

This lab has **three (3)** checkpoints.

Learning Outcomes

Upon completing this lab, students should be able to

- Modify their existing BitString class to make the character array a protected data field.
- extend BitString into an Instruction class as per the Hack machine code specification (reproduced here).
- Be able to use Gradle command-line --args to call the (provided) driver program with different values for checking.

Introduction

A-Instruction												
symbolic	xxx	xxx is decimal or a symbol. $0 \leq xxx \leq 32767$										
binary	0vvvvvvvvvvvvvvvv	vvvvvvvvvvvvvvvv binary for xxx										
C-Instruction												
symbolic	dest=comp;jmp		dest= optional destination comp required computation ;jmp optional jump									
binary	111acccccdddjjj											
comp		c	c	c	c	c	c	dst	d	d	d	comp⇒
0		1	0	1	0	1	0	null	0	0	0	
1		1	1	1	1	1	1	M	0	0	1	RAM[A]
-1		1	1	1	0	1	0	D	0	1	0	D
D		0	0	1	1	0	0	DM	0	1	1	D, RAM[A]
A	M	1	1	0	0	0	0	A	1	0	0	A
!D		0	0	1	1	0	1	AM	1	0	1	A, RAM[A]
!A	!M	1	1	0	0	0	1	AD	1	1	0	A, D
-D		0	0	1	1	1	1	ADM	1	1	1	A, D, RAM[A]
-A	-M	1	1	0	0	1	1	jmp	j	j	j	jump if
D+1		0	1	1	1	1	1	null	0	0	0	never
A+1	M+1	1	1	0	1	1	1	JGT	0	0	1	comp> 0
D-1		0	0	1	1	1	0	JEQ	0	1	0	comp= 0
A-1	M-1	1	1	0	0	1	0	JGE	0	1	1	comp≥ 0
D+A	D+M	0	0	0	0	1	0	JLT	1	0	0	comp< 0
D-A	D-M	0	1	0	0	1	1	JNE	1	0	1	comp≠ 0
A-D	M-D	0	0	0	1	1	1	JLE	1	1	0	comp≤ 0
D&A	D&M	0	0	0	0	0	0	JMP	1	1	1	always
D A	D M	0	1	0	1	0	1					
a=0	a=1	zx	nx	zy	ny	f	no					

1. The above tables are based on those in the Nand2Tetris book. You will use them throughout this assignment.

(a) How would you *encode* the following instructions from Hack assembly to the machine code:

i. DM = D&A;JGT

Example:

DM = 011

D&A = 0 000000

JGT = 001

1110000000011001 or 1110 0000 0001 1001

ii. @544

Solution: 0000 0010 0010 0000

iii. D=M+1

Solution: 1111 1101 1101 0000

iv. A;JLE

Solution: 1110 1100 0000 0110

v. D-A;JNE

Solution: 1110 0100 1100 0101(b) And what about *decoding*?

i. 0000 0000 0001 0000

Example:

A-Instruction: @16

ii. 1111 1100 0001 0000

Solution: C-Instruction: D=M

iii. 1111 0100 1101 0000

Solution: C-Instruction: D=D-M

iv. 1110 0011 0000 0001

Solution: C-Instruction: D;JGT

v. 1110 0000 0000 0111

Solution: C-Instruction: D&A;JMP

vi. 0000 0000 1111 0000

Solution: A-Instruction: @240

✓ Show and explain your answers to these questions to the lab instructor.

2. Initialize a Gradle application project in a new directory. Also initialize git in the root of the project.

The name of the project: instruction.

Starting package: client.

Copy `InstructionDriver.java` into `client` and make it the default executable class.

Add a package, `bits`. Copy your `BitString.java` into `bits`. It is assumed for this program that you have the whole interface (set of public functions) implemented for `BitString`.

Add a new constructor to `BitString` that takes a `String` of sixteen zero and one characters, representing a bit string from bit 15 down to 0 (0000000011110000 is the binary representation of 240 decimal).

Also change the access level of your array of characters to `protected` (it **should** be `private`).

Note: Your code will not yet compile.

✓ Show the lab instructor

- Your new constructor, `BitString(String)` and explain how it works.
- Your changed access to your `char []` field in `BitString`.

3. Time to add the `Instruction` class. An `Instruction` is a `BitString` (that says **inheritance** in big flashing letters) that has some extra capabilities.

`Instruction` will have the following public functions:

```
class Instruction
    extends BitString {
    // constructors should forward all work to the super class
    public Instruction();
    public Instruction(short s);
    public Instruction(Instruction other);
    public Instruction(String allMyBits);

    // kind of instruction
    public boolean AInstruction();
    public boolean CInstruction();

    // comp
    public boolean a();
    public boolean zx();
    public boolean nx();
    public boolean zy();
    public boolean ny();
    public boolean f();
    public boolean no();

    // dest
```

```

    public boolean A();
    public boolean D();
    public boolean M();

    // jump - Relate to lowest order 3 bits
    public boolean JLesser();
    public boolean JEqual();
    public boolean JGreater();
}

```

When it compiles, you can run the client to check it out. The following are argument/output pairs:

```
1111010011110101 0000000001100100 0000000000000001
```

```

ins: 1111 0100 1111 0101
x:   0000 0000 0110 0100(100)
y:   0000 0000 0000 0001(1)
Addressing Memory, not Register A
out: 0000 0000 0110 0011(99)
Storing to: AD
Jump on: <>

```

```
1110000000000000 0000000001100100 0000000000000001
```

```

ins: 1110 0000 0000 0000
x:   0000 0000 0110 0100(100)
y:   0000 0000 0000 0001(1)
out: 0000 0000 0000 0000(0)
Storing to:
Jump on:

```

```
0111111111111111 0000000001100100 0000000000000001
```

```

ins: 0111 1111 1111 1111
x:   0000 0000 0110 0100(100)
y:   0000 0000 0000 0001(1)
An A instruction
@32767

```

✓ Show the lab instructor your new class:

- That your Instruction constructors pass all work to BitString.
- Properly runs some of the commands from the trace portion of the lab.
(If your BitString is not yet complete, that is okay, just show the class to the instructor.)

4. [Not a separate checkpoint but necessary] Add and commit your full gradle project (if you `git add` just the root directory, `.,` git will add all subdirectories recursively).

Create a repository in your Gitea turn-in organization, associate it with your lab work, and push the repo. This will be done at the end of every lab.