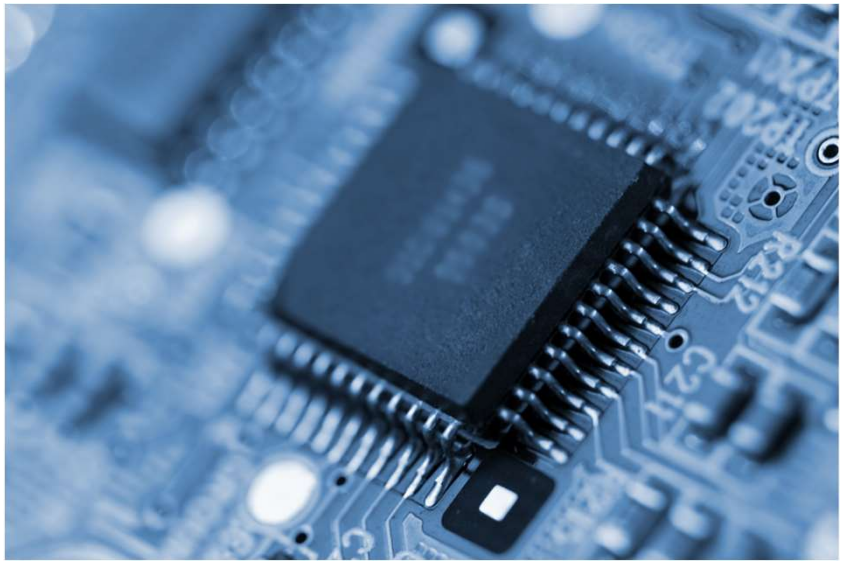
Computer Processors

Assembly Language



Chapter 4: Machine Language

Overview

- ✓ Machine languages
- ✓ The Hack computer
- → The Hack instruction set
- The Hack CPU Emulator

Low Level Programming

- Basic
- Iteration
- Pointers

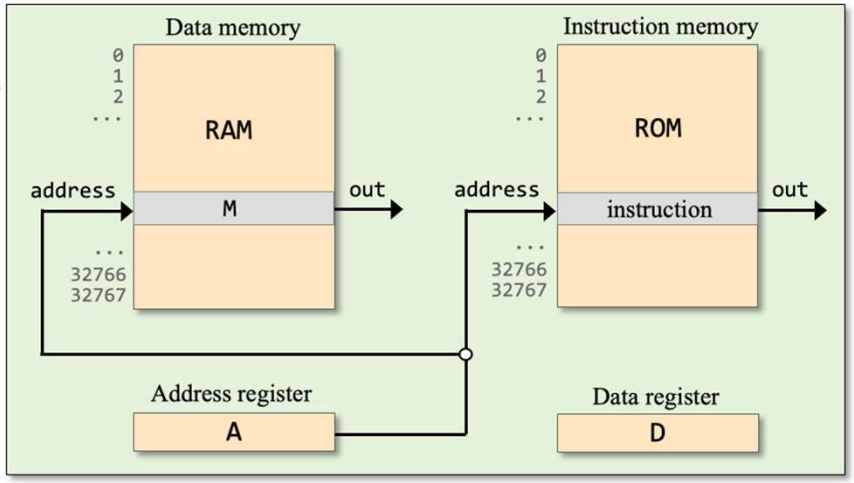
Symbolic programming

- Control
- Variables
- Labels

The Hack Language

- Usage
- Specification
- Output
- Input
- Project 4

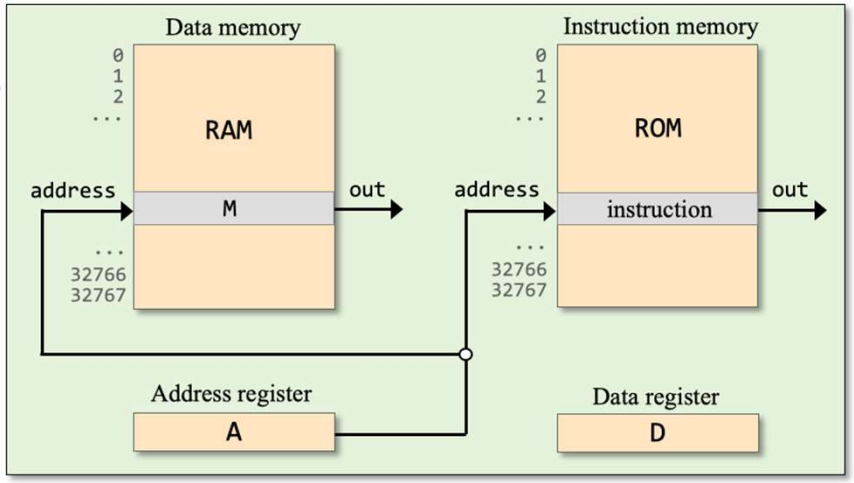
Hack instructions



Instruction set

- `A` instruction
- `C` instruction

Hack instructions



Instruction set

- A instruction
- C instruction

Syntax:

`@` *const*

where *const* is a constant

(Complete / formal syntax, later).

Example:

```
@19
```

Semantics:

A ← 19

Side effects:

- `RAM[A]` (called `M`) becomes selected
- `ROM[A]` becomes selected

Hack instructions

Instruction set

- A instruction
- C instruction

where $reg, reg_1 = \{A|D|M\}$, $op = \{+|-\}$, and
$reg_2 = \{A|D|M|1\}$ and $reg_1 \neq reg_2$

Syntax:

$reg = \{0|1|-1\}$

where $reg = \{A|D|M\}$

$reg_1 = reg_2$

where $reg_1 = \{A|D|M\}$
$reg_2 = [-]\{A|D|M\}$

$reg = reg_1\ op\ reg_2$

where $reg, reg_1 = \{A|D|M\}$, $op = \{+|-\}$, and
$reg_2 = \{A|D|M|1\}$ and $reg_1 \neq reg_2$

Examples:

```
D=0
A=-1
M=1
...
```

```
D=A
D=M
M=-M
...
```

```
D=D+M
A=A-1
M=D+1
...
```

(Complete / formal syntax, later).

Hack instructions

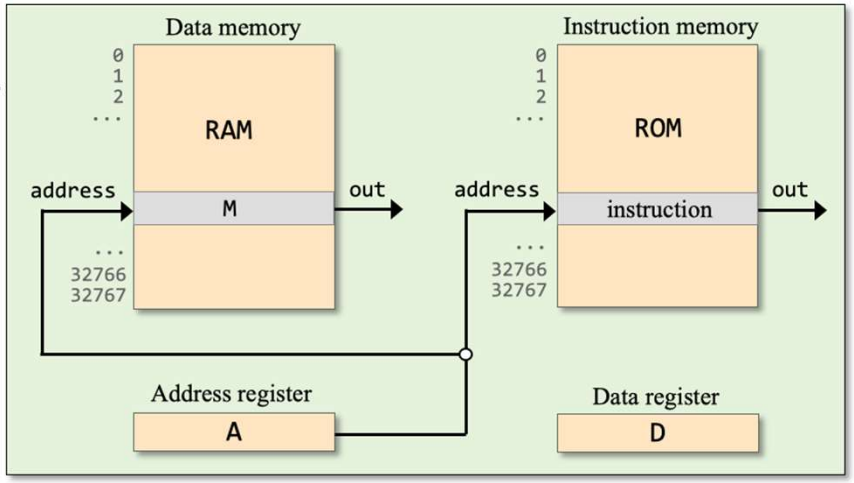
Typical instructions:

@*constant* (A←*constant*)

```
D=1
D=A
D=D+1
...
```

```
D=D+A
D=M
M=0
...
```

```
M=D
D=D+A
M=M-D
...
```



Examples:

```
// D ← 2
?
```

The game: We show some typical Hack instructions (top left), and practice writing code examples that use subsets of these instructions.

Hack instructions

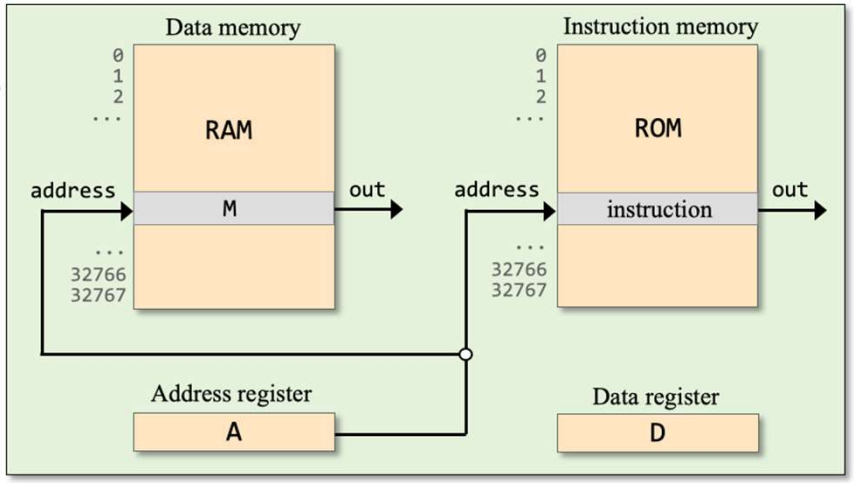
Typical instructions:

@ *constant* (A ← *constant*)

```
D=1
D=A
D=D+1
...
```

```
D=D+A
D=M
M=0
...
```

```
M=D
D=D+A
M=M-D
...
```



Examples:

```
// D ← 2
D=1
D=D+1
```

```
// D ← 1954
?
```

Use only the instructions shown in this slide

Hack instructions

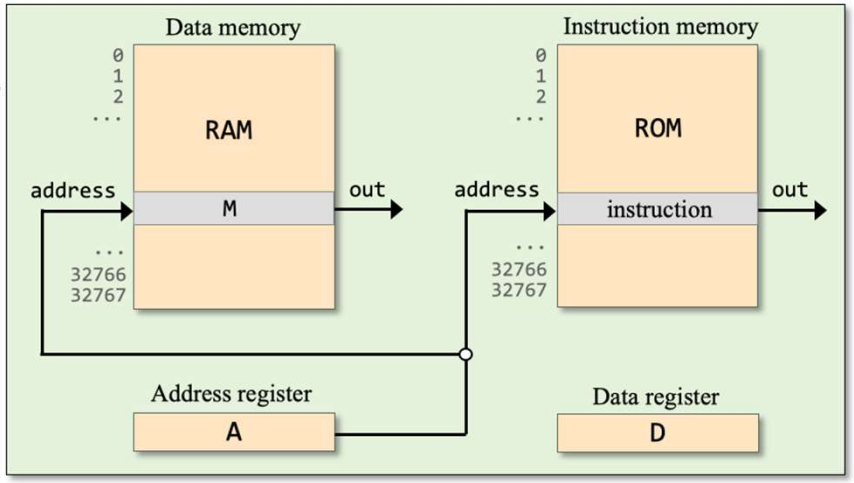
Typical instructions:

`@`*constant* (A←*constant*)

```
D=1
D=A
D=D+1
...
```

```
D=D+A
D=M
M=0
...
```

```
M=D
D=D+A
M=M-D
...
```



Examples:

```
// D ← 2
D=1
D=D+1
```

```
// D ← 1954
@1954
D=A
```

```
// D ← D + 23
?
```

Use only the instructions shown in thus slide

Hack instructions

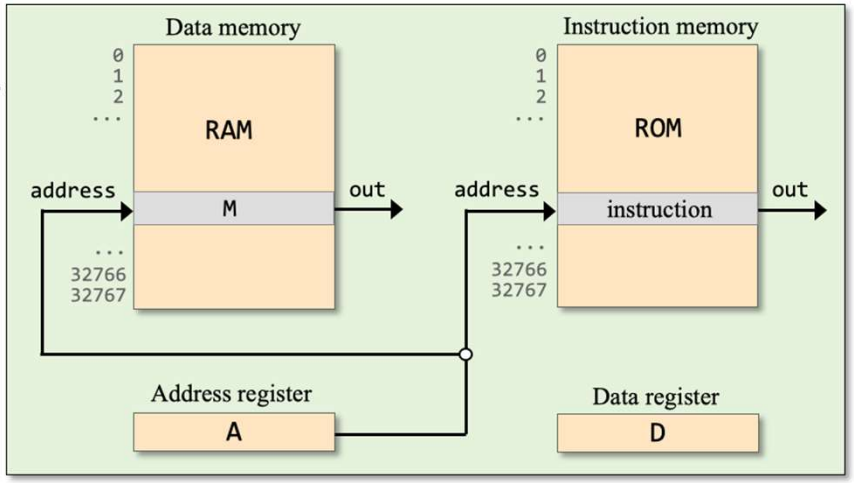
Typical instructions:

@ *constant* (A ← *constant*)

```
D=1
D=A
D=D+1
...
```

```
D=D+A
D=M
M=0
...
```

```
M=D
D=D+A
M=M-D
...
```



Examples:

```
// D ← 2
D=1
D=D+1
```

```
// D ← 1954
@1954
D=A
```

```
// D ← D + 23
@23
D=D+A
```

Observation

In these examples we use the address register `A` as a *data register*:

The addressing side-effects of `A` are ignored.

Hack instructions

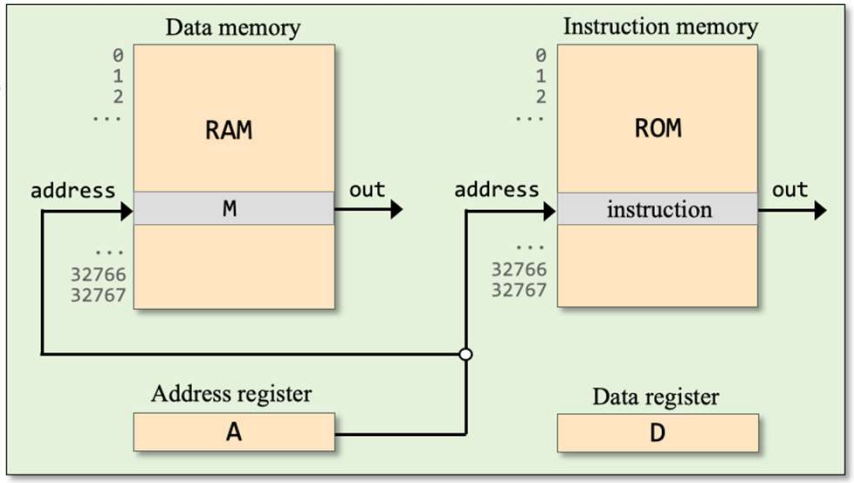
Typical instructions:

`@`*constant* (A ← *constant*)

```
D=1
D=A
D=D+1
...
```

```
D=D+A
D=M
M=0
...
```

```
M=D
D=D+A
M=M-D
...
```



More examples:

```
// RAM[100] ← 0
```

Hack instructions

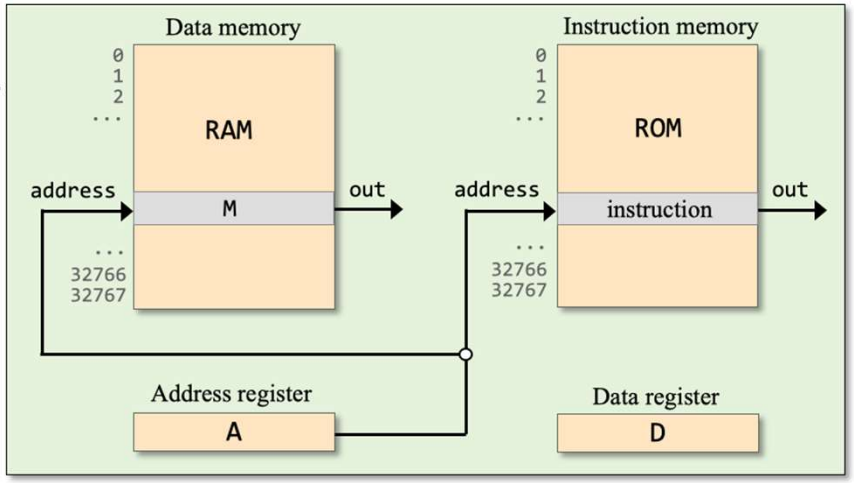
Typical instructions:

@ *constant* (A ← *constant*)

```
D=1
D=A
D=D+1
...
```

```
D=D+A
D=M
M=0
...
```

```
M=D
D=D+A
M=M-D
...
```



More examples:

```
// RAM[100] ← 0
@100
M=0
```

```
// RAM[100] ← 17
```

Hack instructions

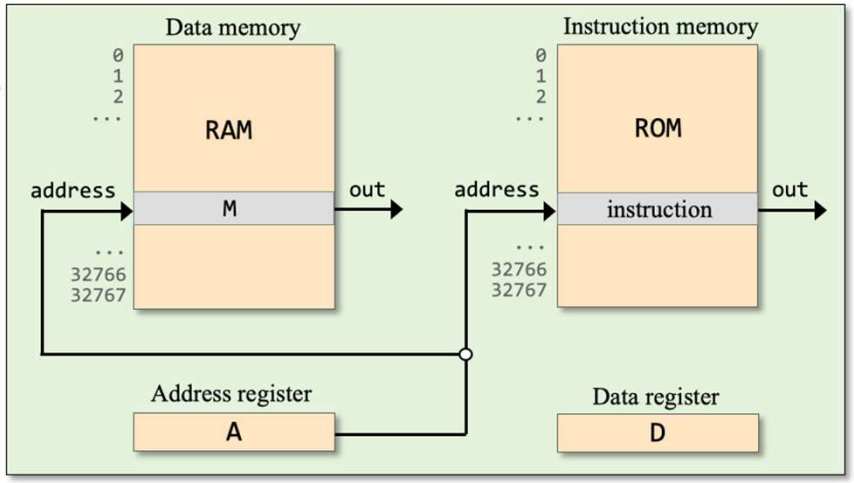
Typical instructions:

`@`*constant* (A ← *constant*)

```
D=1
D=A
D=D+1
...
```

```
D=D+A
D=M
M=0
...
```

```
M=D
D=D+A
M=M-D
...
```



More examples:

```
// RAM[100] ← 0
@100
M=0
```

```
// RAM[100] ← 17
@17
D=A
@100
M=D
```

- First pair of instructions:
  A is used as a *data register*
- Second pair of instructions:
  A is used as an *address register*

Hack instructions

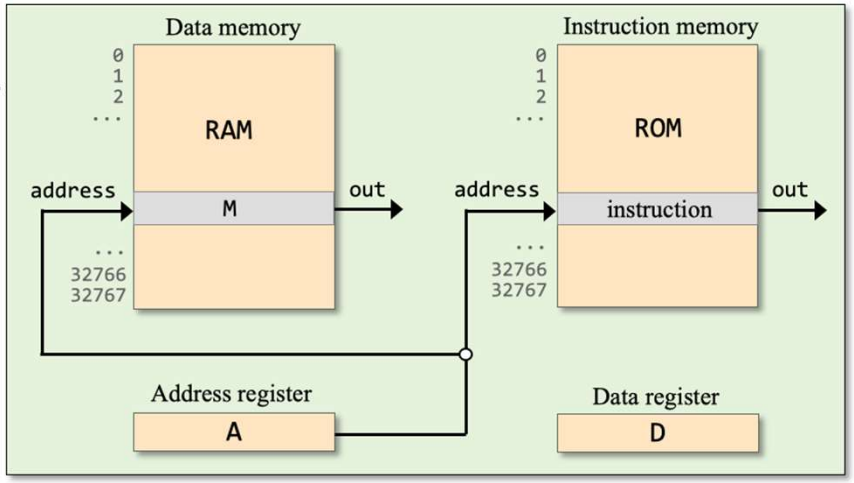
Typical instructions:

`@constant` (A ← *constant*)

```
D=1
D=A
D=D+1
...
```

```
D=D+A
D=M
M=0
...
```

```
M=D
D=D+A
M=M-D
...
```



More examples:

```
// RAM[100] ← 0
@100
M=0
```

```
// RAM[100] ← 17
@17
D=A
@100
M=D
```

```
// RAM[100] ← RAM[200]
```

?

Hack instructions

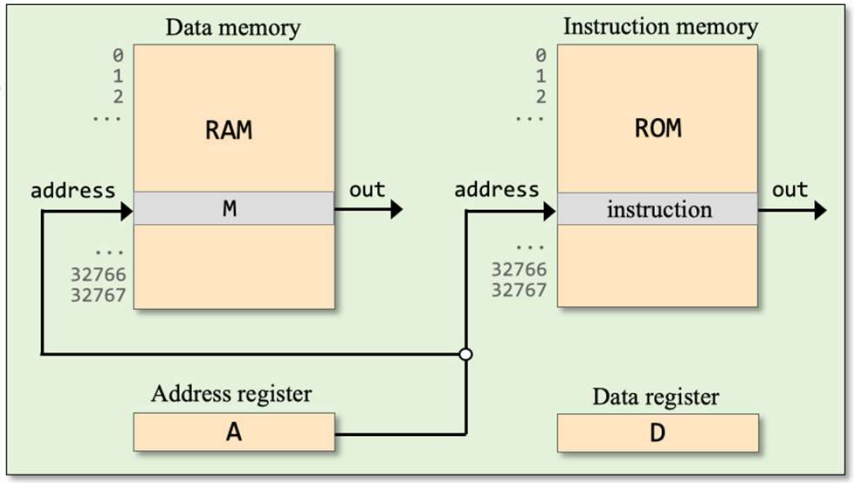
Typical instructions:

`@constant` (A ← *constant*)

```
D=1
D=A
D=D+1
...
```

```
D=D+A
D=M
M=0
...
```

```
M=D
D=D+A
M=M-D
...
```



More examples:

```
// RAM[100] ← 0
@100
M=0
```

```
// RAM[100] ← 17
@17
D=A
@100
M=D
```

```
// RAM[100] ← RAM[200]
@200
D=M
@100
M=D
```

When we want to operate on a memory register, we typically need a pair of instructions:

- A instruction: Selects a memory register
- C instruction: Operates on the selected register.

Hack instructions

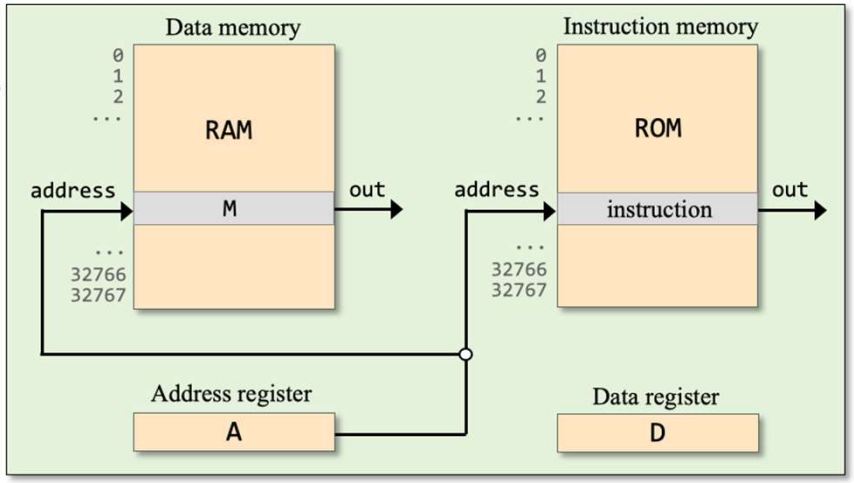
Typical instructions:

@ *constant* (A ← *constant*)

```
D=1
D=A
D=D+1
...
```

```
D=D+A
D=M
M=0
...
```

```
M=D
D=D+A
M=M-D
...
```




```
// RAM[3] ← RAM[3] - 15
```

?

Use only the instructions shown in the current slide

Hack instructions

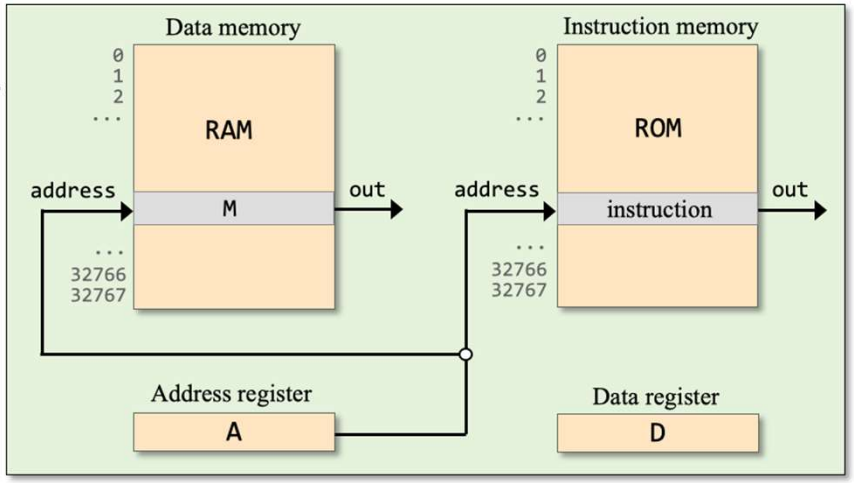
Typical instructions:

@ *constant* (A ← *constant*)

```
D=1
D=A
D=D+1
...
```

```
D=D+A
D=M
M=0
...
```

```
M=D
D=D+A
M=M-D
...
```




```
// RAM[3] ← RAM[3] - 15
@15
D=A
@3
M=M-D
```

Use only the instructions shown in the current slide

```
// RAM[3] ← RAM[4] + 1
```

?

Hack instructions

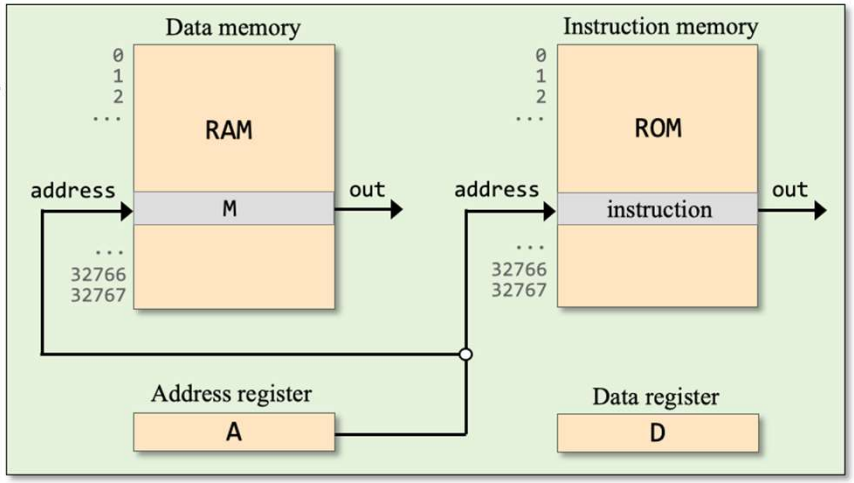
Typical instructions:

@ *constant* (A ← *constant*)

```
D=1
D=A
D=D+1
...
```

```
D=D+A
D=M
M=0
...
```

```
M=D
D=D+A
M=M-D
...
```




```
// RAM[3] ← RAM[3] - 15
@15
D=A
@3
M=M-D
```

Use only the instructions shown in the current slide

```
// RAM[3] ← RAM[4] + 1
@4
D=M+1
@3
M=D
```

Hack instructions

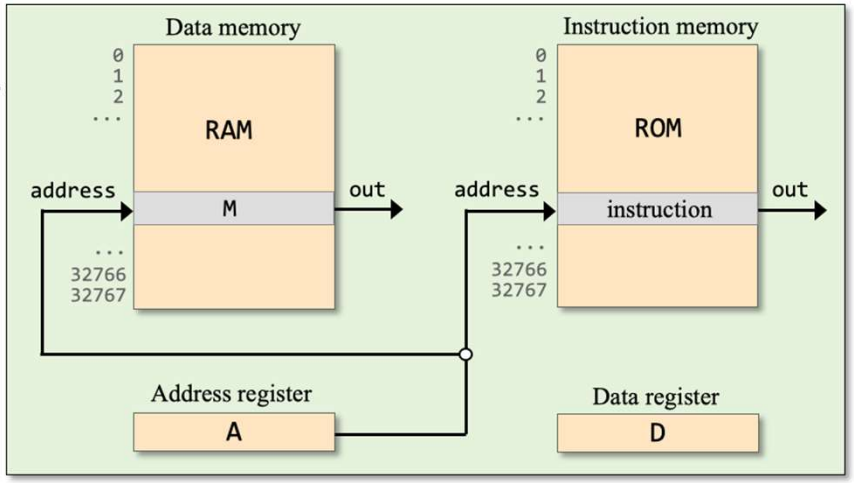
Typical instructions:

`@`*constant* (A←*constant*)

```
A=1
D=-1
M=0
...
```

```
A=M
D=M
M=D
...
```

```
A=D-A
D=D+A
D=D+M
...
```



Add.asm

```
// Computes: RAM[2] = RAM[0] + RAM[1] + 17

?
```

Use only the instructions shown in the current slide

Hack instructions

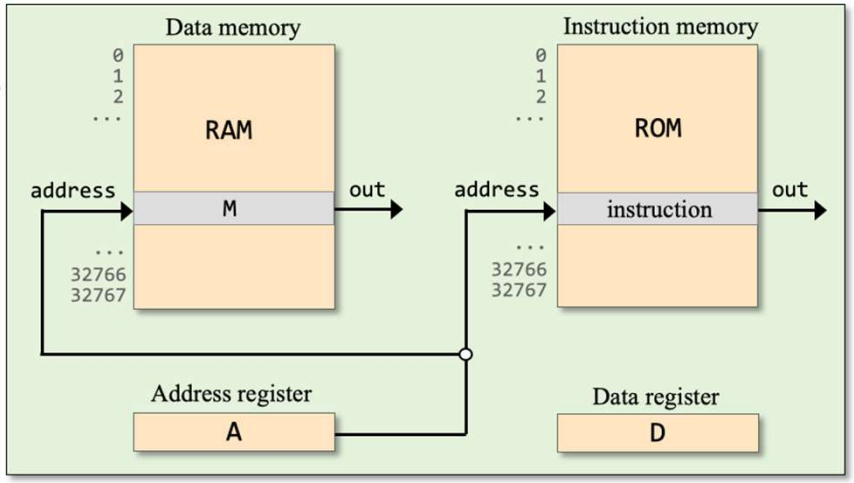
Typical instructions:

`@`*constant* (A←*constant*)

```
A=1
D=-1
M=0
...
```

```
A=M
D=M
M=D
...
```

```
A=D-A
D=D+A
D=D+M
...
```



Add.asm

```
// Computes: RAM[2] = RAM[0] + RAM[1] + 17
// D = RAM[0]
@0
D=M
// D = D + RAM[1]
@1
D=D+M
// D = D + 17
@17
D=D+A
// RAM[2] = D
@2
M=D
```

Hack instructions

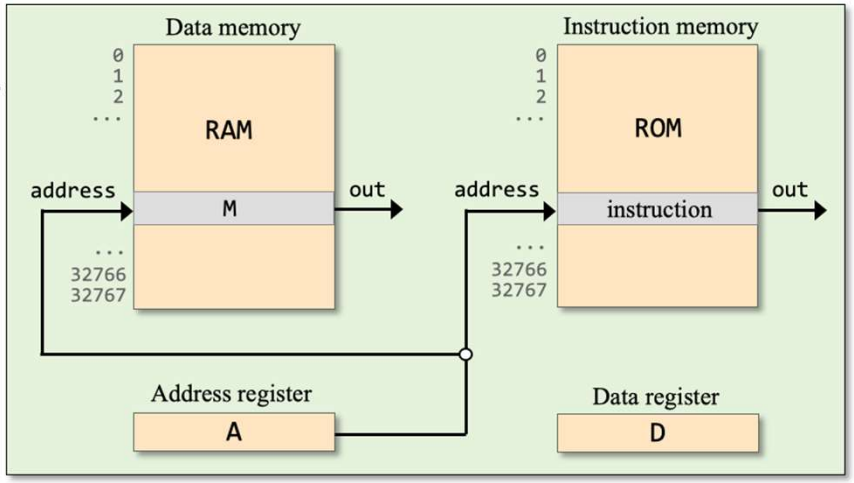
Typical instructions:

`@constant` (A←*constant*)

```
A=1
D=-1
M=0
...
```

```
A=M
D=M
M=D
...
```

```
A=D-A
D=D+A
D=D+M
...
```



Add.asm

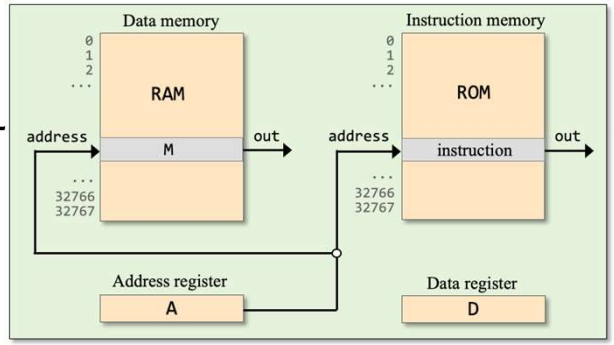
```
// Computes: RAM[2] = RAM[0] + RAM[1] + 17
// D = RAM[0]
@0
D=M
// D = D + RAM[1]
@1
D=D+M
// D = D + 17
@17
D=D+A
// RAM[2] = D
@2
M=D
```

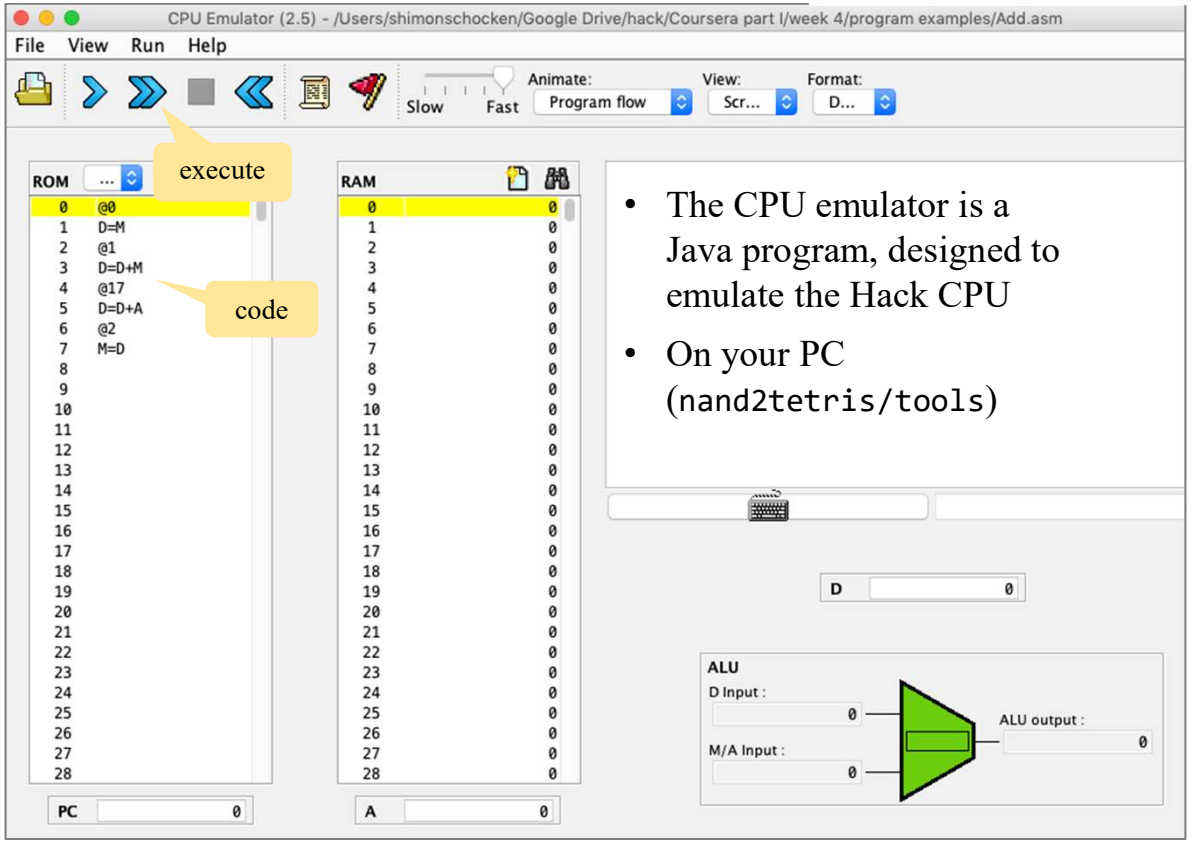
How can we tell that a given program *actually works*?

➡ Testing / simulating

- Formal verification

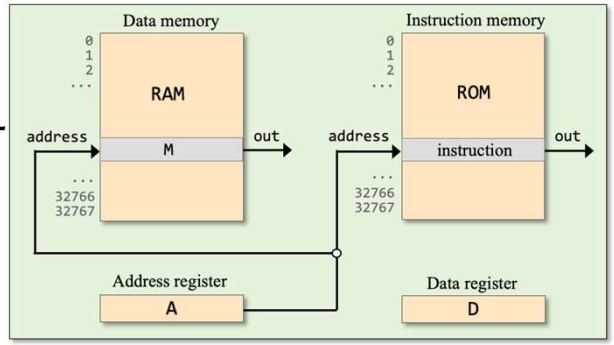
The CPU emulator





- The CPU emulator is a Java program, designed to emulate the Hack CPU
- On your PC (`nand2tetris/tools`)

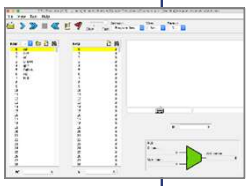
The CPU emulator



Add.asm (example)

```
// Computes: RAM[2] = RAM[0] + RAM[1] + 17
// D = RAM[0]
@0
D=M
// D = D + RAM[1]
@1
D=D+M
// D = D + 17
@17
D=D+A
// RAM[2] = D
@2
M=D
```

Load into the CPU emulator



Binary

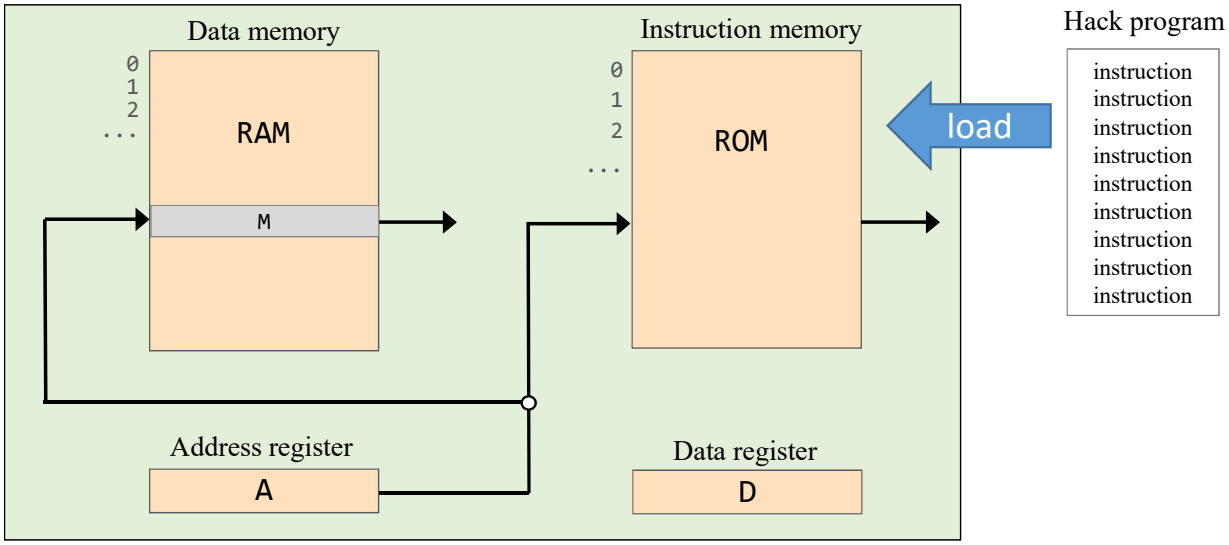
```
0000000000000000
1000010010001101
0000000000000001
1010011001100001
0000000000010001
1001111100110011
0000000000000010
1110010010010011
```

Execute

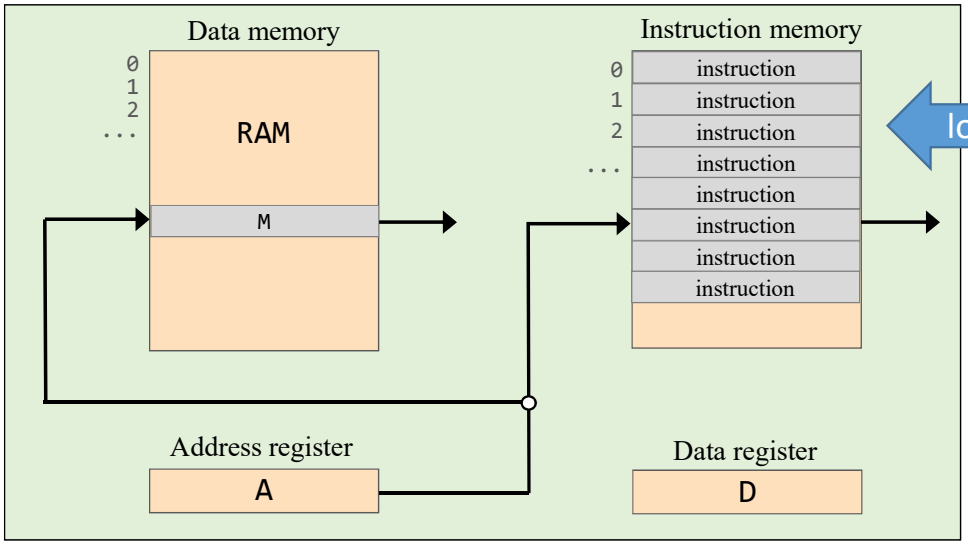
When loading a symbolic program into the CPU emulator, the emulator translates it into binary code (using a built-in assembler)

The CPU emulator

Loading a program



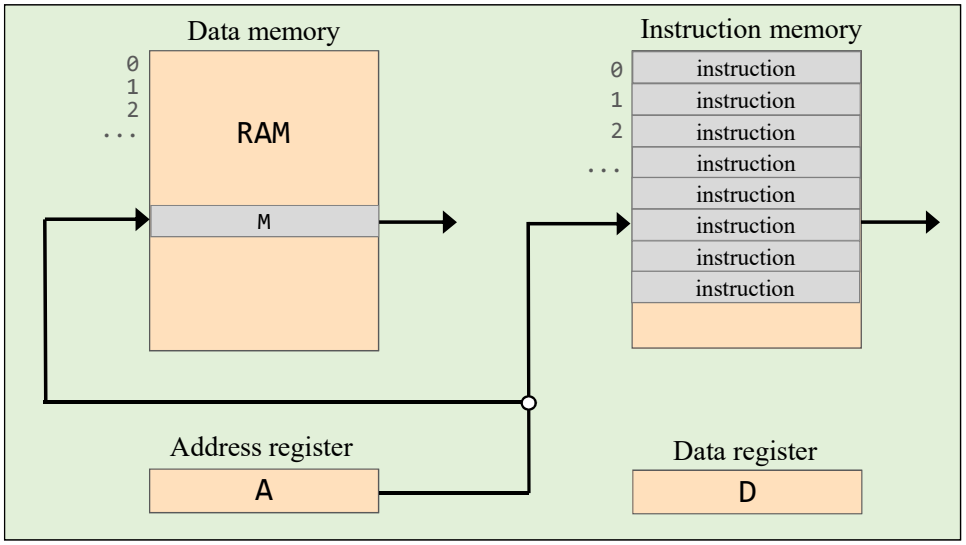
Loading a program



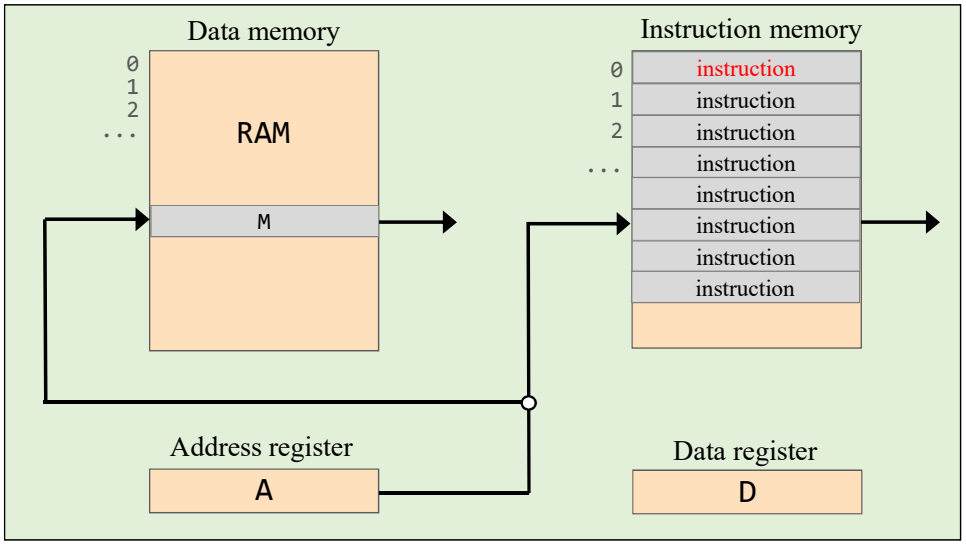
Convention:

The first instruction is loaded into address 0, the next instruction into address 1, and so on.

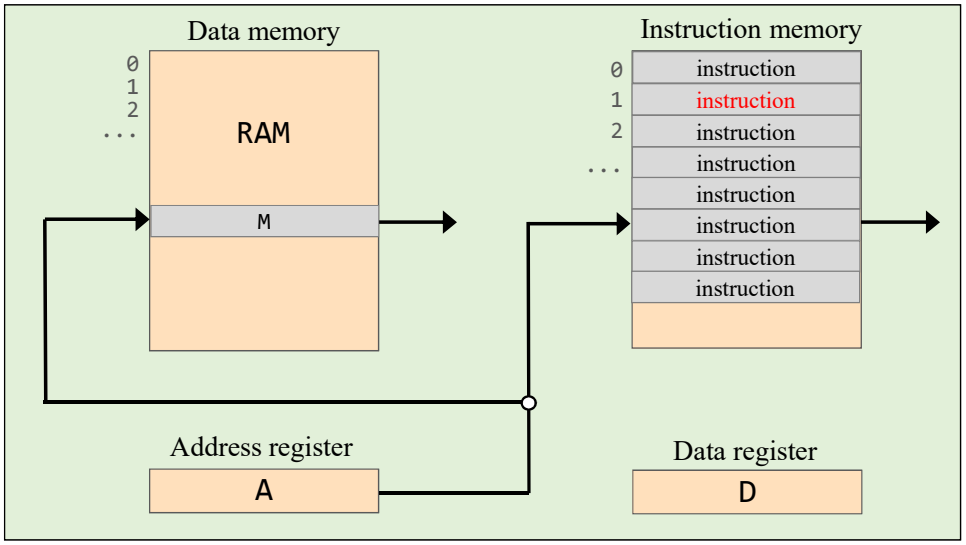
Executing a program



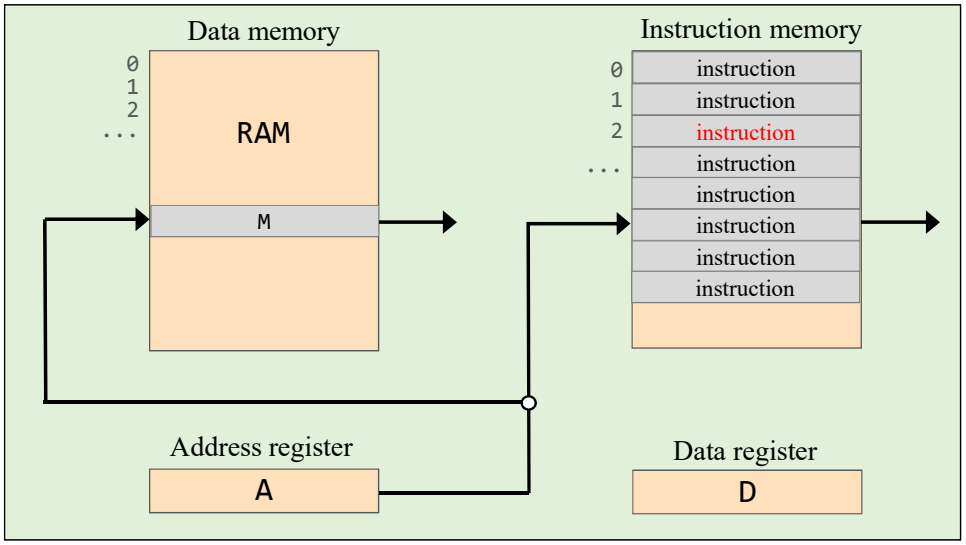
Executing a program



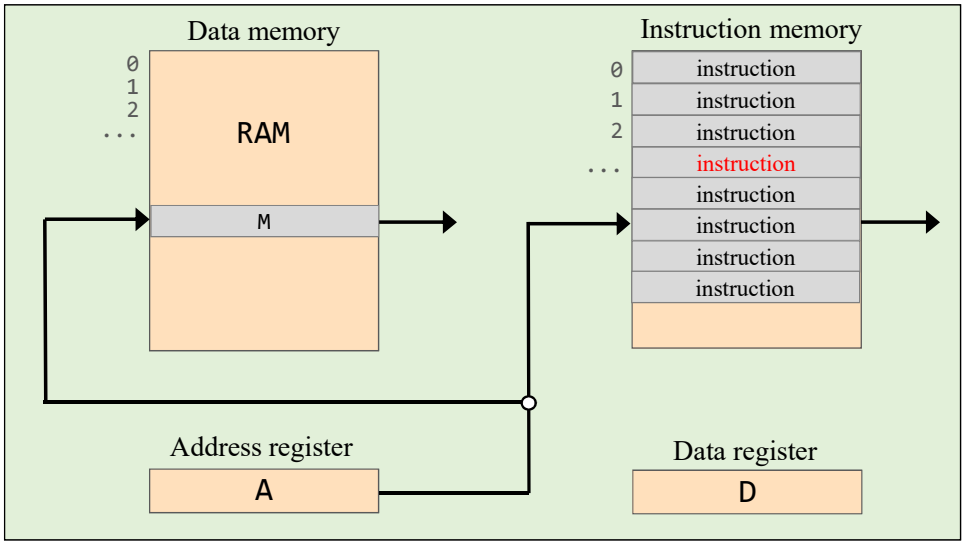
Executing a program



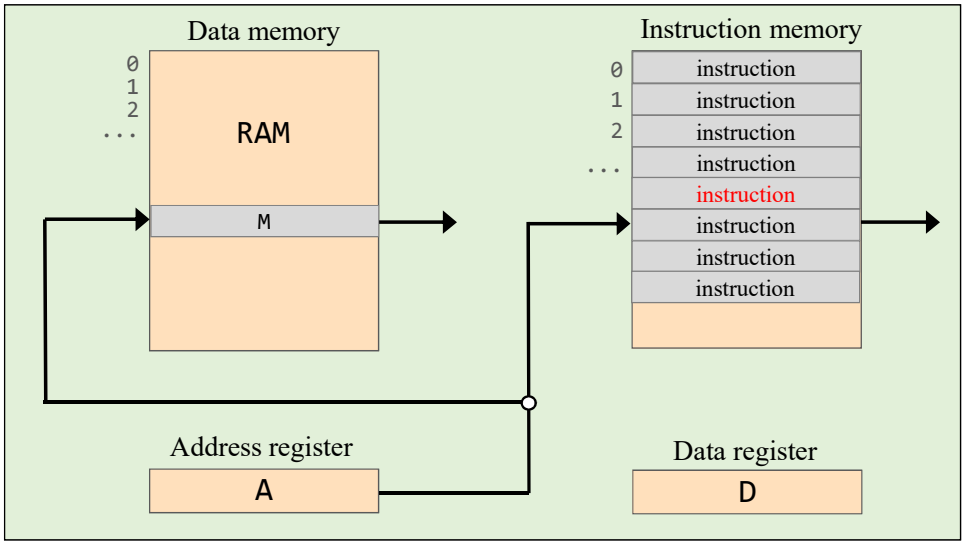
Executing a program



Executing a program

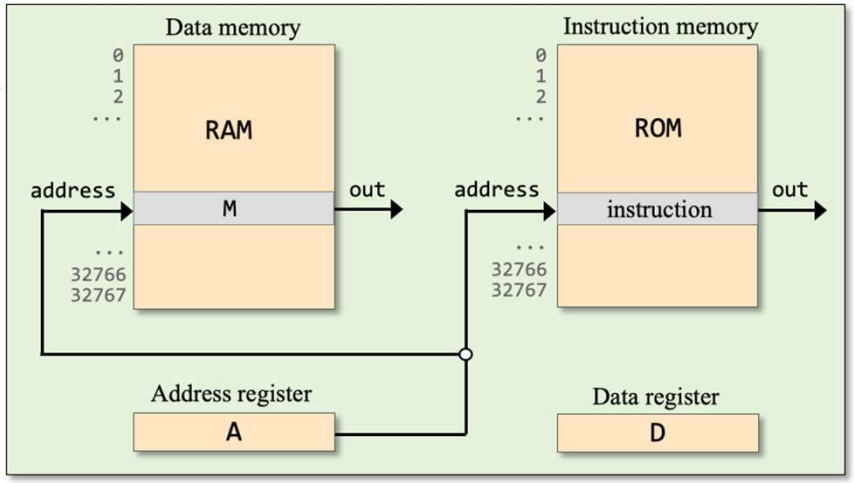


Executing a program



- The default: Execute the next instruction
- Suppose we wish to execute another instruction
- How to specify this *branching*?

Branching



Unconditional branching example (pseudocode)

```
0   instruction
1   instruction
2   instruction
3   instruction
4   goto 7
5   instruction
6   instruction
7   instruction
8   instruction
9   goto 2
10  instruction
11  ...
```

Flow of control:

0,1,2,3,4,

7,8,9,

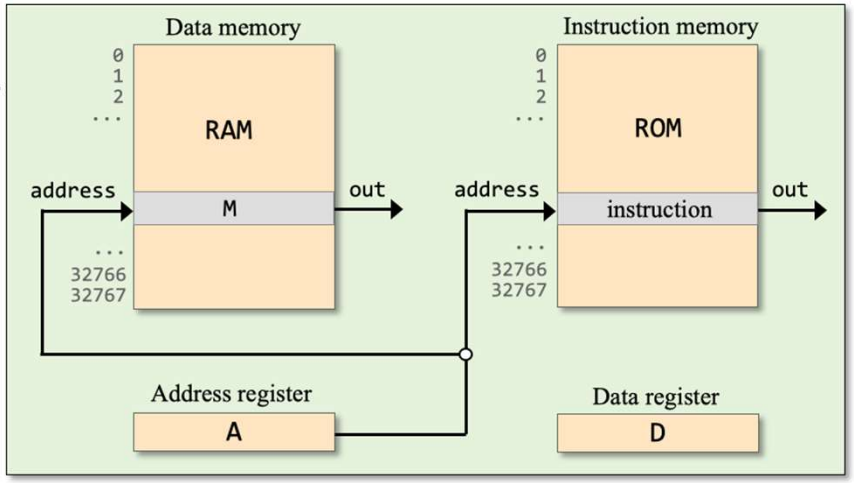
2,3,4,

7,8,9,

2,3,4,

...

Branching



Conditional branching example (pseudocode)

```
0   instruction
1   instruction
2   instruction
3   instruction
4   if (condition) goto 7
5   instruction
6   instruction
7   instruction
8   instruction
9   instruction
... ...
```

Flow of control:

0,1,2,3,4,

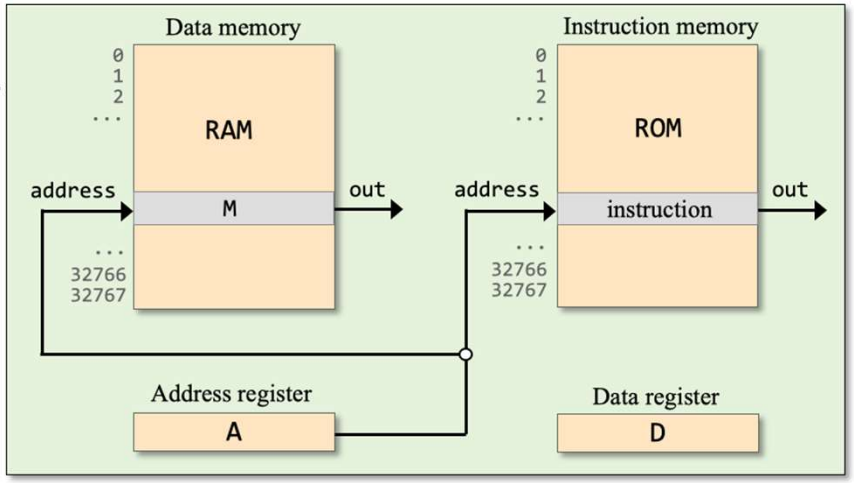
if *condition* is true

7,8,9,...

else

5,6,7,8,9,...

Branching



Branching in the Hack language:

Example (Pseudocode):

```
0    instruction
1    instruction
2    goto 6
3    instruction
4    instruction
5    instruction
6    instruction
7    instruction
...  ...
```

In Hack:

```
...
// goto 6
@6
0;JMP
...
```

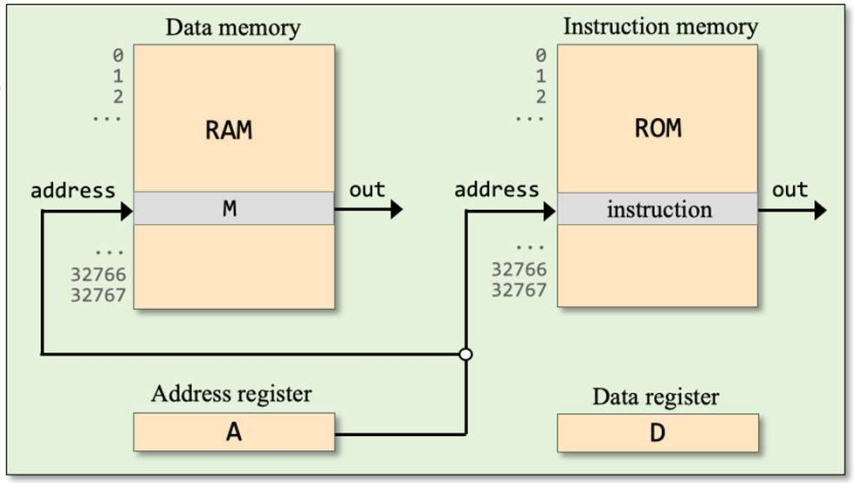
Semantics of `0;JMP`

Jump to the instruction stored in the register selected by `A`

(the "`0;`" prefix will be explained later)

Branching

Branching in the Hack language:



Example (Pseudocode):

```
0    instruction
1    instruction
2    if (D>0) goto 6
3    instruction
4    instruction
5    instruction
6    instruction
7    instruction
...  ...
```

In Hack:

```
...
// if (D>0) goto 6
@6
D;JGT
...
```

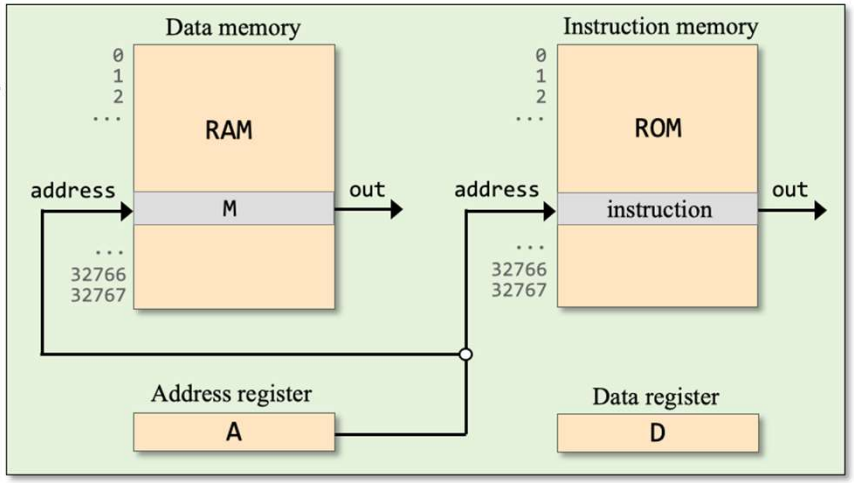
Typical branching instructions:

`D;JGT` // if `D > 0` jump

} to the instruction stored in `ROM[A]`

Branching

Branching in the Hack language:



Example (Pseudocode):

```
0    instruction
1    instruction
2    if (D>0) goto 6
3    instruction
4    instruction
5    instruction
6    instruction
7    instruction
...  ...
```

In Hack:

```
...
// if (D>0) goto 6
@6
D;JGT
...
```

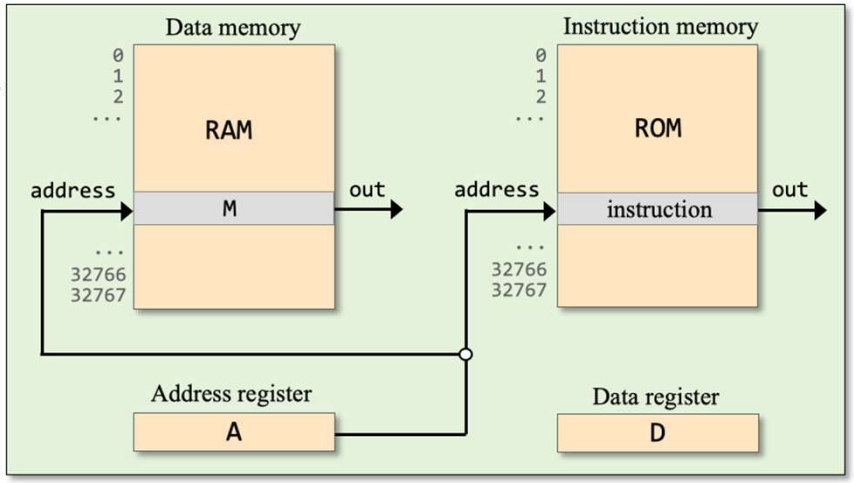
Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
```

to the instruction stored in `ROM[A]`

Branching

Branching in the Hack language:



Example (Pseudocode):

```
0     instruction
1     instruction
2     if (D>0) goto 6
3     instruction
4     instruction
5     instruction
6     instruction
7     instruction
...   ...
```

In Hack:

```
...
// if (D>0) goto 6
@6
D;JGT
...
```

Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to the instruction stored in `ROM[A]`

Branching

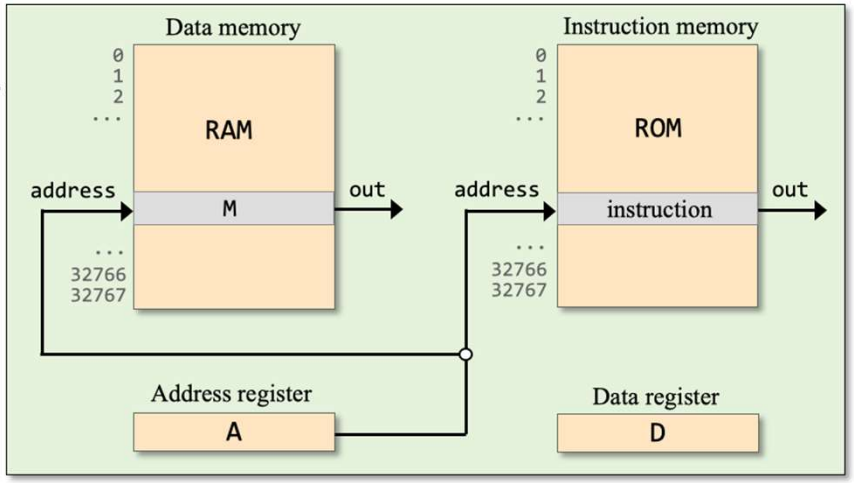
Typical instructions:

`@`*constant* (A←*constant*)

```
A=1
D=-1
M=0
...
```

```
A=M
D=A
M=D
...
```

```
D=D-A
A=A-1
M=D+1
...
```




```
// if (D = 0) goto 300
?
```

Use only the instructions shown in the current slide

Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to the instruction stored in `ROM[A]`

Branching

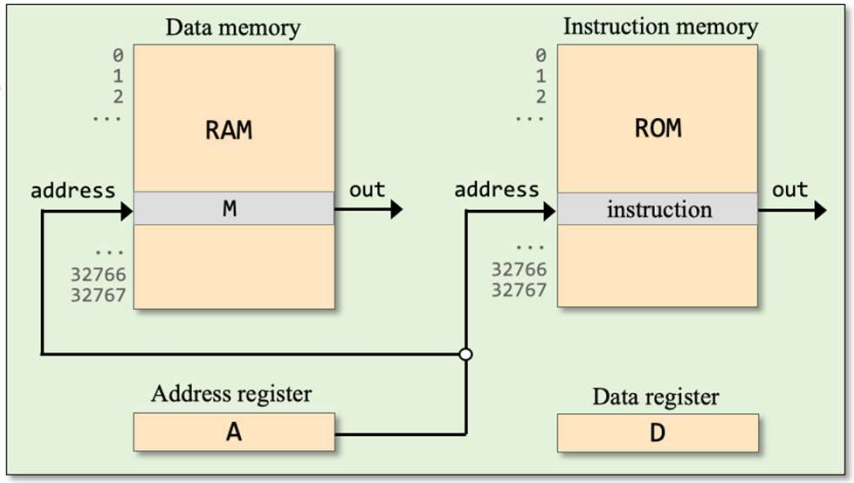
Typical instructions:

@ *constant* (A ← *constant*)

```
A=1
D=-1
M=0
...
```

```
A=M
D=A
M=D
...
```

```
D=D-A
A=A-1
M=D+1
...
```




```
// if (D = 0) goto 300
```

Use only the instructions shown in the current slide

Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to the instruction stored in `ROM[A]`

Branching

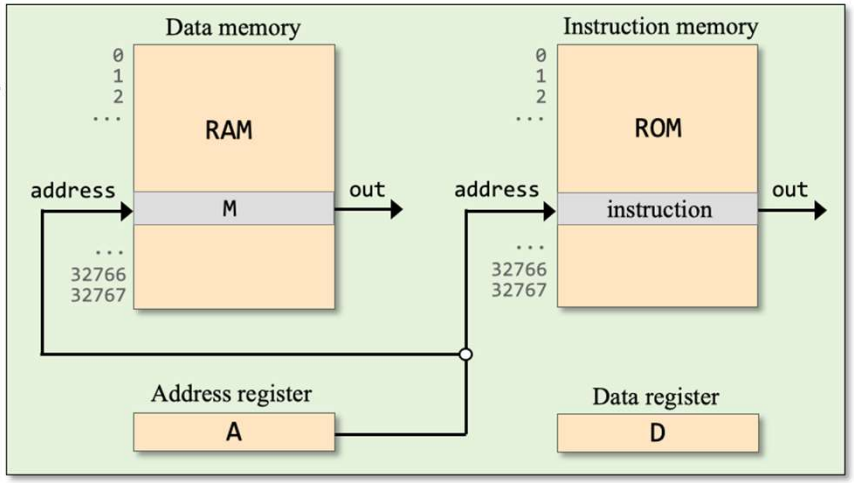
Typical instructions:

@ *constant* (A ← *constant*)

```
A=1
D=-1
M=0
...
```

```
A=M
D=A
M=D
...
```

```
D=D-A
A=A-1
M=D+1
...
```




```
// if (D = 0) goto 300
@300
D;JEQ
```

Use only the instructions shown in the current slide

Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to the instruction stored in `ROM[A]`

Branching

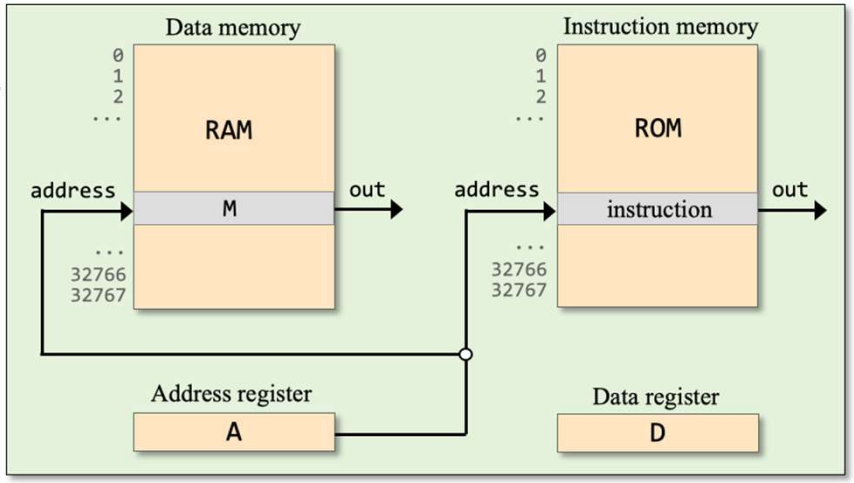
Typical instructions:

`@`*constant* (A ← *constant*)

```
A=1
D=-1
M=0
...
```

```
A=M
D=A
M=D
...
```

```
D=D-A
A=A-1
M=D+1
...
```



Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to the instruction stored in `ROM[A]`

Branching

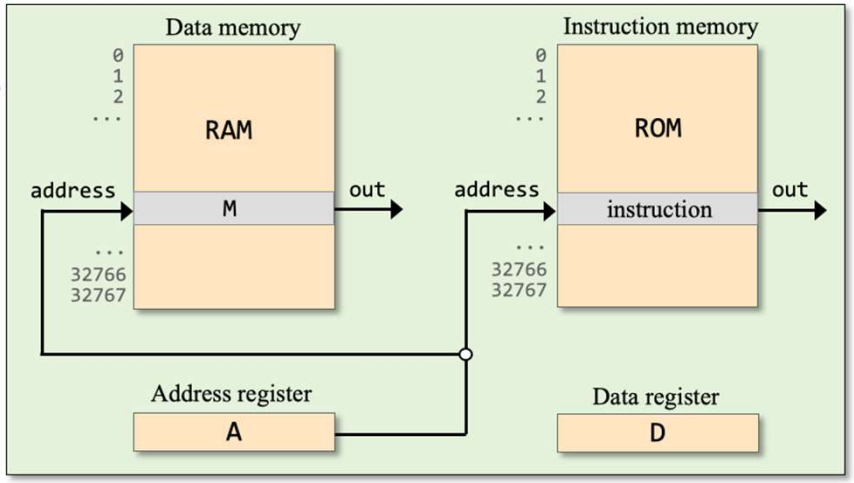
Typical instructions:

`@`*constant* (A ← *constant*)

```
A=1
D=-1
M=0
...
```

```
A=M
D=A
M=D
...
```

```
D=D-A
A=A-1
M=D+1
...
```




```
// if (RAM[3] < 100) goto 12
?
```

Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to the instruction stored in `ROM[A]`

Use only the instructions shown in the current slide

Branching

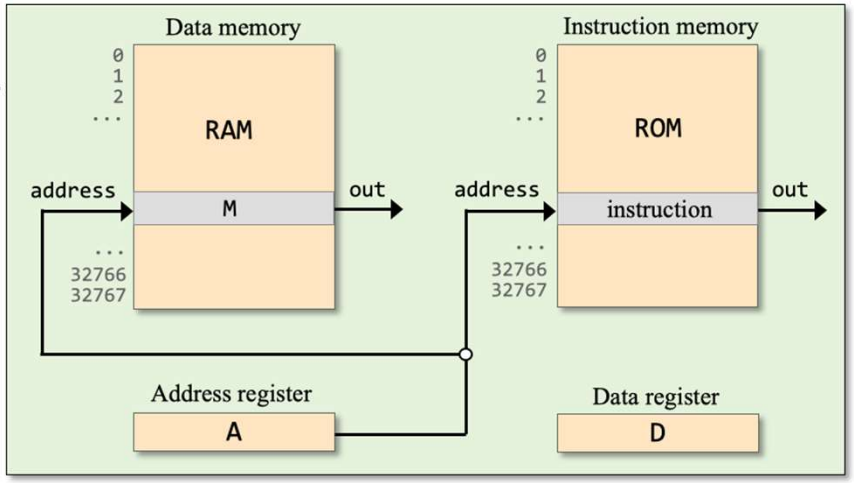
Typical instructions:

`@`*constant* (A←*constant*)

```
A=1
D=-1
M=0
...
```

```
A=M
D=A
M=D
...
```

```
D=D-A
A=A-1
M=D+1
...
```




```
// if (RAM[3] < 100) goto 12
```

Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to the instruction stored in `ROM[A]`

Use only the instructions shown in the current slide

Branching

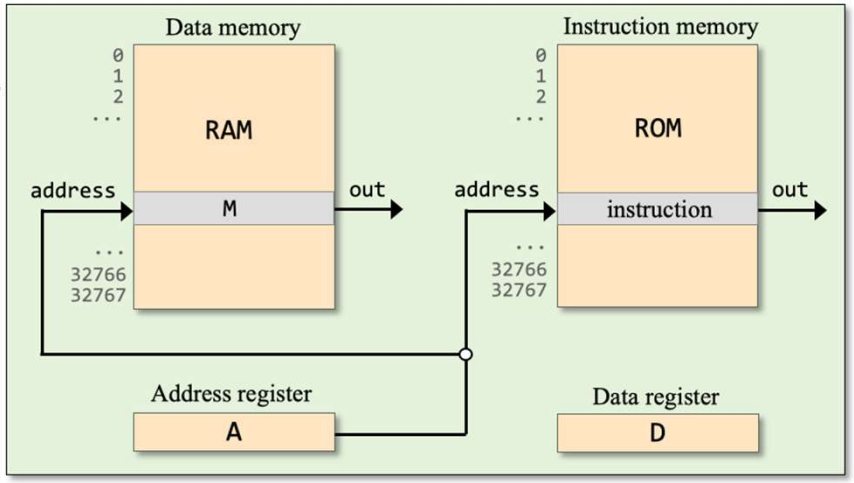
Typical instructions:

@ *constant* (A ← *constant*)

```
A=1
D=-1
M=0
...
```

```
A=M
D=A
M=D
...
```

```
D=D-A
A=A-1
M=D+1
...
```




```
// if (RAM[3] < 100) goto 12
// D = RAM[3] - 100
@3
D=M
@100
D=D-A
// if (D < 0) goto 12
@12
D;JLT
```

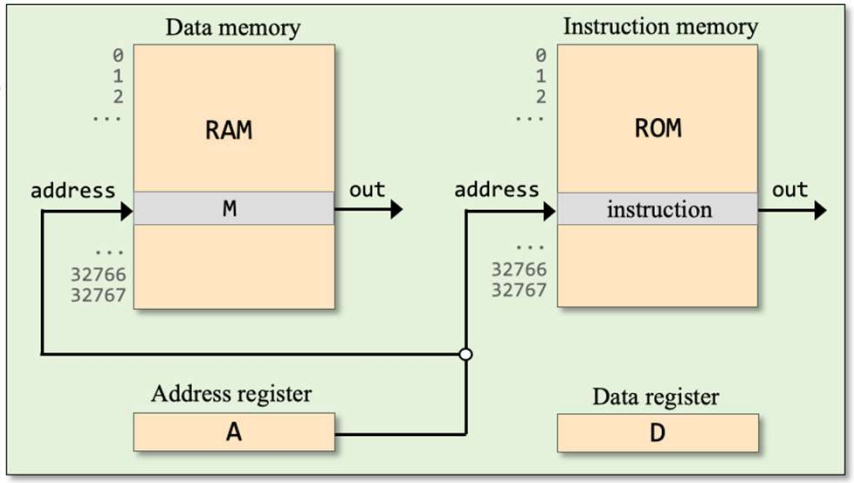
Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to the instruction stored in `ROM[A]`

Use only the instructions shown in the current slide

Hack instructions



- A instruction
- C instruction

Syntax:

`@const`

where *const* is a constant

Example:

```
@19    // A ← 19
```

`@sym`

where *sym* is a symbol bound to a constant

```
@x
```

For example, if `x` is bound to 21, this instruction will set A to 21

This idiom can be used for realizing:

- Variables
- Labels

Variables

Pseudocode (example)

```
...
i = 1
sum = 0
...
sum = sum + i
i = i + 1
...
```

Hack assembly

```
...
// i = 1
```

Variables

Pseudocode (example)

```
...
i = 1
sum = 0
...
sum = sum + i
i = i + 1
...
```

write

Hack assembly

```
...
// i = 1
@i
M=1
// sum = 0
@sum
M=0
...
// sum = sum + i
@i
D=M
@sum
M=D+M
// i = i + 1
@i
M=M+1
...
```

Symbolic programming

- The code writer is allowed to use symbolic variables, as needed
- We assume that there is an agent who knows how to bind these symbols to selected RAM addresses

  This agent is the *assembler*

For example

- If the assembler will bind `i` to 16 and `sum` to 17, every instruction `@i` and `@sum` will end up selecting `RAM[16]` and `RAM[17]`
- Should the code writer worry about what is the actual bindings? No
- The result: a low-level model for representing *variables*.

Variables

Typical instructions:

`@` *constant* — A ← *constant*

`@` *symbol* — A ← the constant which is bound to *symbol*

```
D=0
M=1
D=-1
M=0
...
```

```
D=M
A=M
M=D
D=A
...
```

```
D=D+A
D=A+1
D=D+M
M=M-1
...
```

```
// sum = 0
?
```

```
// x = 512
?
```

```
// n = n - 1
?
```

```
// sum = sum + x
?
```

Use only the instructions shown in the current slide

Variables

Typical instructions:

`@` *constant* — A ← *constant*

`@` *symbol* — A ← the constant which is bound to *symbol*

```
D=0
M=1
D=-1
M=0
...
```

```
D=M
A=M
M=D
D=A
...
```

```
D=D+A
D=A+1
D=D+M
M=M-1
...
```

```
// sum = 0
@sum
M=0
```

```
// x = 512
@512
D=A
@x
M=D
```

```
// n = n - 1
@n
M=M-1
```

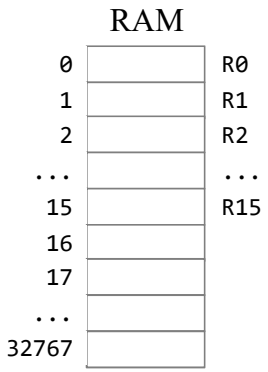
```
// sum = sum + x
@sum
D=M
@x
D=D+M
@sum
M=D
```

Use only the instructions shown in the current slide

Variables

Pre-defined symbols

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15



- As if you have 16 built-in variables named `R0`…`R15`
- We sometimes call them "virtual registers"

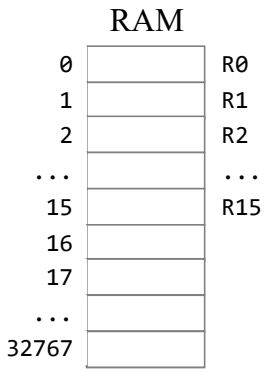
Example:

```
// Sets R1 to 2 * R0
// Usage: Enter a value in R0
```

Variables

Pre-defined symbols

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15



- As if you have 16 built-in variables named `R0`…`R15`
- We sometimes call them "virtual registers"

Example:

```
// Sets R1 to 2 * R0
// Usage: Enter a value in R0
@R0
D=M
@R1
M=D
M=D+M
```

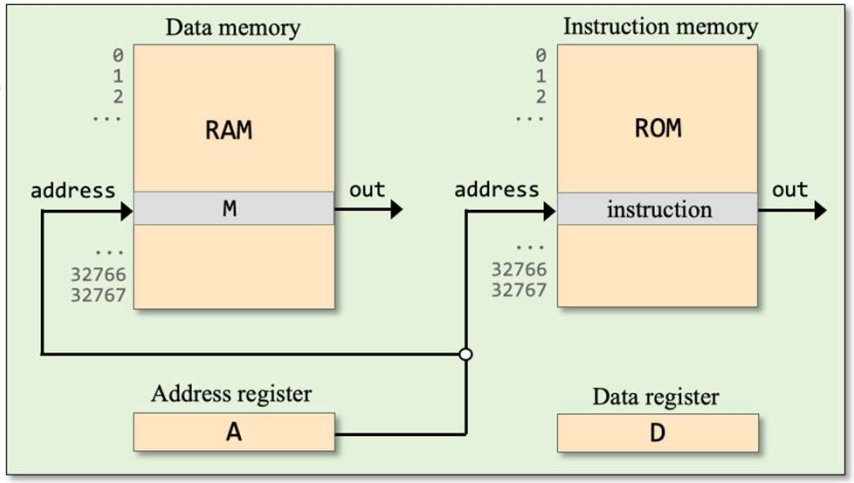
The use of `R0`, `R1`, … (instead of physical addresses `0`, `1`, …) makes it easier to document, write, and debug Hack code.

Labels

Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to `ROM[A]`



Examples (similar to what we did before):

```
// goto 48
```

?

```
// if (D > 0) goto 21
```

?

```
// if (RAM[100] < 0) goto 35
```

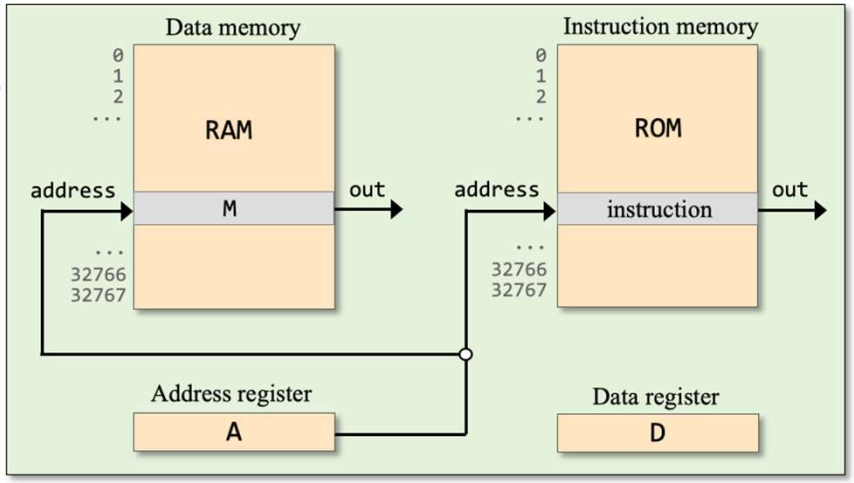
?

Labels

Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to `ROM[A]`



Examples (similar to what we did before):

```
// goto 48
@48
0;JMP
```

```
// if (D > 0) goto 21
@21
D;JGT
```

```
// if (RAM[100] < 0) goto 35
@100
D=M
@35
D;JLT
```

Same examples, using *labels*

```
// goto LOOP
```

```
// if (D > 0) goto CONT
```

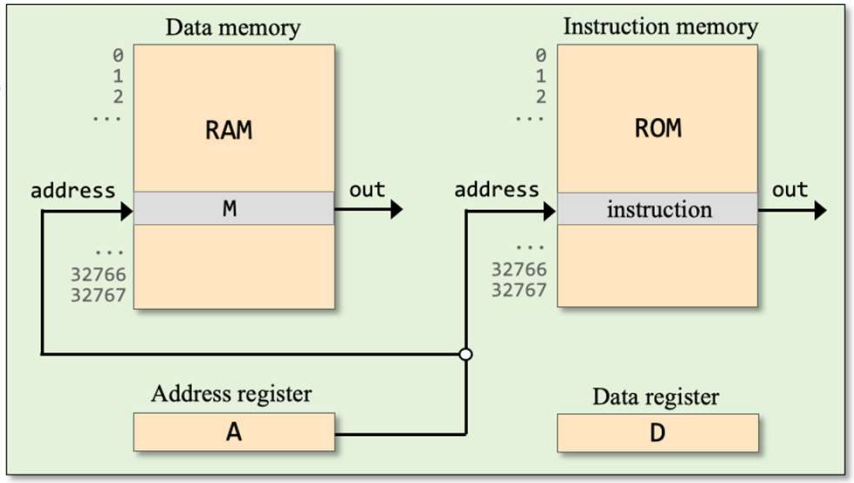
```
// if (x < 0) goto NEG
```

Labels

Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to `ROM[A]`



Examples (similar to what we did before):

```
// goto 48
@48
0;JMP
```

```
// if (D > 0) goto 21
@21
D;JGT
```

```
// if (RAM[100] < 0) goto 35
@100
D=M
@35
D;JLT
```

Same examples, using *labels*

```
// goto LOOP
@LOOP
0;JMP
```

```
// if (D > 0) goto CONT
```

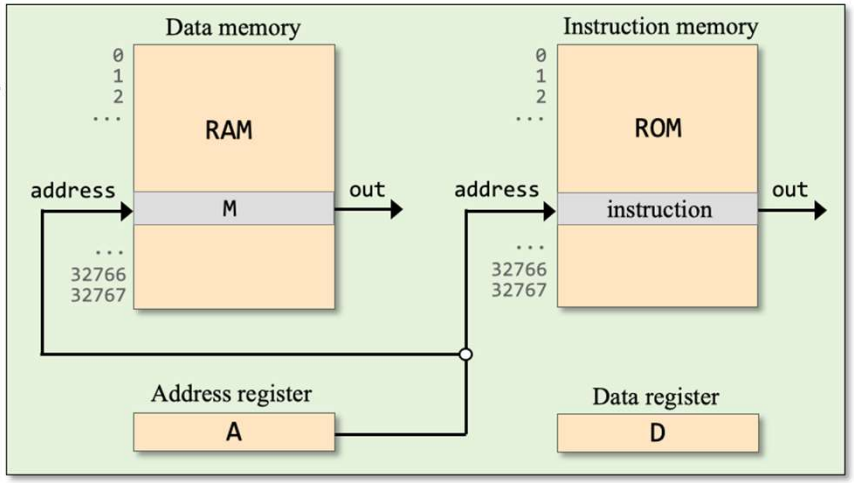
```
// if (x < 0) goto NEG
```

Labels

Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to `ROM[A]`



Examples (similar to what we did before):

```
// goto 48
@48
0;JMP
```

```
// if (D > 0) goto 21
@21
D;JGT
```

```
// if (RAM[100] < 0) goto 35
@100
D=M
@35
D;JLT
```

Same examples, using *labels*

```
// goto LOOP
@LOOP
0;JMP
```

```
// if (D > 0) goto CONT
@CONT
D;JGT
```

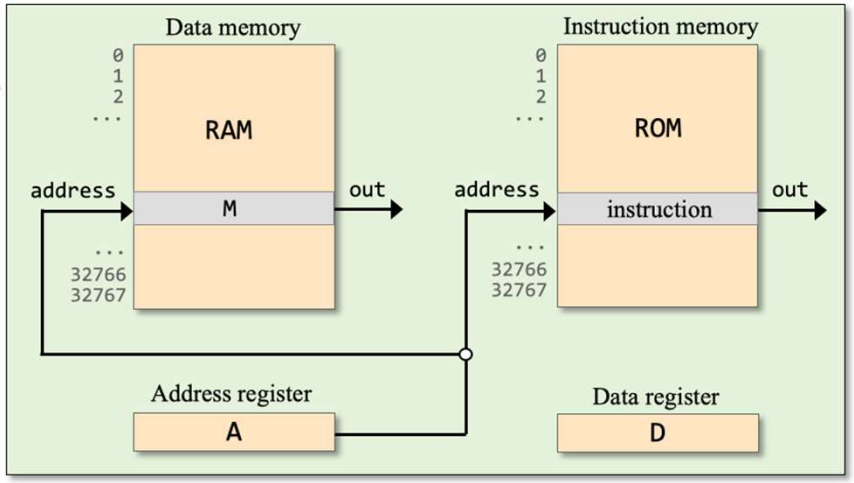
```
// if (x < 0) goto NEG
```

Labels

Typical branching instructions:

```
D;JGT  // if D > 0 jump
D;JGE  // if D ≥ 0 jump
D;JLT  // if D < 0 jump
D;JLE  // if D ≤ 0 jump
D;JEQ  // if D = 0 jump
D;JNE  // if D ≠ 0 jump
0;JMP  // jump
```

to ROM[A]



Examples (similar to what we did before):

```
// goto 48
@48
0;JMP
```

```
// if (D > 0) goto 21
@21
D;JGT
```

```
// if (RAM[100] < 0) goto 35
@100
D=M
@35
D;JLT
```

Same examples, using *labels*

```
// goto LOOP
@LOOP
0;JMP
```

```
// if (D > 0) goto CONT
@CONT
D;JGT
```

```
// if (x < 0) goto NEG
@x
D=M
@NEG
D;JLT
```

Hack convention:

Variables: lower-case symbols

Labels: upper-case symbols

Labels

Example (pseudocode)

```
    ...
LOOP:
    if (i = 0) goto CONT
    do this
    ...
    goto LOOP
CONT:
    do that
    ...
```

Hack assembly

Labels

Example (pseudocode)

```
    ...
LOOP:
    if (i = 0) goto CONT
    do this
    ...
    goto LOOP
CONT:
    do that
    ...
```

Hack assembly

```
    ...
(LOOP)
    // if (i = 0) goto CONT
    @i
    D=M
    @CONT
    D;JEQ
    do this
    ...
    // goto LOOP
    @LOOP
    0;JMP
(CONT)
    do that
    ...
```

Hack assembly syntax:

- A label *sym* is declared using (*sym*)
- Any label *sym* declared somewhere in the program can appear in a @*sym* instruction
- The assembler resolves the labels to actual addresses.

Programs that use symbolic labels and variables are...

- Easy to write / translate from pseudocode
- Readable
- Relocatable.