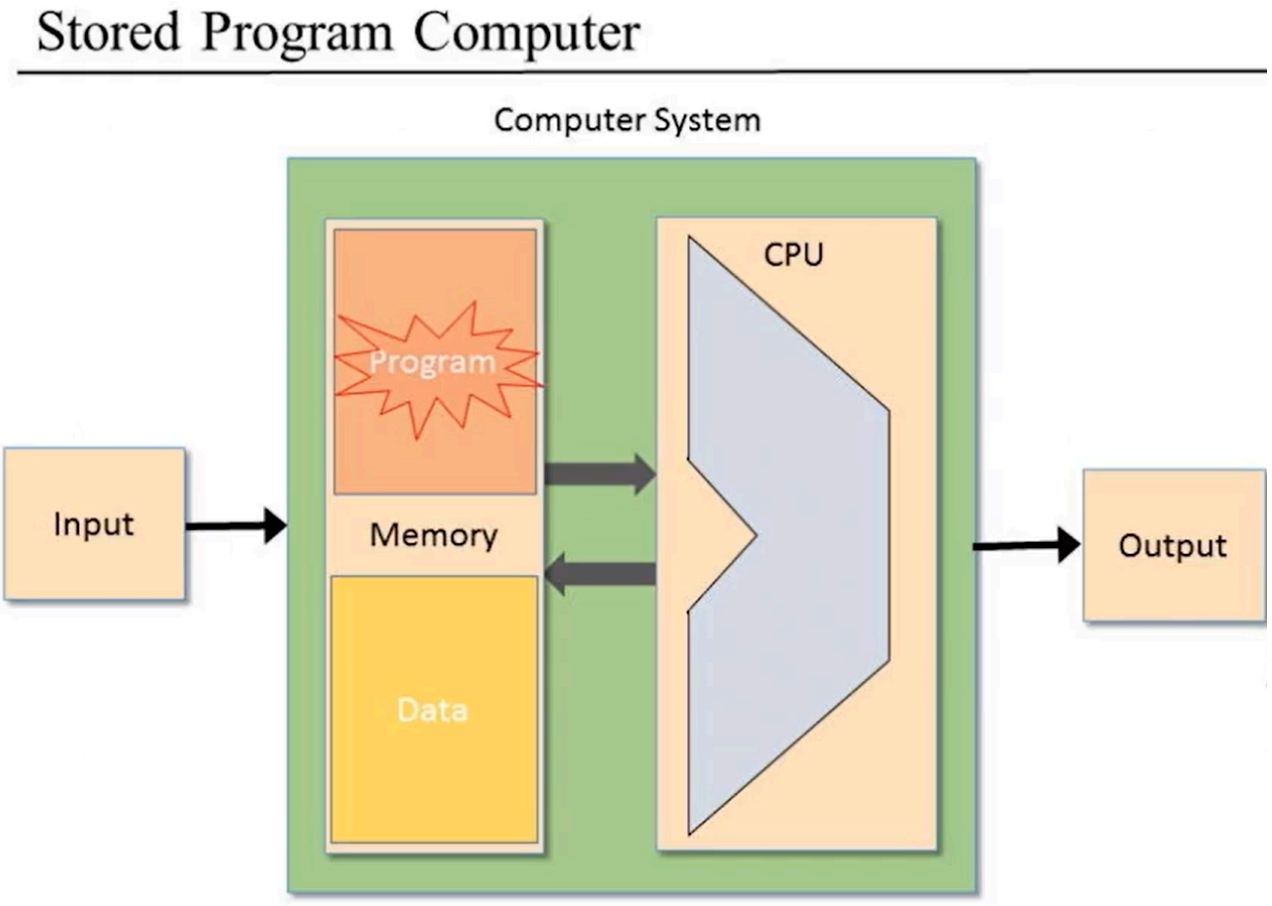
# Module 4 - Machine Language

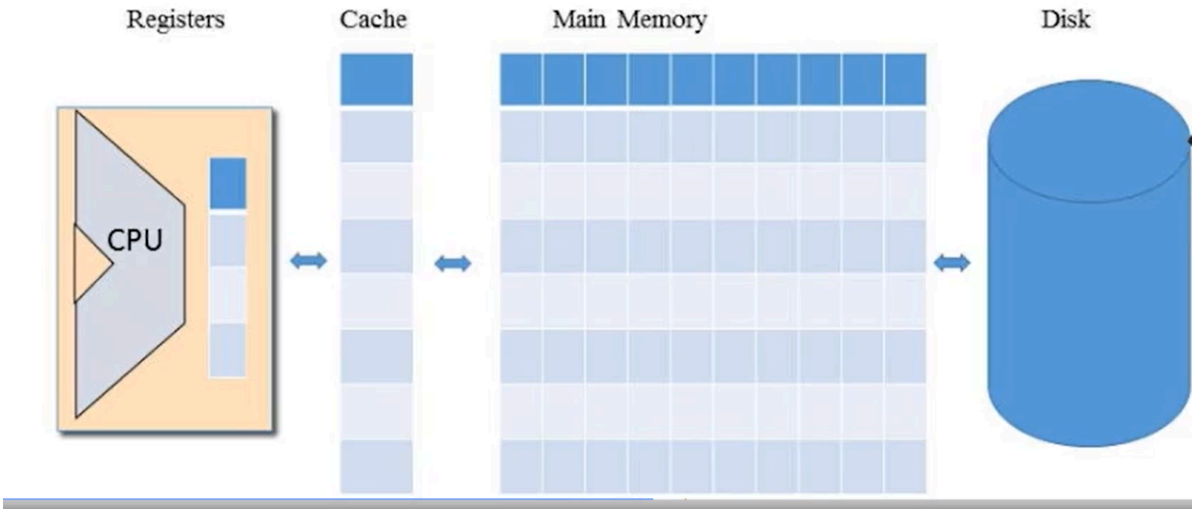



- **Compiler** transform high level programming language to machine language
- **Assembler** transform assembly language (which is just mnemonics) to machine language (bit-form)

## Machine Language

- Specification of the Hardware and Software Interface
  - What are the supported operations?
    - Usually correspond to what is implemented in Hardware
      - Arithmetic operations: add, subtract, ...
      - Logical Operations: and, or, ...
      - Flow Control: "goto instruction X", "if C then goto instruction Y"
    - Differences in machine languages
      - Richness of the set of operations (Divisions? Bulk Copy? ...)
      - Data Types (width (Adding 8-bit numbers or adding 64-bit numbers), floating point, ...)

    - What do they operate on?
    - How is the program controlled?
- There is usually a cost-performance tradeoff
    - Silicon area
    - Time to complete instruction
- **Memory Hierarchy**
    - Accessing a memory location is expensive if memory is big (memory address will be a long bit number)
    - Getting memory contents into the CPU takes time
    - So we have memory hierarchy

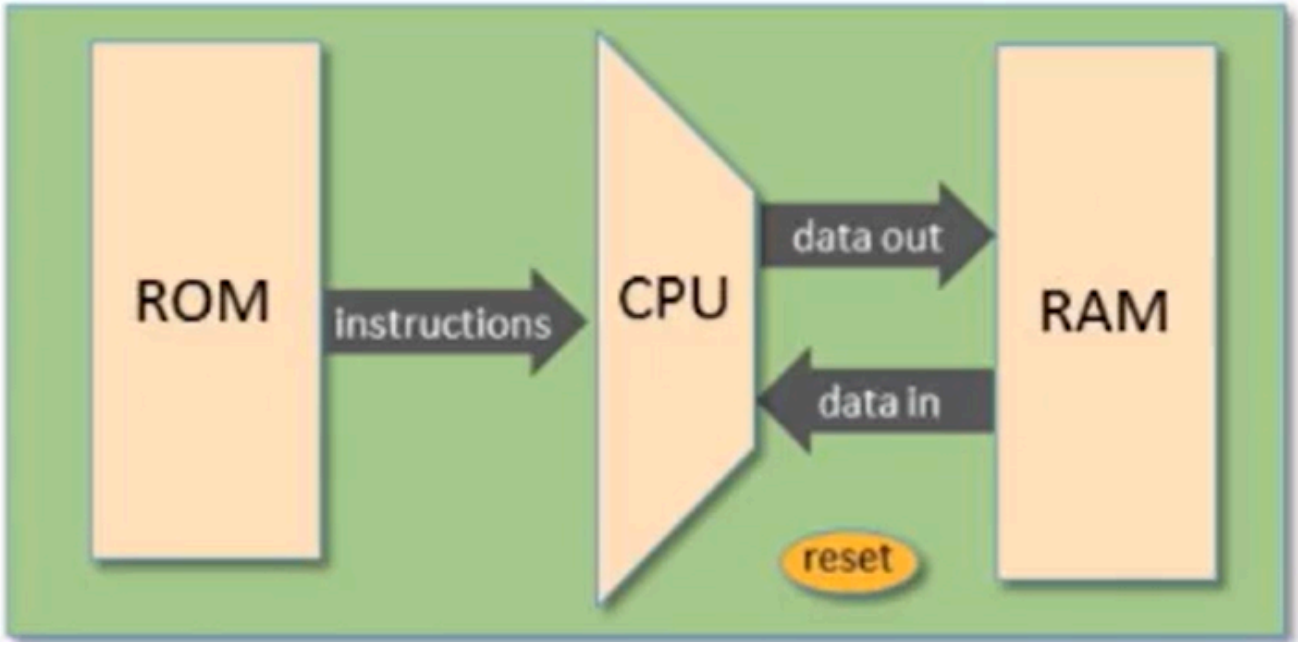



    - CPUs usually contain a few easily accessed "registers".
        - Their number and functions are a central part of the machine language
        - We have:
            - Data registers: store values/data
            - Address Registers: store addresses of memory in bigger memory
        - Addressing Modes:
            - Register
                - E.g. Add R1,R2 : R2 ← R2 + R1
            - Direct
                - E.g. Add R1, Mem[200]
            - Indirect
                - E.g. Add R1, @A [Address of location A]
            - Immediate
                - Add 73, R1
        - Input/Output:
            - Many types of input and output devices: keyboard, mouse, camera, etc.
            - CPU needs some protocol to talk to them: These are included in the software "drivers"
            - One general method of interaction uses "memory mapping"

        - E.g. memory location 12345 holds the direction of the last movement of the mouse
        - E.g. Memory location 45678 is not a real memory location, but a way to tell the printer which paper to use.
    - Flow Control:
        - Usually CPU executes machine instructions in sequence.
        - Sometimes we need to jump unconditionally to another location (e.g. so we can loop)
        - Sometimes we jump only if some condition is met

# Hack Computer and Hack Machine Language

Overview:




- A 16-bit machine consisting of:
    - Data memory (RAM): a sequence of 16-bit registers (RAM[0], RAM[1], ....)
    - Instruction memory (ROM): a sequence of 16-bit registers (ROM[0], ROM[1], ....)
    - Central Processing Unit (CPU): performs 16-bit operations
    - Instruction bus/data bus/ address bus
- Hack machine language:
    - 16-bit A-instructions
    - 16-bit C-instructions
- Hack program = sequence of instructions written in the Hack machine language
- Control:
    - The ROM is loaded with a Hack program
    - The *reset* button is pushed
    - The program starts running

- The Hack machine language recognises three registers:
  - `D` (Data) register holds a 16-bit value
  - `A` (Addressing) register holds a 16-bit value
  - `M` (Memory) register represents the 16-bit RAM register addressed by `A`.

## The A-instruction

Syntax: `@value`

- where `value` is either
  - a non-negative decimal constant, OR
  - a symbol referring to such a constant

Semantics:

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

Example: `@21`

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

Usage Example: Want to set `RAM[100]` to $-1$

```
@100      // A=100  and now M register represent RAM[100]
M=-1      // RAM[100]=-1
```

## The C-instruction

Syntax: `dest = comp ; jump` [Both `dest` and `jump` are optional]

- $comp$ =
```
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]
  - `null` means we don't want to store the result of the computation
- $jump$ = `null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP` if ($comp$ $jump$ 0) jump to execute the instruction in ROM[A]
  - E.g. JGT = jump when greater than 0

- E.g. JMP = jump [with no conditions]

Semantics:

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

Example1: Set the `D` register to $-1$

```
D=-1   \\ dest = D, comp = -1
```

Example2: Set `RAM[100]` to the value of the `D` register minus $1$

```
@100     \\ A=100
M=D-1    \\ RAM[100]=D-1, dest = M, comp = D-1
```

Example3: If `(D-1==0)` jump to execute the instruction stored in `ROM[56]`.

```
@56     \\ A = 56
D-1;JEQ \\ if (D-1==0) goto ROM[56]
```

# Specifications

# 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 $\leq 32767$ $(=2^{15}-1)$ or
- a symbol referring to such a constant (later)

Example:

```
@21
```

Effect: sets the A register to 21

Binary syntax:

`0value`

Where *value* is a 15-bit binary number

Example:

```
0000000000010101
```

Effect: sets the A register to 21

# 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 | | | | | | |

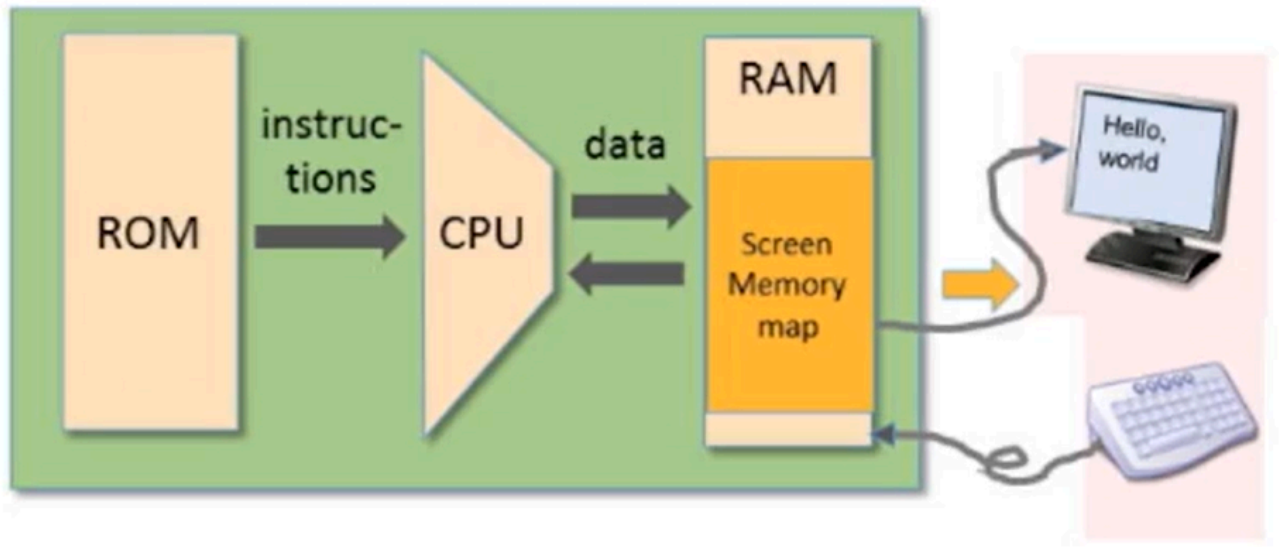
| *dest* | d1 | d2 | d3 | effect: the value is stored in: |
|---|---|---|---|---|
| null | 0 | 0 | 0 | The value is not stored |
| M | 0 | 0 | 1 | RAM[A] |
| D | 0 | 1 | 0 | D register |
| MD | 0 | 1 | 1 | RAM[A] and D register |
| A | 1 | 0 | 0 | A register |
| AM | 1 | 0 | 1 | A register and RAM[A]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

# Input and Output

- Peripheral I/O devices:
  - Keyboard: used to enter inputs
  - Screen: used to display outputs
- High-level approach: Sophisticated software libraries enabling, text, graphics, animation, audio, video, etc.\
- Low-level approach: bits

## Screen Memory Map



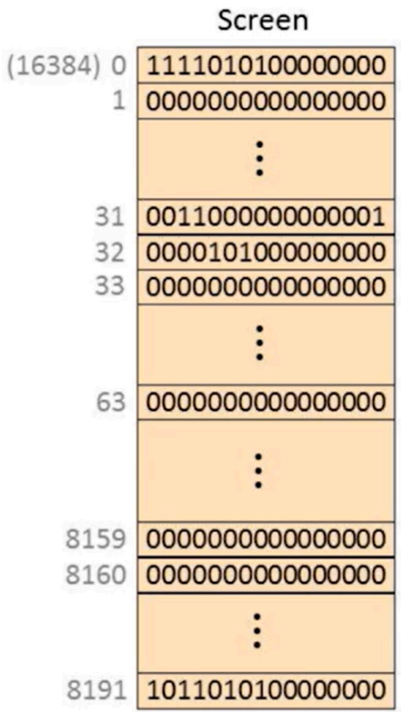

A designated memory area dedicated to manage a display unit.

- The physical display is continuously refreshed from the memory map, many times per second.
- Output is affected by writing code that manipulates the screen memory map.

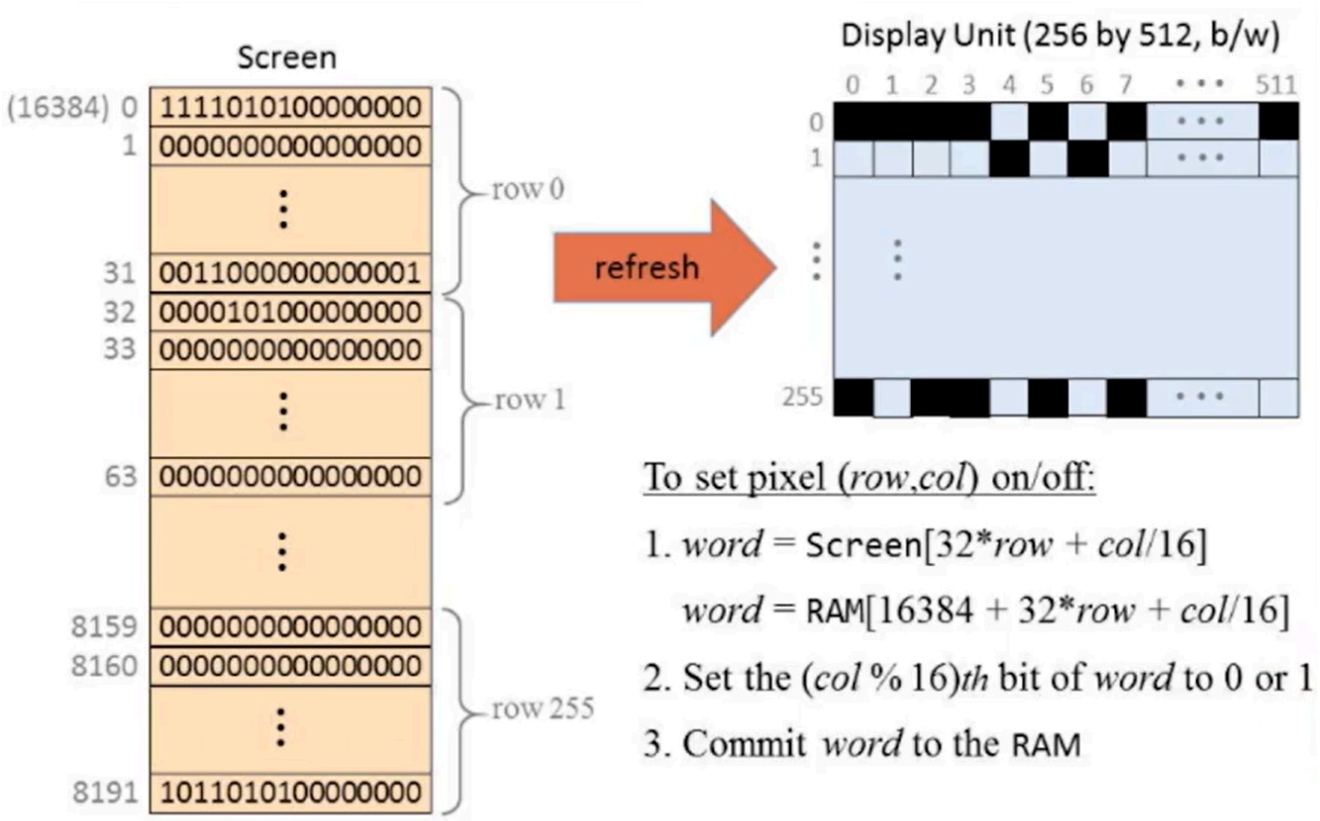
Example: A display unit of 256 by 512, black and white


The screen memory map will be 8192 chunks of 16-bit word.

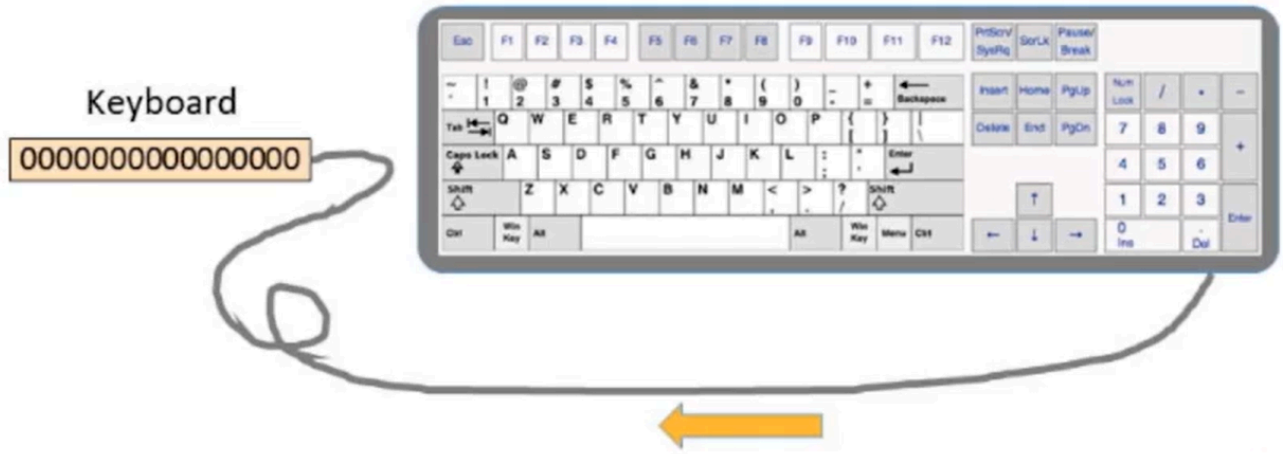
- That's because $256 \times 512 = 8192 \times 16$.




Whole process:




## Keyboard memory map




A 16-bit designated area to record what key is being pressed on the keyboard.

- When a key is pressed on the keyboard, the key's **scan code** appears in the keyboard memory map.

To check which key is currently pressed:

- Probe the contents of the `Keyboard` chip
- In the Hack computer: probe the contents of `RAM[24576]`

If the register contains 0, no key is pressed.

# Hack Programming

## Working with registers and memory

Examples of typical operations:

```
\\ D= 10
@10
D=A


\\ D++
D=D+1

\\ D=RAM[17]
@17
D=M

\\ RAM[17]=0
@17
```

```
M=0

\\ RAM[17]=10
@10
D=A
@17
M=D

\\ RAM[5]=RAM[3]
@3
D=M
@5
M=D
```

Example of Hack assembly code:

```
\\ Program Add2.asm
\\ Computes: RAM[2] = RAM[0] + RAM[1]
\\ Usage: put values in RAM[0], RAM[1]

@0
D=M   \\ D = RAM[0]

@1
D=D+M \\ D = D + RAM[1]

@2
M=D   \\ RAM[2] = D

@6
0;JMP  \\ jump to ROM[6], this is to create an infinite loop to stop the
program from going down the sequence in ROM, guarding against NOP slide
```

- NOP stands for Null instructions or Null Opcodes
- Translated into:


Hack assembly language has built-in symbols:

| symbol | value |
|---|---|
| R0 | 0 |
| R1 | 1 |
| R2 | 2 |
| ... | ... |
| R15 | 15 |
| SCREEN | 16384 |
| KBD | 24576 |

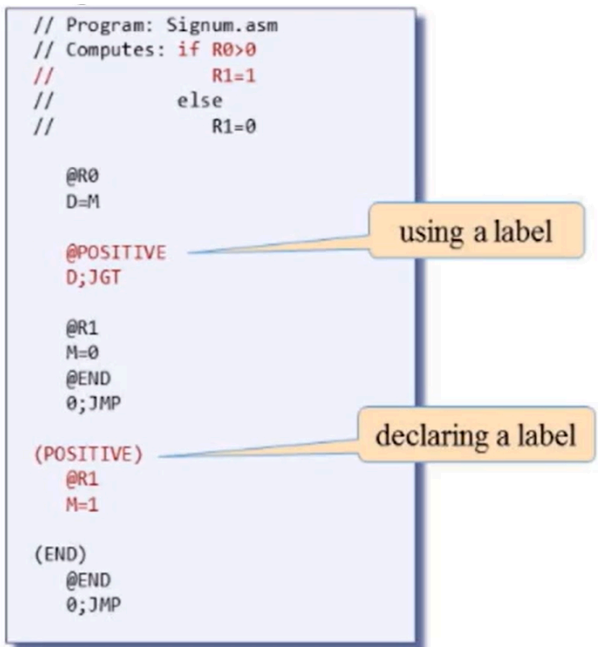
| symbol | value |
|---|---|
| SP | 0 |
| LCL | 1 |
| ARG | 2 |
| THIS | 3 |
| THAT | 4 |

- `R0, R1, ..., R15` are "virtual registers"
  - The convention is to use the virtual registers when we want to address the first 16 registers.
- `SCREEN` and `KBD` are base addresses of I/O memory maps
- Remaining symbols are used in the implementation of the Hack virtual machine discussed in Nand to Tetris Part II

**REMEMBER**: Hack is case-sensitive!!

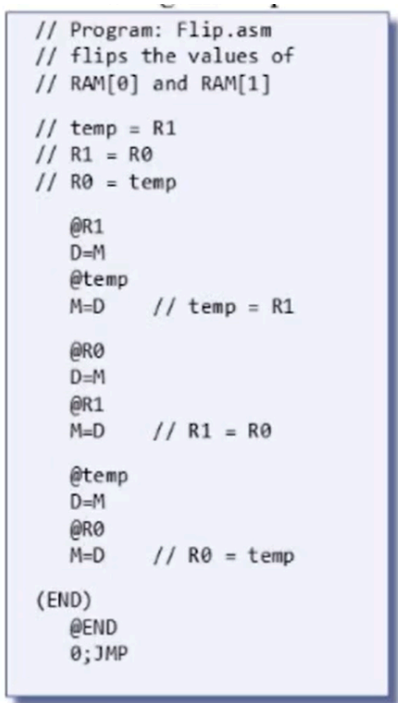
## Branching

- Can use a label as placeholder for jumping/branching

- `@LABEL` translates to `@n`, where `n` is the instruction number following the `(LABEL)` declaration
- 

- Contract:
  - Label declarations are not translated
  - Each reference to a label is replaced with a reference to the instruction number following the label's declaration

## Variables

- 

- `@temp`: find some available memory register (some register $n$) and use it to represent the variable `temp`. Now replace all `@temp` with `@n`.
- Contract:

- A reference to a symbol that has **no corresponding label** declaration is treated as a reference to a variable
- Variables are allocated to the `RAM` from address 16 onward

## Iteration

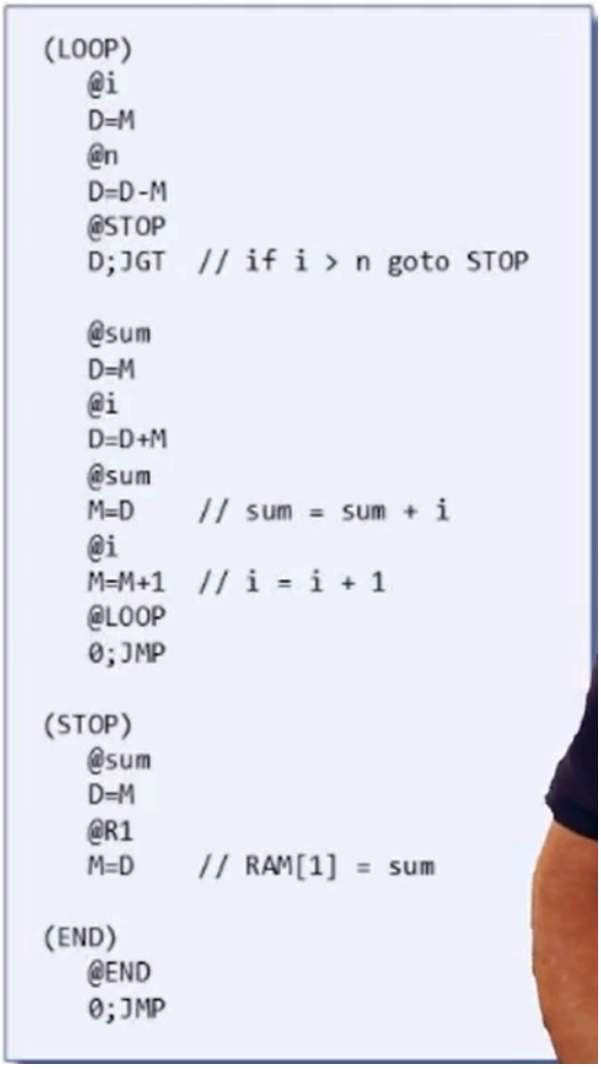
Example:

- Pseudo Code:

```
// Computes RAM[1] = 1+2+ ... +RAM[0]

   n = R0
   i = 1
   sum = 0
LOOP:
   if i > n goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP
STOP:
   R1 = sum
```

- Assembly code:

```
// Program: Sum1toN.asm
// Computes RAM[1] = 1+2+ ... +n
// Usage: put a number (n) in RAM[0]

   @R0
   D=M
   @n
   M=D   // n = R0
   @i
   M=1   // i = 1
   @sum
   M=0   // sum = 0
   ...
```

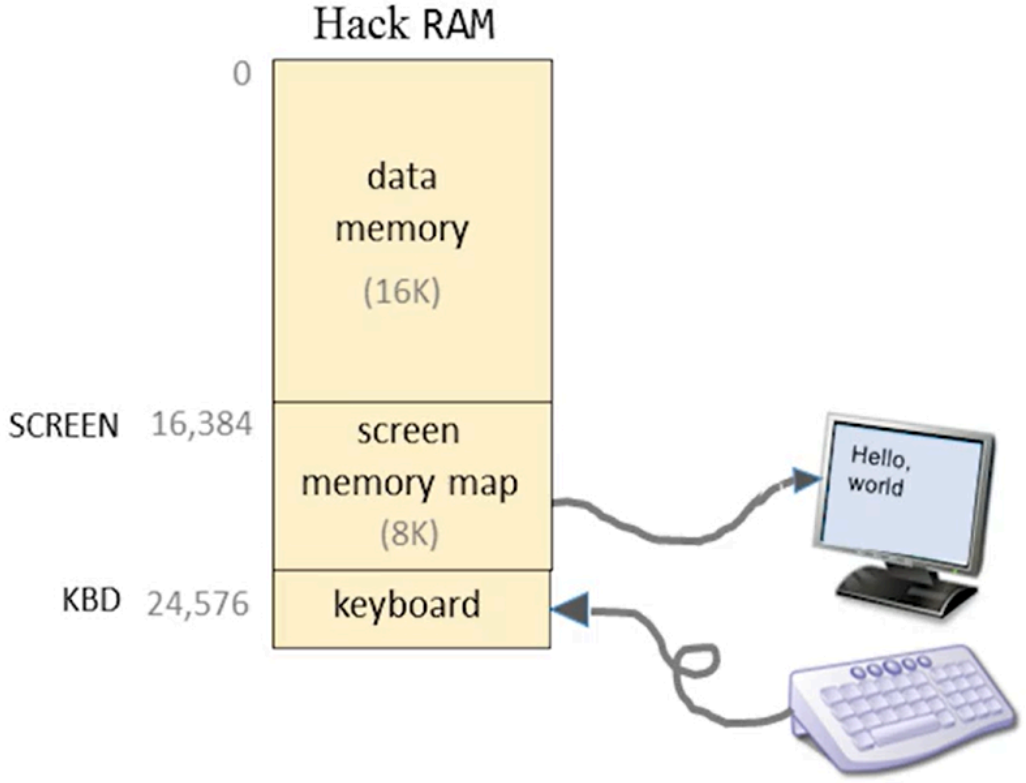


```
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @STOP
   D;JGT  // if i > n goto STOP

   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D    // sum = sum + i
   @i
   M=M+1  // i = i + 1
   @LOOP
   0;JMP

(STOP)
   @sum
   D=M
   @R1
   M=D    // RAM[1] = sum

(END)
   @END
   0;JMP
```

## Pointers

- Variables that store memory addresses are called **pointers**
- Hack pointer logic: Whenever we have to access memory using a pointer, we need an instruction like `A=M`.
- Typical pointer semantics: set the address register to the contents of some memory register (which is storing memory address)

## Input/Output




## Hack language convention:

- SCREEN: base address of the screen memory map
- KBD: address of the keyboard memory map