Name:______________________
Date: ______________________

The A-instruction: symbolic and binary syntax

Semantics: Set the A register to *value*

Symbolic syntax:

```
@value
```

Where *value* is either:

- a non-negative decimal constant ≤ 32767 ($=2^{15}-1$) or
- a symbol referring to such a constant (later)

Example:

```
@21
```

Effect: sets the A register to 21

Binary syntax: 0vvvvvvvvvvvvvvv
For example: @21 **(symbolic syntax – assembly code)**
0000000000010101 **(binary syntax – machine code)**

The C-instruction: symbolic and binary syntax

Symbolic syntax: *dest* = *comp* ; *jump*

Binary syntax: `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`

comp		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a=0	a=1						

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Name:______________________
Date: ______________________

Lab #6 – Computer Architecture

Recall the two Assembly Instructions, A and C:

The A-instruction

Syntax: `@value`

Where *value* is either:
- a non-negative decimal constant or
- a symbol referring to such a constant (later)

Semantics:
- Sets the A register to *value*
- Side effect: RAM[A] becomes the selected RAM register

Example: `@21`

Effect:
- Sets the A register to 21
- RAM[21] becomes the selected RAM register

The C-instruction

dest = *comp* ; *jump* (both *dest* and *jump* are optional)

where:

comp = `0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D&A, D|A`
`M, !M, -M, M+1, M-1, D+M, D-M, M-D, D&M, D|M`

dest = `null, M, D, MD, A, AM, AD, AMD` M refers to RAM[A]

jump = `null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP` if (*comp jump* 0) jump to execute the instruction in ROM[A]

Semantics:
- Compute the value of *comp*
- Stores the result in *dest*;
- If the Boolean expression (*comp jump* 0) is true, jumps to execute the instruction stored in ROM[A].

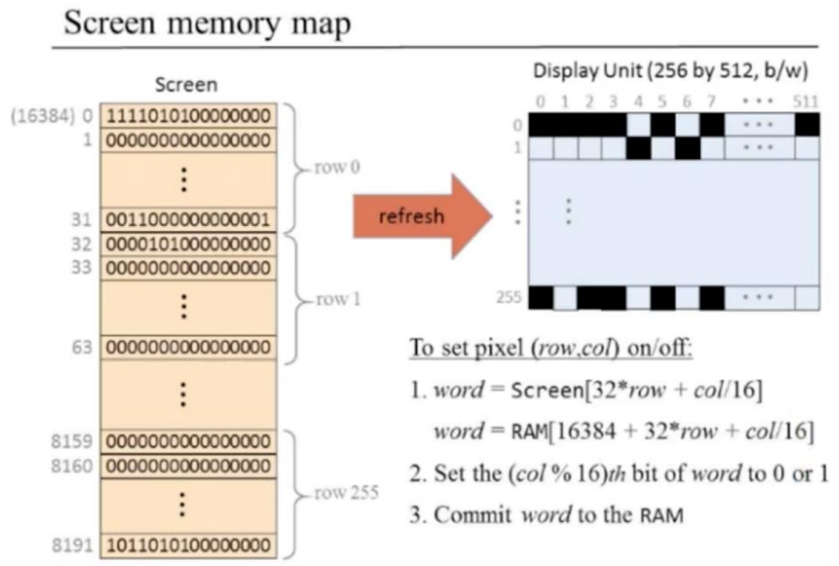
1) Write an assembly program to perform the multiplication of two integers (stored at R0 and R1) and store the result in R2. You can perform multiplication through successive addition. For example:

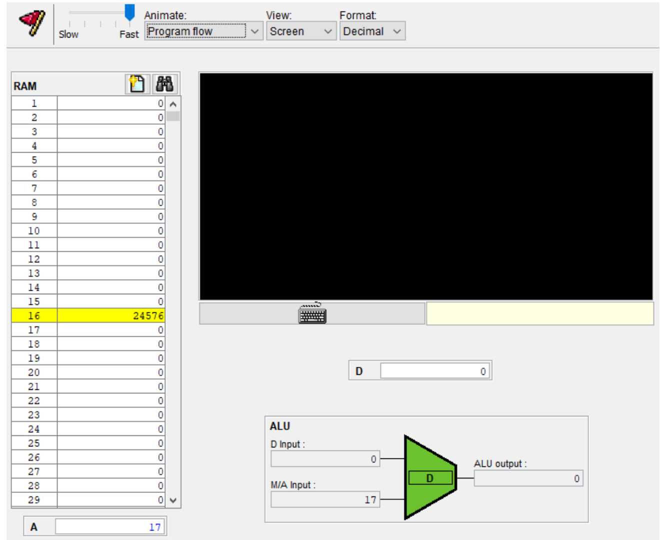
5 * 4 = (5 + 5 + 5 + 5) = 20

a. First write Java code using a for loop to perform the successive addition. Assume the following:
int R0 = 5;
int R1 = 4;
int R2; // Stores the result of R0 * R1 (using successive addition in a for loop)

b. Next, convert the high-level Java code into assembly code:

2) Given the following screen memory map:





Write the assembly code that will loop through all the screen locations (in RAM) and darken them to produce the screen shot above:

Directions: Complete the translation from Assembly to Machine code for the following instructions.

1) @1

parts																
values																

2) @SCREEN //@________

parts																
values																

3) @KBD //@________

parts																
values																

4) @R5 //@________

parts																
values																

5) A=-1

parts																
values																

6) A=D

parts																
values																

7) A=M

parts																
values																

8) A=D&M

parts																
values																

9) D=A

parts																	
values																	

10) D=M

parts																
values																

11) D=D|A

parts																
values																

12) D=!D

parts																
values																

13) M=A

parts																
values																

14) M=D

parts																
values																

15) M=D-1

parts																
values																

16) M=D+A

parts																
values																

17) MD=M-D

parts																
values																

18) AD=M+1

parts																
values																

19) AMD=!M

parts																
values																

20) 0;JMP //0 is _______

parts																
values																

21) D;JGT

parts																
values																

22) D;JEQ

parts																
values																

23) D;JLE

parts																
values																

24) D;JNE

parts																
values																

Lab #6 Computer Architecture - Memory

Name:______________________

Section/Time:______________________

Date: ______________________

❑ Write in the decimal values for each address below, and convert to binary:

NO CALCULATOR! Use back as scratch paper

RAM16K minimum address (first address) = ($)_{10}$														
[14]	[13]	[12]	[11]	[10]	[9]	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]

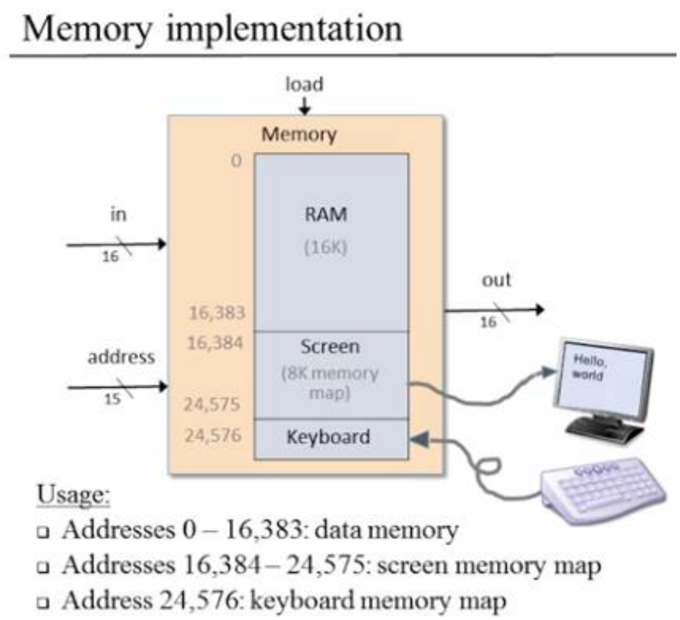
RAM16K maximum address (last address) = ($)_{10}$														
[14]	[13]	[12]	[11]	[10]	[9]	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]

SCREEN Pointer address (start of screen memory map) = ($)_{10}$														
[14]	[13]	[12]	[11]	[10]	[9]	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]

Last possible address for screen memory map = ($)_{10}$														
[14]	[13]	[12]	[11]	[10]	[9]	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]

KBD Pointer address = ($)_{10}$														
[14]	[13]	[12]	[11]	[10]	[9]	[8]	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]

Lab #6 Computer Architecture - Memory



- ❑ When interacting with the Memory chip, how can it differentiate between an address for the screen memory map, the keyboard register, and RAM16?

- ❑ What are the bus sizes for each chip in Memory.hdl?

- ❑ What does the **`load`** pin accomplish?

- ❑ How is it used for the Memory chip?
  - ❑ Play through scenarios for the value of load and what SHOULD happen to each part of memory. How can the chip differentiate where load should go?