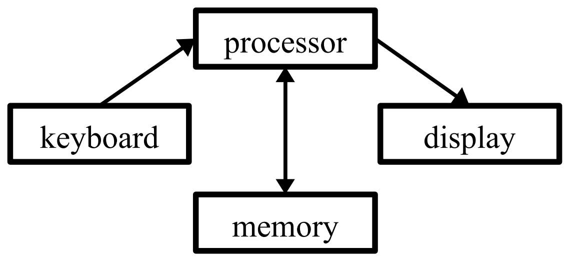
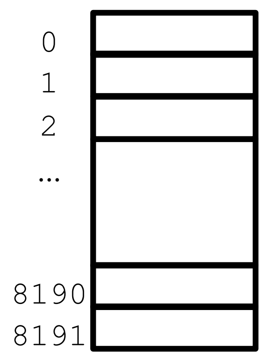
CS252 Object-Oriented Programming with Java (Zaring)
The VM252 Virtual Machine

1. Overview

The VM252 is an extremely simple virtual computer, somewhat reminiscent of some of the earliest electronic computing devices. It can be roughly depicted as

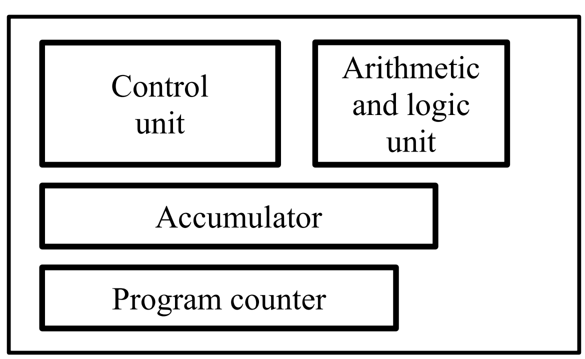


where input (in the form of human-readable integer values) is read from a keyboard and output (in the form of human-readable integer values) is printed to a display. The memory is a collection of eight-bit bytes



Each byte can be accessed by referring to its unique index (hereafter called the *memory address* or *address* of the byte). The bytes of memory hold both the binary encodings of the data values manipulated by programs as well as the binary encodings of the instructions that make up the program.

The processor of the machine contains four components



The *control unit* is responsible for executing the sequences of instructions that make up programs. The *arithmetic and logic unit* (abbrev. *ALU*) is responsible for performing any/all arithmetic operations within the processor. The *accumulator* is a sixteen-bit storage unit (a *register*) that holds values interpreted as signed integers and is the focus of most of the instructions the processor can execute. The *program counter* (a register) is used by the control unit to determine where in memory the encoding of the next instruction to be executed resides.

2. Control-Unit Semantics Modeled as Java Pseudocode

(For those unfamiliar with the programming language Java, see Appendix A for a Python pseudocode-based presentation of the material in the section.)

The major components of the VM252 correspond roughly to

```
public static void ModelOfVM252ControlUnit()
{

    short ACC;  // the accumulator
    short PC;  // the program counter
    final byte [] memory = new byte[ 8192 ];
    final Scanner in = new Scanner(System.in);
    final PrintStream out = System.out;

    …

    }
```

The behavior of the twelve instructions that comprise the *instruction set* of the VM252 (i.e., the complete repertoire of instructions that the control unit, in concert with the ALU, can carry out can then be described as in Table 1, noting that in each instance in this table, $0 \le a \le 8191$ and $-2048 \le c \le 2047$. (See Appendix B for a brief discussion of *two's complement integer representation*.)

Table 1: VM252 Instruction Semantics

Instruction in symbolic form	*Instruction meaning in Java pseudocode*

`INPUT`	`{`
	`ACC = in.nextShort();`
	`in.nextLine();`
	`}`

`OUTPUT`	`out.println(ACC);`
`NOOP`	`;  // do nothing`
`STOP`	*halt the processor*`;`
`LOAD a`	`ACC =` *(the 16 bits from* `memory[a]` *and* `memory[(a+1)%8192]` *treated collectively as a 16-bit two's complement integer)*`;`
`STORE a`	*(the 16 bits in* `memory[a]` *and* `memory[(a+1)%8192]` *treated collectively as a 16-bit two's complement integer variable)* `=` ACC;
`ADD a`	`ACC +=` *(the 16 bits from* `memory[a]` *and* `memory[(a+1)%8192]` *treated as a 16-bit two's complement integer)*`;`

Instruction in symbolic form	*Instruction meaning in Java pseudocode*
`SUB` *a*	`ACC -=` *(the 16 bits from* `memory[`*a*`]` *and* `memory[(`*a*`+1)%8192]` *treated as a 16-bit two's complement integer)*`;`
`JUMP` *a*	`PC =` *a*`;`

`JUMPZ` *a*	`if (ACC == 0)`
	`    PC =` *a*`;`
	`else`
	`    PC = (PC + 2) % 8192;`

`JUMPP` *a*	`if (ACC > 0)`
	`    PC =` *a*`;`
	`else`
	`    PC = (PC + 2) % 8192;`

`SET` *c*	`ACC =` *(the 12 bits of* `c` *treated as a 12-bit two's complement integer sign-extended to a 16-bit two's complement integer)*`;`

These twelve instructions are sufficient to perform any integer computation that can be carried out on any existing computer.

The program hardwired into the hardware of the control unit corresponds roughly to the pseudocode

```
copy the B bytes of the executable representations of the program instructions into the
        respective elements of memory[0] ... memory[B−1];

PC = 0;

opcode = the portion of memory[PC] that distinguishes among the twelve different
        types of instructions (i.e., the "operation code" or "opcode");

while (opcode != STOP) {

    perform the operation indicated by opcode and (when necessary) the operand formed
     from  the relevant portions of memory[PC] and memory[(PC+1) % 8192];

    if (opcode == JUMP || opcode == JUMPZ || opcode == JUMPP)
        ; // do nothing
    else if (opcode == NOOP || opcode == INPUT
            || opcode == OUTPUT)
        PC = (PC + 1) % 8192;
    else
        PC = (PC + 2) % 8192;

    opcode = the opcode portion of memory[PC];

    }

halt the processor;
```

3. Instruction Encoding:

Every type of instruction is encoded as a sequence of either eight or sixteen bits, depending on the type of the instruction, as shown in Table 2 and Table 3. Note that, in these tables,

- The values of any/all bits shown as x are irrelevant and are ignored.
- The values of bits shown as a_j collectively encode a memory-byte address as a 13-bit unsigned integer.
- The values of bits shown as c_j encode a signed-integer data value as a 12-bit two's complement integer.

(See Appendix B for a brief discussion of unsigned integer representation and two's complement integer representation.)

Table 2: Instructions Having Eight-Bit Encodings

Instruction shown in symbolic form	*Instruction as encoded in 8 bits and stored in a byte of memory*
`INPUT`	`111100`*xx*
`OUTPUT`	`111101`*xx*
`NOOP`	`111110`*xx*
`STOP`	`111111`*xx*

Table 3: Instructions Having Sixteen-Bit Encodings

Instruction as shown in symbolic form	*Instruction as encoded in 16 bits*	*Instruction bits as stored in two consecutive bytes of memory*
`LOAD` a	$000a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	$000a_{12}a_{11}a_{10}a_9a_8$ $a_7a_6a_5a_4a_3a_2a_1a_0$
`STORE` a	$001a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	$001a_{12}a_{11}a_{10}a_9a_8$ $a_7a_6a_5a_4a_3a_2a_1a_0$
`ADD` a	$010a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	$010a_{12}a_{11}a_{10}a_9a_8$ $a_7a_6a_5a_4a_3a_2a_1a_0$
`SUB` a	$011a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	$011a_{12}a_{11}a_{10}a_9a_8$ $a_7a_6a_5a_4a_3a_2a_1a_0$
`JUMP` a	$100a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	$100a_{12}a_{11}a_{10}a_9a_8$ $a_7a_6a_5a_4a_3a_2a_1a_0$
`JUMPZ` a	$101a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	$101a_{12}a_{11}a_{10}a_9a_8$ $a_7a_6a_5a_4a_3a_2a_1a_0$
`JUMPP` a	$110a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$	$110a_{12}a_{11}a_{10}a_9a_8$ $a_7a_6a_5a_4a_3a_2a_1a_0$
`SET` c	$1110c_{11}c_{10}c_9c_8c_7c_6c_5c_4c_3c_2c_1c_0$	$1110c_{11}c_{10}c_9c_8$ $c_7c_6c_5c_4c_3c_2c_1c_0$

The specific type of an instruction can be determined from the leftmost six bits of the first byte of that instruction's encoded form.

For example, the following program (which reads in an integer and then prints out that integer plus one)

```
        INPUT
        STORE  subject
        SET  1
        ADD  subject
        OUTPUT
        STOP
    subject:
        DATA  0
```

would be encoded as the following 11 bytes (where the color of the bits in the below matches the color of the opcode or operand represented by those bits)

`11110000`	***INPUT***
`00100000`	***STORE subject*** (≡ ***STORE 9***)
`00001001`	
`11100000`	***SET 1***
`00000001`	
`01000000`	***ADD subject*** (≡ ***ADD 9***)
`00001001`	
`11110100`	***OUTPUT***
`11111100`	***STOP***
`00000000`	*DATA* ***0***
`00000000`	

4. AssemblyLanguage

When programming the VM252, the only programming language available is *assembly language*, a minimally-humanized notation for writing down the instructions in the VM252 instruction set in addition to a very few conveniences.

A VM252 assembly-language program consists of a plain-text file containing a sequence of ASCII characters defined by the grammar

program	→	*statement newline* | *statement newline program*
statement	→	*instruction* | *dataDirective* | *symbolicAddressDefinition*
instruction	→	`LOAD` *numericValue* | `load` *numericValue*
	|	`STORE` *numericValue* | `store` *numericValue*
	|	`ADD` *numericValue* | `add` *numericValue*
	|	`SUB` *numericValue* | `sub` *numericValue*
	|	`JUMP` *numericValue* | `jump` *numericValue*
	|	`JUMPZ` *numericValue* | `jumpz` *numericValue*
	|	`JUMPP` *numericValue* | `jumpp` *numericValue*
	|	`SET` *numericValue* | `set` *numericValue*
	|	`INPUT` | `input`
	|	`OUTPUT` | `output`
	|	`NOOP` | `noop`
	|	`STOP` | `stop`
dataDirective	→	`DATA` *numericValue* | `data` *numericValue*
symbolicAddressDefinition	→	*identifier* **:**
numericValue	→	*decimalIntegerLiteral*
	|	*hexadecimalIntegerLiteral*
	|	*identifier*
decimalIntegerLiteral	→	*any string of one or more digits* `0-9`*, optionally preceded by a + or −*
hexadecimalIntegerLiteral	→	`0x` *or* `0X` *followed by any string of one or more digits 0-9,* `a-f`*, and/or* `A-F`*, optionally preceded by a + or −*
identifier	→	*any string of one or more digits* `0-9`*, letters* `a-z`*, letters* `A-Z`*, or underscores, staring with a letter* `a-z`*, letter* `A-Z`*, or underscore*
newline	→	*the character sequence that signals the end of a line of text*

Whitespace is considered to be irrelevant, except for the newline that must terminate every statement. A bang/exclamation point ("!") starts a to-the-end-of-the-line style of comment.

An *instruction* specifies an executable instruction in the VM252 instruction set (see Section 2 for more details).

A *dataDirective* specifies a two-byte signed-integer value to be stored initially into memory at the point in the program where the directive occurred. Such directives do not correspond to any execution-time operation and are used to reserve bytes to serve as program variables.

A *symbolicAddressDefinition* defines a name that may be used to stand for the run-time memory of the point in the program corresponding to the relative position at which the definition occurred in the program. Such definitions do not correspond to any execution-time operation.

As an example, consider the following program to read in three integers, calculate their difference, and then print out that difference:

```
        INPUT
        STORE  subjectA
        INPUT
        STORE  subjectB
        INPUT
        STORE  subjectC
        LOAD  subjectA
        SUB subjectB
        SUB subjectC
        OUTPUT
        STOP
    subjectA:
        DATA  0
    subjectB:
        DATA  0
    subjectC:
        DATA  0
```

Note that it would be incorrect simply to move the symbolic-address definitions and data directives o the beginning of the program, as in

```
    subjectA:
        DATA  0
    subjectB:
        DATA  0
    subjectC:
        DATA  0
        INPUT
        STORE  subjectA
        INPUT
        STORE  subjectB
        INPUT
        STORE  subjectC
        LOAD  subjectA
        SUB subjectB
        SUB subjectC
        OUTPUT
        STOP
```

since, according to the control-unit's semantics (see Section 2), the bytes reserved by the data directives would then be "executed" as if they were the first instructions in the program. The only way to prevent this would be to do something like the following instead:

```
        JUMP  main
    subjectA:
        DATA  0
    subjectB:
        DATA  0
    subjectC:
        DATA  0
    main:
        INPUT
        STORE  subjectA
        INPUT
        STORE  subjectB
```

```
        INPUT
        STORE  subjectC
        LOAD  subjectA
        SUB subjectB
        SUB subjectC
        OUTPUT
        STOP
```

5. Basic Assembly-Language Programming

When working with assembly language, it's always best to analogize with high-level language programming wherever possible. This approach is most likely to produce a working program; however, it may yield a program that naively contains needlessly redundant code and/or code that fails to exploit some of the possibilities available when programming with assembly language. If desired, any such issues can be addressed in a late-stage round of program "optimization".

To start programming, first design the program in Java (perhaps including some pseudo-code, when necessary). Consider a program to read in two integers and print the larger of the two. An obvious Java program for this would be something like

```
    public static void main(String [] commandLineArguments)
    {

        final Scanner in = new Scanner(System.in);

        int a, b;

        a = in.nextInt();
        b = in.nextInt();

        System.out.println(b > a ? b : a);

        }
```

Continue by simplifying the Java code to use the simplest, least "exotic" types of Java expressions and statements. For example, in the current program, the conditional expression should be replaced with a simpler conditional statement (which, in this situation, requires the declaration of an additional variable):

```
    public static void main(String [] args)
    {

        final Scanner in = new Scanner(System.in);

        int a, b, larger;

        a = in.nextInt();
        b = in.nextInt();

        if (b > a)
            larger = b;
        else
            larger = a;

        System.out.println(larger);

        }
```

Once the Java has been simplified to use only the simplest possible kinds of Java statements and expressions, one proceeds with a line-by-line conversion of the Java code into assembly language. To start with,

- The variable declarations should become labeled DATA directives
- The `println`'s should become OUTPUT instructions
- The `nextInt`'s should become INPUT instructions
- The assignments to variables should become STORE instructions
- The expressions should become combinations of LOAD instructions, SET instructions, ADD instructions, and SUB instructions (possibly with additional STORE instructions to save intermediate values)
- There should be a STOP instruction at the end
- There should be an initial JUMP to the executable instruction that begins the program (in order to avoid "executing the variables"), which requires that a symbolic address be defined for that executable instruction

Such an initial conversion gives us the partial assembly-language program shown in Table 4.

Table 4: Initial Conversion of Java Program to Assembly Language

Java program	*Equivalent assembly-language program*

`public static void main(String [] args)`
`{`
 	
`    final Scanner in = new Scanner(System.in);`
 	
`    int a, b, larger;`
 	
`    a = in.nextInt();`
`    b = in.nextInt();`
 	
`    if (b > a)`
`        larger = b;`
`    else`
`        larger = a;`
 	
`    System.out.println(larger);`
 	
`    }`
	`    JUMP   main`
	`a:`
	`    DATA   0`
	`b:`
	`    DATA   0`
	`larger:`
	`    DATA   0`
	`main:`
	`    INPUT`
	`    STORE   a`
	`    INPUT`
	`    STORE   b`
	`????`
	`    LOAD   b`
	`    STORE   larger`
	`????`
	`    LOAD   a`
	`    STORE   larger`
	`    LOAD   larger`
	`    OUTPUT`
	`    STOP`

Converting the if-statement to assembly language is more challenging, since there's no immediately-direct equivalent in assembly language.

The only VM252 instructions available to implement conditional statements, loops, and (if one should need to go this far) function calls/returns are the JUMP, JUMPZ, and JUMPP instructions. Happily, a formulaic translation of simple Java control structures into assembly language isn't unreasonably hard.

For if-statements, first translate

`if (`***expr***`)`
    stmt$_1$
`else`
    stmt$_2$
	into an if-statement	`if (`***expr′***` <= 0)`
		    stmt$_1$
		`else`
		    stmt$_2$

where ***expr′*** is an integer expression that produces a non-positive value in exactly those situations where the Boolean expression ***expr*** would produce a `true` value.

The if-statement can then be turned into the assembly language

assembly code to calculate the value of expr′ and place it into the accumulator
```
        JUMPP  else
```
assembly code for stmt$_1$
```
        JUMP   endif
    else:
```
assembly code for stmt$_2$
```
    endif:
```

In some cases, it may instead be preferable or easier to translate

`if (`***expr***`)`
    stmt$_1$
`else`
    stmt$_2$
	into an if-statement	`if (`***expr′***` != 0)`
		    stmt$_1$
		`else`
		    stmt2

where ***expr′*** is an integer expression that produces a non-zero value in exactly those situations where the Boolean expression ***expr*** would produce a `true` value. This can then be turned into the assembly language

assembly code to calculate the value of expr′ and place it into the accumulator
```
        JUMPZ  else
```
assembly code for stmt$_1$
```
        JUMP   endif
    else:
```
assembly code for stmt$_2$
```
    endif:
```

In the current program, this means first translating our Java program into

```
public static void main(String [] args)
{

    final Scanner in = new Scanner(System.in);

    int a, b, larger;

    a = in.nextInt();
    b = in.nextInt();

    if (a - b <= 0)
        larger = b;
```

```
    else
        larger = a;

    System.out.println(larger);

    }
```

which results in the final assembly-language program shown in Table 5.

Table 5: Final Conversion of Java Program to Assembly Language

Java program	*Equivalent assembly-language program*

`public static void main(String [] args)`
`{`
 	
`    final Scanner in = new Scanner(System.in);`
 	
`    int a, b, larger;`
 	
`    a = in.nextInt();`
`    b = in.nextInt();`
 	
`    if (a - b <= 0)`
`        larger = b;`
`    else`
`        larger = a;`
 	
`    System.out.println(larger);`
 	
`    }`
	`    JUMP   main`
	`a:`
	`    DATA   0`
	`b:`
	`    DATA   0`
	`larger:`
	`    DATA   0`
	`main:`
	`    INPUT`
	`    STORE   a`
	`    INPUT`
	`    STORE   b`
	`    LOAD   a`
	`    SUB   b`
	`    JUMPP   else`
	`    LOAD   b`
	`    STORE   larger`
	`    JUMP   endif`
	`else:`
	`    LOAD   a`
	`    STORE   larger`
	`endif:`
	`    LOAD   larger`
	`    OUTPUT`
	`    STOP`

As suggested earlier, this formulaic translation process gave us a correct program, but a notably non-optimal one. A more optimal assembly-language program would be something like

```
    JUMP  main
a:
    DATA  0
b:
    DATA 0
larger:
    DATA  0
main:
    INPUT
    STORE  a
    INPUT
    STORE  b
    SUB  a
    JUMPP  else
    LOAD  a
    JUMP  endif
```

```
else:
    LOAD  b
endif:
    OUTPUT
    STOP
```

For loops, a similar translation scheme is possible. Translate

`while (`***expr***`)`
    stmt

into an equivalent loop

`while (`***expr′*** `<= 0)`
    stmt

and then into

```
while:
     assembly code to calculate the value of expr′ and place it into the accumulator
     JUMPP  endwhile
     assembly code for stmt
     JUMP  while
endwhile:
```

Alternatively, translate

`while (`***expr***`)`
    stmt

into an equivalent loop

`while (`***expr′*** `!= 0)`
    stmt

and then into

```
while:
     assembly code to calculate the value of expr′ and place it into the accumulator
     JUMPZ  endwhile
     assembly code for stmt
     JUMP  while
endwhile:
```

Similar translations are possible for do-loops. For-loops should be translated into equivalent while-loops and those loops then translated into assembly language. Switch-statements should be translated into equivalent if-cascades and those cascades then translated into assembly language.

6. VM252 Software Suite

The suite of VM252-related software is distributed as the Java jar file `VM252.jar` and contains a number of tools.

6.1. The VM252 Assembler

To assemble a file containing a VM252 assembly language program so that it can subsequently be run, execute the command

`java -cp VM252.jar VM252asm` *assemblyLanguageProgramTextFileName*

If one assembled the file `foo.vm252al` with the command

```
java -cp VM252.jar VM252asm foo.vm252al
```

assuming there are no errors in `foo.vm252al`, the file `foo.vm252obj` would then contain the *object code* for the assembled program. If are errors in `foo.vm252al`, messages attempting to describe the errors would appear on the standard error stream and no object file would be produced.

6.2. The VM252 Object-File Dumper

To display a human-readable summary of the contents of a VM252 object file, execute the command

`java -cp VM252.jar VM252dmp` *objectCodeFileName*

If one successfully assembled the file `foo.vm252al` to produce the file `foo.vm252obj`, that object file could be displayed using the command

```
java -cp VM252.jar VM252dmp foo.vm252obj
```

6.3. The VM252 Runner

To execute a file containing VM252 object code, execute the command

`java -cp VM252.jar VM252run` *objectCodeFileName*

If one successfully assembled the file `foo.vm252al` to produce the file `foo.vm252obj`, the program could be run using the command

```
java -cp VM252.jar VM252run foo.vm252obj
```

6.4. The VM252 Debugger

To execute a file containing VM252 object code under the control of a basic debugger, execute the command

`java -cp VM252.jar VM252dbg` *objectCodeFileName*

If one successfully assembled the file `foo.vm252al` to produce the file `foo.vm252obj`, the program could be run under the control of the debugger using the command

```
java -cp VM252.jar VM252dbg foo.vm252obj
```

The debugger provides a number of commands for running and diagnosing errors in programs. When the debugger is running, entering the command `h` will print a summary of the available commands.

6.5. The VM252 Object-File Stripper

To remove all debugging information from a VM252 object-code file (to reduce the size of the object file and/or to hide the details of the source code), execute the command

```
java -cp VM252.jar VM252strip objectCodeFileName
```

A stripped object-code file can still be run and debugged, but note that not all VM252 debugger commands will be available when running a stripped file.

7. Object-Code File Format

The object-code file that results from a successful assembly contains not only the binary encoding of the instructions in the assembled program but also additional information. To simulate execution of the program, one needs to consider only the binary encodings of the instructions; however, the additional information could be used by a debugger (or other software) to provide human-readable information about the program for program-analysis and error-finding purposes.

The object-code file that results from a successful compilation contains, in the order shown,

- 4 bytes holding a 32-bit integer P giving the size, in bytes, of the binary encoding of the *object code* (see below) for the assembled program
- 4 bytes holding a 32-bit integer S giving the size, in bytes, of the binary encoding of the *source-file information* (see below)
- 4 bytes holding a 32-bit integer L giving the size, in bytes, of the binary encoding of the *executable source-line map* (see below)
- 4 bytes holding a 32-bit integer A giving the size, in bytes, of the binary encoding of the *symbolic-address information* (see below)
- 4 bytes holding a 32-bit integer C giving the size, in bytes, of the binary encoding of the *byte-content map* (see below)
- P bytes holding the binary encoding of the instructions in the assembled program
- S bytes holding the binary encoding of the source-file information
- L bytes holding the binary encoding of the executable source-line map
- A bytes holding the binary encoding of the symbolic-address information
- C bytes holding the binary encoding of the byte-content map

for a total object-code file size of $20 + P + S + L + A + C$ bytes.

The VM252 object-file dumper can be used to display this information in readable form (see Section 6 for more information).

7.1. Format of the Object Code

This portion of the object-code file contains the binary encoding the instructions in the assembled program (see Section 3 for details).

7.2. Format of the Source-File information

This portion of the object-code file contains information about the assembly-language file that was assembled to produce this object-code file and contains

- consecutive bytes holding the ASCII characters for the name of the source file that was assembled, followed by a zero byte (i.e., a byte containing 00000000 – the character `'\0'`)
- 8 bytes holding the binary encoding of a long integer representing "last modified" date and time of the source file as of the time that file was assembled to produce the object file (This could be used to check to see if the source file is newer than the object file.)

7.3. Format of the Executable Source-Line Map

This portion of the object-code file holds information about which line of the assembly-language source file that a particular executable instruction came from and consists of pairs of the form

- 4 bytes holding a 32-bit integer giving the number of a line in the source file that contained an executable instruction in the assembly-language program that was assembled
- 4 bytes holding a 32-bit integer giving the address in memory at which the binary encoding of the assembled instruction is located

There will be one such pair for each executable instruction in the assembly-language program.

7.4. Format of the Symbolic-Address Information

This portion of the object-code file hold the names of all the symbolic addresses (often informally called *labels*) defined in the assembly-language program and the address in memory to which that symbolic address corresponds and consists of pairs of the form

- consecutive bytes holding the ASCII characters for the name of the symbolic address, followed by a zero byte (i.e., a byte containing 00000000 – the character `'\0'`)
- 4 bytes holding a 32-bit integer giving the numeric memory address to which that symbolic address corresponds

There will be one such pair for each symbolic address defined in the assembly-language program.

7.5. Format of the Byte-Content Map

This portion of the object-code file holds information telling which bytes of memory hold encodings of instructions from the assembly-language program and which bytes of memory were allocated (via DATA directives) to hold data and consists of consecutive bytes, where the j^{th} byte is

- 00000001, if the j^{th} byte of the object code contains any portion of the binary encoding of an executable instruction from the assembly-language program
- 00000000, if the j^{th} byte of the object code was allocated as a result of a DATA directive in the assembly-language program

In an unstripped object-code file, there will be the same number of bytes in this section of the object-code file as there are in the object-code section.

Appendix C for a sample object-code file, with the contents of the various bytes annotated.

Appendix A: Control-Unit Semantics Modeled as Python Pseudocode

Note that precise modeling is complicated by the fact Python doesn't require or enforce variable or formal-parameter type-declarations. What follows is a best-effort attempt relying on Python type hints together with explanatory prose.

The major components of the VM252 correspond roughly to

```
def ModelOfVM252ControlUnit( \
        ACC : int = an arbitrary integer j, −32768 ≤ j ≤ 32767, \
        PC : int = an arbitrary integer j, 0 ≤ j ≤ 8191, \
        memory : list[int] \
            = [an arbitrary integer j, 0 ≤ j ≤ 255] * 8192 \
        ) -> None :
    ...
```

where `ACC` models the accumulator and `PC` models the program counter of the VM252 control unit.

The behavior of the twelve instructions that comprise the *instruction set* of the VM252 (i.e., the complete repertoire of instructions that the control unit, in concert with the ALU, can carry out can then be described as in Table 6, noting that in each instance in this table, $0 \le a \le 8191$ and $-2048 \le c \le 2047$. (See Appendix B for a brief discussion of *two's complement integer representation*.)

Table 6: VM252 Instruction Semantics

Instruction in symbolic form	*Instruction meaning in Python pseudocode*
`INPUT`	`rawInput = int(input())` `ACC =` *(the least-significant 16 bits of* `rawInput` *treated collectively as a 16-bit two's complement integer)*
`OUTPUT`	`print(ACC)`
`NOOP`	`pass # do nothing`
`STOP`	*halt the processor*
`LOAD` *a*	`ACC =` *(the 16 bits from* `memory[`*a*`]` *and* `memory[(`*a*`+1)%8192]` *treated collectively as a 16-bit two's complement integer)*
`STORE` *a*	*(the 16 bits in* `memory[`*a*`]` *and* `memory[(`*a*`+1)%8192]` *treated collectively as a 16-bit two's complement integer variable)* = ACC
`ADD` *a*	`rawSum = ACC +` *(the 16 bits from* `memory[`*a*`]` *and* `memory[(`*a*`+1)%8192]` *treated as a 16-bit two's complement integer)* `ACC =` *(the least-significant 16 bits of* `rawSum` *treated collectively as a 16-bit two's complement integer)*

Instruction in symbolic form	*Instruction meaning in Python pseudocode*
`SUB a`	`rawDifference = ACC -` *(the 16 bits from* `memory[a]` *and* `memory[(a+1)%8192]` *treated as a 16-bit two's complement integer)* `ACC =` *(the least-significant 16 bits of* `rawDifference` *treated collectively as a 16-bit two's complement integer)*
`JUMP a`	`PC = a`

`JUMPZ a`	`if ACC == 0 :`
	    `PC = a`
	`else :`
	    `PC = (PC + 2) % 8192`

`JUMPP a`	`if ACC > 0 :`
	    `PC = a`
	`else :`
	    `PC = (PC + 2) % 8192`

`SET c`	`ACC =` *(the 12 bits of* `c` *treated as a 12-bit two's complement integer sign-extended to a 16-bit two's complement integer)*

These twelve instructions are sufficient to perform any integer computation that can be carried out on any existing computer.

The program hardwired into the hardware of the control unit corresponds roughly to the pseudocode

```
copy the B bytes of the executable representations of the program instructions into the
    respective elements of memory[0 : B]

PC = 0

opcode = the portion of memory[PC] that distinguishes among the twelve different
    types of instructions (i.e., the "operation code" or "opcode")

while opcode != STOP :

    perform the operation indicated by opcode and (when necessary) the operand formed
     from the relevant portions of memory[PC] and memory[(PC+1) % 8192]

    if opcode in [JUMP, JUMPZ, JUMPP] :
        pass  # do nothing
    elif opcode in [NOOP, INPUT, OUTPUT] :
        PC = (PC + 1) % 8192
    else :
        PC = (PC + 2) % 8192

    opcode = the opcode portion of memory[PC]

halt the processor
```

Appendix B: Integer Representation

The topic of number representation will be covered more thoroughly in lecture. The following is just a brief introduction for purposes of facilitating reading sections of this document.

On the VM252 (as in all modern computing devices), an integer value (whether as a portion of an instruction or as a value stored in memory or in a register) is represented and stored using a fixed number of binary bits.

Unsigned Integers

N-bit unsigned integer representation is used to encode a non-negative integer using exactly n bits. Since only n bits are used, the only integers that can be represented are integers in the range 0 ... 2^n-1 (integers larger than 2^n-1 simply can't be used in such a scheme).

For example, consider 4-bit unsigned integer representation. Only integers in the range 0 ... 15 can be represented, and the integers in the range are represented as

Table 7: 4-Bit Unsigned Integer Representation

Integer	*4-Bit Unsigned Representation*	*Integer*	*4-Bit Unsigned Representation*
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

On the VM252, in certain contexts, integers are represented using 13-bit unsigned integer representation

Table 8: 13-Bit Unsigned Integer Representation

Integer	*13-Bit Unsigned Representation*
0	0000000000000
1	000000000 0001
2	0000000000010
3	0000000000011
4	0000000000100
...	...
8187	1111111111011
8188	1111111111100
8189	1111111111101

Integer	*13-Bit Unsigned Representation*
8190	1111111111110
8191	1111111111111

Signed Integers

N-bit two's complement integer representation is used to encode a signed integer (i.e., an integer that can be negative or non-negative) using exactly *n* bits. Since only *n* bits are used, the only integers that can be represented are integers in the range $-2^{n-1} \ldots 2^{n-1}-1$ (integers smaller than -2^{n-1} or larger than $2^{n-1}-1$ simply can't be used in such a scheme).

For example, consider 4-bit two's complement integer representation. Only integers in the range −8 ... 7 can be represented, and the integers in the range are represented as

Table 9: 4-Bit Two's Complement Integer Representation

Integer	*4-Bit Two's Complement Representation*	*Integer*	*4-Bit Two's Complement Representation*
−8	1000	0	0000
−7	1001	1	0001
−6	1010	2	0010
−5	1011	3	0011
−4	1100	4	0100
−3	1101	5	0101
−2	1110	6	0110
−1	1111	7	0111

On the VM252, in certain contexts, integers are represented using 12-bit two's complement representation

Table 10: 12-Bit Two's Complement Integer Representation

Integer	*12-Bit Two's Complement Representation*
−2048	100000000000
−2047	100000000001
−2046	100000000010
−2045	100000000011
−2044	100000000100
...	...
−4	111111111100
−3	111111111101
−2	111111111110
−1	111111111111

Integer	*12-Bit Two's Complement Representation*
0	000000000000
1	000000000001
2	000000000010
3	000000000011
4	000000000100
...	
2043	011111111011
2044	011111111100
2045	011111111101
2046	011111111110
2047	011111111111

while in other contexts, integers are represented using 16-bit two's complement representation

Table 11: 16-Bit Two's Complement Integer Representation

Integer	*16-Bit Two's Complement Representation*
−32768	1000000000000000
−32767	1000000000000001
−32766	1000000000000010
−32765	1000000000000011
−32764	1000000000000100
...	...
−4	1111111111111100
−3	1111111111111101
−2	1111111111111110
−1	1111111111111111
0	0000000000000000
1	0000000000000001
2	0000000000000010
3	0000000000000011
4	0000000000000100
...	
32763	0111111111111011
32764	0111111111111100
32765	0111111111111101
32766	0111111111111110
32767	0111111111111111

Appendix C: An Annotated Object-Code File Example

Consider the program

```
       JUMP  main
   a:
        DATA  0
   b:
        DATA 0
   larger:
        DATA  0
   main:
        INPUT
        STORE  a
        INPUT
        STORE  b
        SUB  a
        JUMPP  else
        LOAD  a
        JUMP  endif
   else:
        LOAD  b
   endif:
        OUTPUT
        STOP
```

The object file for this program contains the following 251 bytes, in the following order, with the contents of the bytes shown as two hexadecimal digits:

4 bytes collectively holding the 32-bit integer 26, the size, in bytes, of the binary encoding of the object code

```
00
00
00
1a
```

4 bytes collectively holding the 32-bit integer 32, the size, in bytes, of the binary encoding of the source-file name and last-modified date and time at the moment the source file was assembled

```
00
00
00
20
```

4 bytes collectively holding the 32-bit integer 96, the size, in bytes, of the binary encoding of the executable source-line map

```
00
00
00
60
```

4 bytes collectively holding the 32-bit integer 51, the size, in bytes, of the binary encoding of the symbolic-address information

```
00
00
00
33
```

4 bytes holding the 32-bit integer 26, the size, in bytes, of the binary encoding of the byte-content map

```
00
00
00
1a
```

26 bytes holding the binary encoding of the program instructions and the initial values in the bytes allocated via DATA directives

```
80    JUMP   main
08
00    0
00
00    0
00
00    0
00
f0    INPUT
20    STORE  a
02
f0    INPUT
20    STORE  b
04
60    SUB  a
02
c0    JUMPP  else
16
00    LOAD   a
02
80    JUMP   endif
18
00    LOAD   b
04
f4    OUTPUT
fc    STOP
```

32 bytes holding the source-file name and last-modified date and time

```
6c    'l'
61    'a'
72    'r'
67    'g'
65    'e'
72    'r'
4f    'O'
70    'p'
74    't'
69    'i'
6d    'm'
69    'i'
```

7a **'z'**
65 **'e'**
64 **'d'**
2e **'.'**
76 **'v'**
6d **'m'**
32 **'2'**
35 **'5'**
32 **'2'**
61 **'a'**
6c **'l'**
00 **'\0'**
00 ***8 bytes collectively holding an integer representing March 5, 2021 at 10:21:41 AM CST***
00
01
78
03
31
d8
93

96 bytes holding the executable source-line map
00 ***the 32-bit integer 1***
00
00
01
00 ***the 32-bit integer 0, hence the code for source line 1 is at memory address 0***
00
00
00
00 ***the 32-bit integer 9***
00
00
09
00 ***the 32-bit integer 8, hence the code for source line 9 is at memory address 8***
00
00
08
00 ***the 32-bit integer 10***
00
00
0a
00 ***the 32-bit integer 9, hence the code for source line 10 is at memory address 9***
00
00
09

821	`00`	*the 32-bit integer 11*
822	`00`	
823	`00`	
824	`0b`	
825	`00`	*the 32-bit integer 11, hence the code for source line 11 is at memory address 11*
826	`00`	
827	`00`	
828	`0b`	
829	`00`	*the 32-bit integer 12*
830	`00`	
831	`00`	
832	`0c`	
833	`00`	*the 32-bit integer 12, hence the code for source line 12 is at memory address 12*
834	`00`	
835	`00`	
836	`0c`	
837	`00`	*the 32-bit integer 13*
838	`00`	
839	`00`	
840	`0d`	
841	`00`	*the 32-bit integer 14, hence the code for source line 13 is at memory address 14*
842	`00`	
843	`00`	
844	`0e`	
845	`00`	*the 32-bit integer 14*
846	`00`	
847	`00`	
848	`0e`	
849	`00`	*the 32-bit integer 16, hence the code for source line 14 is at memory address 16*
850	`00`	
851	`00`	
852	`10`	
853	`00`	*the 32-bit integer 15*
854	`00`	
855	`00`	
856	`0f`	
857	`00`	*the 32-bit integer 18, hence the code for source line 15 is at memory address 18*
858	`00`	
859	`00`	
860	`12`	
861	`00`	*the 32-bit integer 16*
862	`00`	
863	`00`	
864	`10`	
865	`00`	*the 32-bit integer 20, hence the code for source line 16 is at memory address 20*
866	`00`	
867	`00`	
868	`14`	

`00` *the 32-bit integer 18*
`00`
`00`
`12`
`00` *the 32-bit integer 22, hence the code for source line 18 is at memory address 22*
`00`
`00`
`16`
`00` *the 32-bit integer 20*
`00`
`00`
`14`
`00` *the 32-bit integer 24, hence the code for source line 20 is at memory address 24*
`00`
`00`
`18`
`00` *the 32-bit integer 21*
`00`
`00`
`15`
`00` *the 32-bit integer 25, hence the code for source line 21 is at memory address 25*
`00`
`00`
`19`

<u>51 bytes holding the symbolic-address information</u>

`61` `'a'`
`00` `'\0'`
`00` *the 32-bit integer 2, hence the label* `a` *corresponds to memory address 2*
`00`
`00`
`02`
`62` `'b'`
`00` `'\0'`
`00` *the 32-bit integer 4, hence the label* `b` *corresponds to memory address 4*
`00`
`00`
`04`
`6c` `'l'`
`61` `'a'`
`72` `'r'`
`67` `'g'`
`65` `'e'`
`72` `'r'`
`00` `'\0'`
`00` *the 32-bit integer 6, hence the label* `larger` *corresponds to memory address 6*
`00`
`00`
`06`

`6d`	`'m'`
`61`	`'a'`
`69`	`'i'`
`6e`	`'n'`
`00`	`'\0'`
`00`	***the 32-bit integer 8, hence the label `main` corresponds to memory address 8***
`00`	
`00`	
`08`	
`65`	`'e'`
`6c`	`'l'`
`73`	`'s'`
`65`	`'e'`
`00`	`'\0'`
`00`	***the 32-bit integer 22, hence the label `else` corresponds to memory address 22***
`00`	
`00`	
`16`	
`65`	`'e'`
`6e`	`'n'`
`64`	`'d'`
`69`	`'i'`
`66`	`'f'`
`00`	`'\0'`
`00`	***the 32-bit integer 24, hence the label `endif` corresponds to memory address 24***
`00`	
`00`	
`18`	

26 bytes holding the byte-content map

01	*the corresponding byte of the object code holds executable code*
01	*the corresponding byte of the object code holds executable code*
00	*the corresponding byte of the object code holds data*
00	*the corresponding byte of the object code holds data*
00	*the corresponding byte of the object code holds data*
00	*the corresponding byte of the object code holds data*
00	*the corresponding byte of the object code holds data*
00	*the corresponding byte of the object code holds data*
01	*the corresponding byte of the object code holds executable code*
01	*the corresponding byte of the object code holds executable code*
01	*the corresponding byte of the object code holds executable code*
01	*the corresponding byte of the object code holds executable code*
01	*the corresponding byte of the object code holds executable code*
01	*the corresponding byte of the object code holds executable code*
01	*the corresponding byte of the object code holds executable code*
01	*the corresponding byte of the object code holds executable code*
01	*the corresponding byte of the object code holds executable code*
01	*the corresponding byte of the object code holds executable code*
01	*the corresponding byte of the object code holds executable code*
01	*the corresponding byte of the object code holds executable code*

968	01	*the corresponding byte of the object code holds executable code*
969	01	*the corresponding byte of the object code holds executable code*
970	01	*the corresponding byte of the object code holds executable code*
971	01	*the corresponding byte of the object code holds executable code*
972	01	*the corresponding byte of the object code holds executable code*
973	01	*the corresponding byte of the object code holds executable code*