight-line Code

The simplest of our example programs is one that accepts two integers and prints their difference. It has only five executable instructions: read, read, subtract, write, and halt. The instructions, shown below, are executed one at a time in order. Notice the use of `r0` in the load and store instructions to access the CS41B Machine's input and output facilities. This program appears among the example programs as `subtract.a41`.

```
loa r2 r0          ; get first value
loa r3 r0          ; get second value
sub r2 r2 r3       ; subtract them
sto r0 r2          ; print result
```

```
        hlt                 ; quit
        end
```

2.4 Loops

Suppose that we want to multiply two numbers instead of subtracting them. The CS41B 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.) 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--;
}
```

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.

```
        add r3 r0 r0        ; result = 0;
loop
        ble r1 r0 endloop   ; stop if a <= 0
        add r3 r3 r2        ; result += b;
        sbc 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.

```
        add r3 r0 r0        ; result = 0;
        ble r1 r0 endloop   ; test if a <= 0
loop
        add r3 r3 r2        ; result += b;
        sbc r1 r1 1         ; a--;
        blt r0 r1 loop      ; return for another iteration,
                            ;    if necessary
endloop
```

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 size of the loop body by one-fourth.

The loop above forms the core of the example program `multiply.a41` 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`.

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++;
}
```

2.5 Conditional Execution

The basic pattern for conditional execution is if-then-else. There are two blocks of code, and only one 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
        adc r3 r0 -1        ; place result -1 in r3
        sto r0 r3           ; write the value in r3
        brs endif           ; skip other clauses

elseif
        ble r2 r0 else      ; second comparison
        adc r3 r0 1         ; place result +1 in r3
        sto r0 r3           ; write the value in r3
        brs endif           ; skip other clauses

else
        sto r0 r0           ; write the value zero

endif
```

The example `sign.a41` has the full program.

2.6 Arrays and Nested Loops

The loop template from Section 2.4 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 zero or one, according to whether or not `i` is still on the list.

```
for (i = 2; i <= limit; i++)
    num[i] = 0;
```

```
for (i = 2; i <= limit; i++)
    if (num[i] == 0) {
        write(i);
        for (j = i; j <= limit; j += i)
            num[j] = 1;
    }
```

In the CS41B 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 byte, then we can put the value `num[i]` into `r1` with just two instructions.

```
        add r2 r3 r3
        loa r1 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 100. (Indices 0 and 1 are not used, but it is less confusing to include them.)

```
i       dat 2               ; i
num     dat  202            ; an array indexed 0 through 100;
                            ;   indices 0 and 1 are not used
```

The first loop in the sieve is easy to write. (It is also unnecessary, since the CS41B Machine initializes memory to zero.) We use `r3` for `i` and `r1` for the upper bound (100, in this case).

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

        blt r1 r3 enda      ; check if done
loopa
        add r2 r3 r3        ; get word offset
        sto r2 r0 num       ; num[i] := 0
        adc r3 r3 1         ; i++;
        ble r1 r3 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] = 1;
    }
```

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.

```
loopb
        blt r0 r? elsepart  ; branch if (num[i] != 0)

loopc

endc
        brs endif
elsepart
endif

endb
```

The first thing to notice is that the else-part is empty, so we need only one of the labels `elsepart` and `endif`, and we can omit the instruction `brs endif`. The outer loop has the same structure as `loopa` above, so we can copy that pattern.

```
        adc r3 r0 2         ; use r3 for i
        adc r1 r0 100       ; restore limit = 100;

        blt r1 r3 endb      ; check if done
loopb
        add r2 r3 r3        ; get word offset
        loa r2 r2 num       ; r2 now contains the value num[i]
        blt r0 r2 endif     ; branch if (num[i] != 1)

        ...                 ; write(i)

loopc
        ...

endc
        loa r3 r0 i         ; restore i
        adc r1 r0 100       ; restore limit = 100;
```

```
endif
        adc r3 r3 1         ; i++;
        ble r3 r1 loopb     ; go back for another round

endb
```

The only remaining tasks are to insert the output instruction and to fill in the body of `loopc`. One difficulty is that the body of `loopc` requires all three registers. We need `j`, `2j`, and the constant 1. We have to store `i` elsewhere, and that is why we reserved space for it at the top of our program.

Because the label `i` corresponds to a small address, we can use the same shortcut that we used with `num`. The label `i` is simply an "offset" from the address in register `r0`. Here is the piece that is missing from the code above.

```
        sto r0 r3           ; write(i);
        sto r0 r3 i         ; temporarily store i,
                            ;    and use r3 for j

        blt r1 r3 endc      ; check if done

loopc
        adc r1 r0 1         ; set r1 to 1
        add r2 r3 r3        ; get word offset
        sto r2 r1 num       ; num[j] = 1;
        loa r2 r0 i         ; get i back
        add r3 r3 r2        ; j += i;
        adc r1 r0 100       ; restore limit = 100;
        ble r3 r1 loopc     ; go back for another round

endc
```

None of these steps are hard, but the program is intricate. Programming at this level is a delicate task. All the pieces are assembled in the example program `sieve.a41`.

2.7 Stacks

Our method for saving and restoring the value of `i` in the Sieve of Eratosthenes was `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 CS41B 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*.

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 Probably the most important use 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. It does not matter much what decisions one makes, but once the decisions are made, strict adherence to them is vital. We adopt the following conventions for using registers and the stack on CS41B.

- `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 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
        sbc 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.a41`.

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              ; save local a
psh r2              ; save local 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.

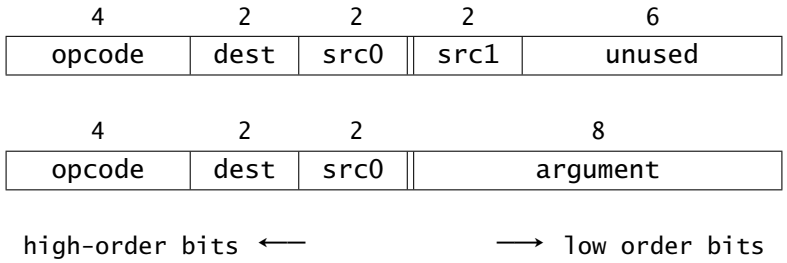
```
loopm
        loa r2 r1 2         ; recover b
        add r3 r3 r2        ; product += b;
        loa r2 r1 4         ; recover a
        sbc r2 r2 1         ; a--;
        sto r1 r2 4         ; store a
        blt r0 r2 loopm     ; go back for another round
```

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.a41`.

3 The CS41B Instruction Set

Until now, we have been using assembly language instructions and have not worried about how they are encoded as bits in memory. We now turn to the details of the encoding.

A machine-level instruction is a word whose bits encode an atomic action of the machine. An instruction has an operation name and up to three arguments. All the arguments, except perhaps the last, are registers. The last argument may be a register or a byte value. The two encoding schemes are shown below. The registers `r0` through `r3` are specified with two bits each.



The names of the instructions are *opcodes*. With four bits, we have sixteen opcodes, which correspond to some of the assembly instructions we have already seen. The remaining instructions are translated into machine instructions by the assembler.

Table 6 summarizes the instructions of the CS41B Machine. Every instruction except `hlt` modifies the instruction counter. An instruction may modify the value in a data register or one in a memory location, but not both. As is customary, the first register argument is denoted `dest`, and the other two are denoted `src0` and `src1`.

For example, the (nonsensical) instruction sequence

opcode (hex)	abbreviation	arguments	action
		Control Instructions	
0	nop	---	do nothing
1	hlt	---	stop the machine
2	pau	---	pause the machine
3	cal	RR-	dest = ic and ic = src0
		Branch Instructions	
4	beq	RRS	if dest = src0, ic += arg
5	bne	RRS	if dest ≠ src0, ic += arg
6	blt	RRS	if dest < src0, ic += arg
7	bge	RRS	if dest ≥ src0, ic += arg
		Data Moving Instructions	
8	sto	RRS	mem[dest + arg] = src0
9	loa	RRS	dest = mem[src0 + arg]
a	lcl	R-U	lowbyte(dest) = arg
b	lch	R-U	highbyte(dest) = arg
		Arithmetic Instructions	
c	add	RRR	dest = src0 + src1
d	sub	RRR	dest = src0 − src1
e	adc	RRS	dest = src0 + arg
f	sbc	RRS	dest = src0 − arg

Table 6: The CS41B Machine instruction set. The third argument, when it exists, may be a register, a signed byte, or an unsigned byte.

```
target
        add r3 r2 r2
        loa r2 r1 4
        blt r3 r0 target
```

is encoded as

```
ce80
9904
66fa
```

Notice that the signed byte argument to the `blt` instruction is `0xfa` or -6, not -4 as one might expect. That is because the branch is relative to the current instruction counter, which is the location of the *next* instruction to be executed. The assembly language branch instructions take arguments that are relative to the location of the current instruction, while the machine branch instructions take arguments that are relative to the current value of the instruction counter. Keep that distinction in mind if you are ever tempted to give a numerical argument to a branch instruction! One big advantage of using labels is that the assembler takes care of computing the correct offset for the machine instruction. Practice translating a few other instruction sequences by hand.

One important consequence of the instruction format is that the argument to a branch instruction is stored in eight bits, interpreted as a signed value. That means that the target of the branch must be between -128 and $+127$ bytes away from the current value of the instruction counter. It can be a severe limitation in some cases, but there are creative ways to work around it.

At the machine level, the byte argument to `sto` and `loa` is *not* optional. The assembler inserts a zero value when necessary.

The remaining assembly language instructions are *synthetic instructions.* The assembler translates them into one or more machine instructions from Table 6. Here are the definitions of the synthetic instructions.

`jmp dest` is represented as `cal r0 dest`.

`brs arg` is represented as `beq r0 r0 arg`. It is the conditional branch that is always taken.

`ble dest src0 arg` is represented as `bge src0 dest arg`.

`bgt dest src0 arg` is represented as `blt src0 dest arg`.

`psh dest` is represented by the sequence `sto r1 dest 0; sbc r1 r1 2`.

`pop dest` is represented by the sequence `adc r1 r1 2; loa dest r1 0`.

`mov dest src0` is represented by `add dest r0 src0`.

`neg dest src0` is represented by `sub dest r0 src0`.

`lcw dest word` is represented by the sequence `lcl dest wlow`; `lch dest whigh`, where `wlow` and `whigh` are the low and high bytes, respectively, of `word`.

The opcodes for the conditional branch instructions illustrate how the encoding can be chosen to make instructions easy to decode. The bit pattern for a conditional branch opcode is `01xy`. The value of `x` specifies the type of test: equal or less-than. The value of `y` tells whether the result of the test is taken as-is or is negated.

The format for a CS41B program file is a sequence of lines, each one containing four hexadecimal characters. The first line is the address in memory where the program is to be loaded and execution is to start. When loading a program, the machine sets `ic` to the value on the first line and then puts the subsequent values into memory sequentially, starting with the address on the first line.

4 Real Computers

The CS41B Machine contains most of the features of a real computer, at least in primitive form. In this section, we explore how the CS41B concepts are extended in real computers.

4.1 Registers and Memory

As in the CS41B Machine, memory in a real computer is indexed by bytes, but the word size is larger. The 32-bit word is the standard, and 64-bit words are becoming more common.

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.

4.2 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 floating point numbers. Finally, there are specialized instructions that are used by the operating system; see Section 4.5.

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 CS41B 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 kinds of instructions, `RRR` and `RRV`. (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 CS41B 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 1 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.

4.3 Conditional Branches

The conditional branch instructions on the CS41B 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 2. Such solutions involve complicated case-by-case analyses and would give unacceptable

```
_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 1: 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.

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 dest src0 arg` would then, in three steps, compute `dest` − `src0`, 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.
- 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 1 and notice the branches. The sequence `cmpl; jle` corresponds to CS41B's `ble`.

4.4 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.

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

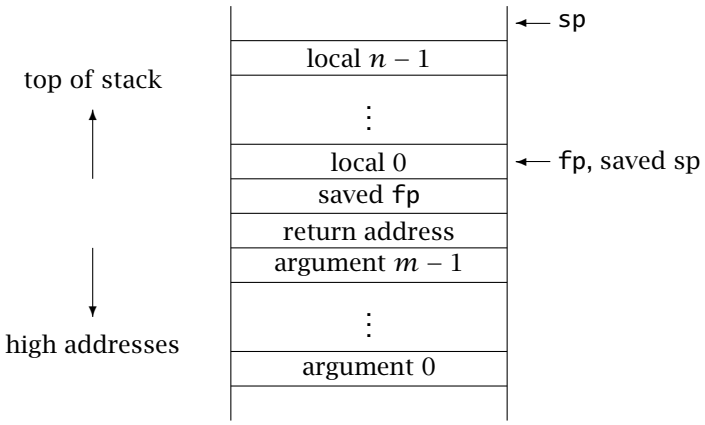


Figure 2: A typical stack frame for a subprogram call with m arguments and n local variables. As shown, the stack grows from higher addresses to lower ones.

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 2, 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 value of `fp` is copied to `sp` by the callee, effectively discarding the local 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 essential 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 1 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 `ret`. It is a very good example of how a complex instruction set can package frequently-done operations.

4.5 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 CS41B application is the *loader*, which reads an external program file and copies its contents into memory. In the CS41B 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 can write a loader for the CS41B Machine; see Exercise 17. It is awkward, however, to enter a program through CS41B's only input channel as a sequence of signed integers.

Memory Management On a real computer, several programs run at the same time. Each one has 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 CS41B 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 CS41B 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, 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 CS41B Application

We will work with a Java application that assembles and runs CS41B programs. The code for the application is contained in a Java jar file, found at `/common/cs/cs052/cs41b/cs41b.jar`. In the laboratories, you may start it by double clicking on the jar file or by executing the command `/common/cs/cs052/bin/cs41b` in a terminal window. 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:

`.a41`	a source program in assembly language
`.i41`	the intermediate file
`.m41`	the object program, executable by the CS41B Machine

The intermediate files is temporary and should be present during the assembly process.

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 3. There are two views of memory: On the left is the

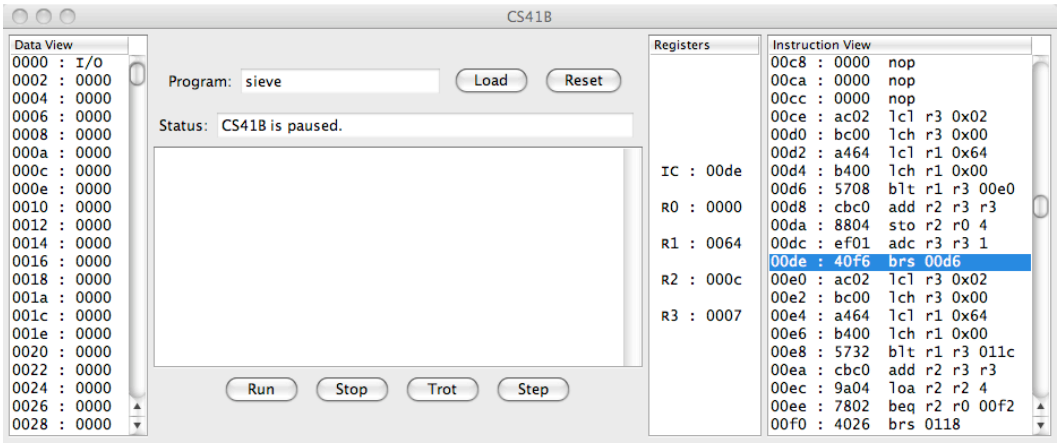


Figure 3: The appearance of the CS41B Machine.

Data View in which the memory values are interpreted as data, and on the right is the Instruction View in which the memory values (all of them!) are interpreted as instructions. Adjacent to 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.

The buttons have been designed to work in an intuitive manner. The Load button opens a file selector window from which you can choose the file to run. The Reset button takes the machine back to the state at which the program was originally loaded.

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.

The four buttons at the bottom control the execution of the program. The Run button causes the program to run at full speed. The Trot button slows the program to about two instructions per second. The Step button allows you to step through the program one instruction at a time. The Stop button halts execution temporarily. You may shift between execution modes at any time.

Executing a pau instruction in the visual application has the same effect as pressing the Stop button. You can debug a portion of your program by inserting a pau instruction and then examining the registers or stepping through the code.

If the machine has halted normally, pressing the Run, Trot, or Step 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.

Exit the application by closing the window.

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 `cs41b.jar`. Examples of paths are `~/cs052/cs41b.jar` and `/common/cs/cs052/cs41b/cs41b.jar`.

Assembler The argument for the stand-alone assembler is `-a <filename>`. The named file must be a CS41B source file with the extension `.a41`. The action is to assemble the file and produce the object file with the same name and extension `.m41`. Example:

```
java -jar <path-to-jar-file> -a sieve.a41
```

Simulator The argument for the stand-alone simulator is `-r <filename>`. The named file must be a CS41B object file with the extension `.m41`. The action is to run the program in the file. One can optionally give a specification for the machine's memory size, `-m <memory-size>`. The memory size is specified in bytes ranging from 64 to $2^{16} = 65,536$; the default size is 512. Example:

```
java -jar <path-to-jar-file> -r sieve.m41 -m 256
```

Executing the the `pau` instruction in the simulator is the same as a `nop`.

Visual Application One can also start the visual application from the command line, with the advantage of optional arguments that control the memory size and trot speed. The optional specifications are `-m <memory-size>`, as described above, and `-t <trot-speed>`. The trot speed is specified in instructions per minute, ranging from 6 to 60,000. The default speed is 120, two instructions per second. Example:

```
java -jar <path-to-jar-file> -m 64 -t 3000
```

At this time, the command line is the only way to specify memory size and trot speed.

A.4 Error Messages

Here is a list of error messages generated by the application. Most of them correspond to errors found by the assembler, before the program runs.

Illegal character The assembler found a character that is not in alphabet of assembly language. Use only letters, plus, minus, underscore and semicolon.

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

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

Data in executable block A `dat` directive may not appear between instructions; see page 8.

No end directive found Your file must conclude with `end`.

No executable code The assembler found no instructions to execute.

Wrong number of arguments The assembler encountered an instruction with too many, or two few, arguments.

Incorrect arguments The assembler found an instruction with the wrong type of arguments.

Byte value out of range The numeric argument to instructions like `adc` and `loa` must fit into a single byte.

Byte offset out of range for label A branch, when taken, adds an eight-bit signed quantity to the IC. This error is telling you that the target is too far away from the branch instruction. See page 22.

Byte value out of range for label It is unlikely that you will use a label for the numeric argument to instructions like `adc` or `loa`, but when you do, it must be in the range of an eight-bit unsigned quantity.

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.

File not found A file with the name you specified and the extension `.a41` could not be located by the assembler.

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

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

IC out of range The instruction counter is referring to a location that is not in the machine's memory.

Address out of bounds The machine was asked to load from, or store to, an address that is not in the machine's memory.

File not found A file with the name you specified and the extension `.m41` could not be located by the computer.

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.

Cannot happen

Garbled intermediate file

Illegal register index

A.5 Defects and Improvements

Please report errors and unexpected behaviors. Suggestions for improvement are also welcome. Here are some proposed enhancements; let us know if you think they are worthwhile.

- Add a way to set the memory size from within the application.
- Add a way to adjust the trot speed from within the application. It could be a simple way to change the trot speed, or it could be a single "speed control" slider that replaces the Run, Trot, and Step buttons.
- Add a Stack View of memory. It would track the top of the stack, just as the Instruction View tracks the instruction counter. It could be an additional panel, or an option to the Data View.
- Add a resettable step counter.
- Add a "step backwards" facility. It would require extensive programming.

A.6 Legal Stuff

B Sample Programs

Here is a complete listing of all the sample programs derived in the text. The programs are also available in the source directory /common/cs/cs052/cs41b.

B.1 Subtract

```
;
; subtract.a41
;
; A simple CS41B program that subtracts
; two numbers.
;
; Rett Bull
; Pomona College
; July 17, 2009
;
        loa r2 r0           ; get first value
        loa r3 r0           ; get second value
        sub r2 r2 r3        ; subtract them
        sto r0 r2           ; print result
        hlt                 ; quit
        end
```

B.2 Sign

```
;
; sign.a41
;
; A CS41B program that illustrates conditional
; execution by determining the sign (-1, 0, or +1)
; of an integer.
;
; Rett Bull
; Pomona College
; August 7, 2009
;
;
;       if (a < 0)
;          write(-1);
;       else if (0 < a)
;          write(1);
;       else
;          write(0);
;
;
        loa r1 r0 0         ; get a value for a

        ble r0 r1 elseif    ; first comparison
        adc r3 r0 -1        ; place result -1 in r3
        brs endif           ; skip other clauses

elseif
        ble r1 r0 else      ; second comparison
        adc r3 r0 1         ; place result +1 in r3
        brs endif           ; skip other clauses

else
        add r3 r0 r0        ; place result 0 in r3

endif
        sto r0 r3 0         ; write the value in r3
        hlt                 ; halt
        end
```

B.3 Multiply

```
;
; multiply.a41
;
; A CS41B program that illustrates a loop by
; using iterated addition to multiply two numbers.
;
; Rett Bull
; Pomona College
; August 7, 2009
;
;
;       result = 0;
;       while (0 < a) {
;          result += b;
;          a--;
;       }
;       write(result);
;
;
        loa r1 r0 0         ; get a value for a
        loa r2 r0 0         ; get a value for b
        add r3 r0 r0        ; result = 0;

        ble r1 r0 endloop   ; test if a <= 0
loop
        add r3 r3 r2        ; result += b;
        sbc r1 r1 1         ; a--;
        blt r0 r1 loop      ; return for another iteration

endloop
        sto r0 r3 0         ; write the value of product
        hlt                 ; halt
        end
```

B.4 Sieve of Eratosthenes

```
;
; sieve.a41
;
; A CS41B program that executes the Sieve of Eratosthenes
; to illustrate arrays, conditional execution, and nested
; loops.
;
; Rett Bull
; Pomona College
; Originally written for CS41A in the Fall of 1989
; Adapted to CS41B on July 28, 2009
; August 7, 2009
;
;
;       for (i = 2; i <= 100; i++)
;           num[i] = 0;
;       for (i = 2; i <= 100; i++)
;           if (num[i] == 0) {
;              write(i);
;              j = i;
;              while (j <= 100) {
;                  num[j] = 1;
;                  j += i;
;              }
;           }
;
; We use all three general registers; there is no
; stack.
;
;



;
; Data area: We set aside a location to store i and a sequence
; of words for the array num.
;
i       dat 2               ; i
                            ;   indices 0 and 1 are not used
num     dat  202            ; an array indexed 0 through 100;
```

```
;
; First loop: It is actually unnecessary because memory is
; already initialized to zeroes
;
; 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.
;
        adc r3 r0 2         ; i = 2;
        adc r1 r0 100       ; limit = 100;

        blt r1 r3 enda      ; check if done
loopa
        add r2 r3 r3        ; get word offset
        sto r2 r0 num       ; num[i] := 0
        adc r3 r3 1         ; i++;
        ble r1 r3 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.
;
        adc r3 r0 2         ; use r3 for i
        adc r1 r0 100       ; restore limit = 100;

        blt r1 r3 endb      ; check if done
loopb
        add r2 r3 r3        ; get word offset
        loa r2 r2 num       ; r2 now contains the value num[i]
        blt r0 r2 endif     ; branch if (num[i] != 1)

        sto r0 r3           ; write(i);
        sto r0 r3 i         ; temporarily store i,
                            ;    and use r3 for j

        blt r1 r3 endc      ; check if done

loopc
        adc r1 r0 1         ; set r1 to 1
        add r2 r3 r3        ; get word offset
        sto r2 r1 num       ; num[j] = 1;
        loa r2 r0 i         ; get i back
        add r3 r3 r2        ; j += i;
        adc r1 r0 100       ; restore limit = 100;
        ble r3 r1 loopc     ; go back for another round

endc
        loa r3 r0 i         ; restore i
        adc r1 r0 100       ; restore limit = 100;

endif
        adc r3 r3 1         ; i++;
        ble r3 r1 loopb     ; go back for another round

endb
        hlt                 ; we're done!
        end                 ;
```

B.5 Factorial

```
;
; factorial.a41
;
; A CS41B program that computes factorials. It
; illustrates subprogram calls and the use of the
; stack for local variables.
;
; Rett Bull
; Pomona College
; August 7, 2009
;
;       int factorial(int x) {
;           if (x <= 0)
;              return 1;
;           else
;              return mult(factorial(x-1), x);
;       }
;
;       int mult(int a, int b) {
;           int product = 0;
;           if (a < 0) {
;              a = -a;
;              b = -b;
;           }
;           while (0 < a) {
;              product += b;
;              a--;
;           }
;           return product;
;       }
;
; We adopt the CS41B 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 r0 r3           ; write result,
        hlt                 ;    and halt


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

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

recursion
        psh r3              ; save argument
        sbc 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
                            ; result is already in r3

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

```
;
; mult subprogram: The incoming arguments are a (in
; r3) and b (on the stack). We use copies of them as
; local variables on the stack.
;
mult
        psh r2              ; save return address
        loa r2 r1 4         ; get b

        ble r0 r3 notneg    ; adjust signs, if necessary
        neg r3 r3           ;
        neg r2 r2           ;

notneg
        psh r3              ; save a
        psh r2              ; save b
        adc r3 r0 0         ; initialize product

        loa r2 r1 4         ; recover a
        ble r2 r0 endm      ; test if done

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

endm
        pop r0              ; discard local b
        pop r0              ; discard local a
        pop r2              ; restore return address
        jmp r2              ; return


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

C Historical Note

The CS41 machines and the assignments surrounding them were inspired in the 1980's 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. 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.

Even though course numbers have changed over the years, we have not changed the name. The CS41B Machine honors those students who labored with CS41A in the early years of Computer Science at Pomona College.

D Exercises

1. The sample program `multiply.a41` works only when the first number is non-negative. Modify it to handle negative values.

2. A "missing" instruction in the CS41B Machine is `blu`, branch on *unsigned* less-than. Write a sequence of instructions to simulate it.

3. Write a sequence of CS41B 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;
```

4. Euclid's algorithm for computing the greatest common divisor of two integers is expressed in the following program. Write a CS41B subprogram to simulates it.

```
int gcd(int a, int b) {
    a = |a|;
    b = |b|;
    while (0 < a && 0 < b)
       if (a < b)
          b = b - a;
       else
          a = a - b;
    return (a == 0) ? b : a;
}
```

5. Here is a recursive version of Euclid's algorithm, in which we assume that the arguments are not negative. Write a CS41B subprogram to simulate it.

```
int gcd(int a, int b) {
    if (a == 0)
       return b;
    else if (a < b)
       return gcd(b, a);
    else
       return gcd(a - b, b);
}
```

6. The sequence of Fibonacci numbers is defined recursively: $f_0 = f_1 = 1$ and $f_n = f_{n-1} + f_{n-2}$. Write a CS41B subprogram to compute elements of the Fibonacci sequence recursively, by following the definition precisely.

7. Write a CS41B subprogram to compute elements of the Fibonacci sequence iteratively, as illustrated in the function below.

```
int fib(int n) {
    int ultimate = 1;
    int penultimate = 1;
```

```
    for (int j = 1; j < n; j++) {
       ultimate = ultimate + penultimate;
       penultimate = ultimate - penultimate;
    }
    return ultimate;
}
```

8. The Takeuchi function was designed to test the speed at which computer systems compute recursive functions. It makes many calls, but the numbers do not get very large.

$$\texttt{tak}(x,y,z) = \begin{cases} y & \text{if } x \le y, \text{ and} \\ \texttt{tak}(\texttt{tak}(x-1,y,z), & \\ \quad \texttt{tak}(y-1,z,x), & \\ \quad \texttt{tak}(z-1,x,y)) & \text{otherwise} \end{cases}$$

Write a CS41B subprogram that computes the Takeuchi function.

9. The following code shifts the bits of `orig`, one at a time, into `copy`.

```
int copy = 0;
for (int j = 16; 0 < j; j--) {
    copy = copy << 1;
    if (orig < 0)  // if (sign bit == 1)
       copy++;
    orig = orig << 1;
}
```

Note that the one-bit left shift is the equivalent of doubling a number. It can be carried out by a CS41B instruction like `add r3 r3 r3`, and so the code can be expressed in a CS41B program. Use a variation of this idea to write a CS41B subprogram that carries out an arithmetic *right* shift. In other words, write a subprogram that takes a number n and returns $n/2$.

10. The Collatz function is defined on positive integers by the equation

$$\texttt{collatz}(n) = \begin{cases} 1 & \text{if } n = 1, \\ \texttt{collatz}(n/2) & \text{if } n \text{ is even, and} \\ \texttt{collatz}(3n+1) & \text{otherwise.} \end{cases}$$

The Collatz function is interesting to mathematicians because it is not known if the function is defined on all the positive integers. Write a CS41B subprogram that computes the Collatz function. Have your function return zero if the argument is not positive. (Suggestion: Use a modification of the subprogram from Exercise 9 that returns $n/2$ in `r3` and $n\%2$ on the stack.)

11. Write a short C or C++ program that explores its own stack. See if you can locate the arguments, frame pointer, and return address. (Hint: Get an approximate value of the frame pointer by using the & operator to find the address of a local variable.)

12. The CS41B instruction set, as described in Table 6, is not as compact as it could be. Here are three suggestions for replacing machine instructions with synthetic assembly language instructions. For each one, determine if the substitution is an exact match for the original instruction. If it is not, explain how the instructions are different and whether or not you think the synthetic instruction is an "adequate substitute."

i. Replace `nop arg` with `bne r0 r0 arg`.

ii. Replace `lcl dest arg` with `adc dest r0 arg`.

iii. Replace `sbc dest src0 arg`with `adc dest src0 (-arg)`, where the negation is computed by the assembler.

13. Write a CS41B program that prints its own sequence of machine instructions. The result will appear in the machine's output as a sequence of words expressed as signed integers.

14. Write a CS41B subprogram `mulbytes` that takes two byte arguments, one in `r3` and one on the stack, multiplies them, and returns the result in `r3`. (By byte argument, we mean a word whose upper byte is zero. Treat the bytes as unsigned values.)

15. Using the result of the previous exercise, write a subprogram `mulwords` that takes two full-word (unsigned) arguments, multiplies them, and returns the product. The product will be two words wide; return the lower word in `r3` and the upper word on the stack.

16. Use the result of the previous exercise to write a subprogram that does *signed* multiplication.

17. Write a CS41B program that behaves as a *loader:* it receives a program as a sequence of signed integers from the machine's input, places the program in memory, executes it, and then returns for another program. (We must redefine "program" for this exercise. A program to be loaded and executed is a subprogram called by the

loader. After setting up its stack, the program should push the return address in `r2`. On completion, instead of executing `hlt`, it should pop the return address and jump to it. Also, the loader must be told when to stop reading a program from the input. We can assume that a program contains no `nop` instructions, so that a zero word signals the end-of-input.)

18. Design a variable-length instruction set with the same capabilities as the CS41B Machine. Try to minimize the number of unused bits.

E Assembly Language Quick Reference

abbreviation	arguments	action

Register Instructions

mov	RR-	dest = src0
neg	RR-	dest = −src0
add	RRR	dest = src0 + src1
sub	RRR	dest = src0 − src1
adc	RRS	dest = src0 + arg
sbc	RRS	dest = src0 − arg
lcw	R-W	dest = arg
lcl	R-U	lowbyte(dest) = arg
lch	R-U	highbyte(dest) = arg

Memory Instructions

sto	RR[S]	mem[dest + arg] = src0
loa	RR[S]	dest = mem[src0 + arg]
psh	R--	push the value in dest
pop	R--	pop the top of stack into dest

Control and Branch Instructions

nop	---	do nothing
hlt	---	stop the machine
pau	---	pause the machine
jmp	R--	ic = dest
cal	RR-	dest = ic and ic = src0
brs	--S	ic = loc + arg
beq	RRS	if dest = src0, ic = loc + arg
bne	RRS	if dest ≠ src0, ic = loc + arg
blt	RRS	if dest < src0, ic = loc + arg
ble	RRS	if dest ≤ src0, ic = loc + arg
bgt	RRS	if dest > src0, ic = loc + arg
bge	RRS	if dest ≥ src0, ic = loc + arg