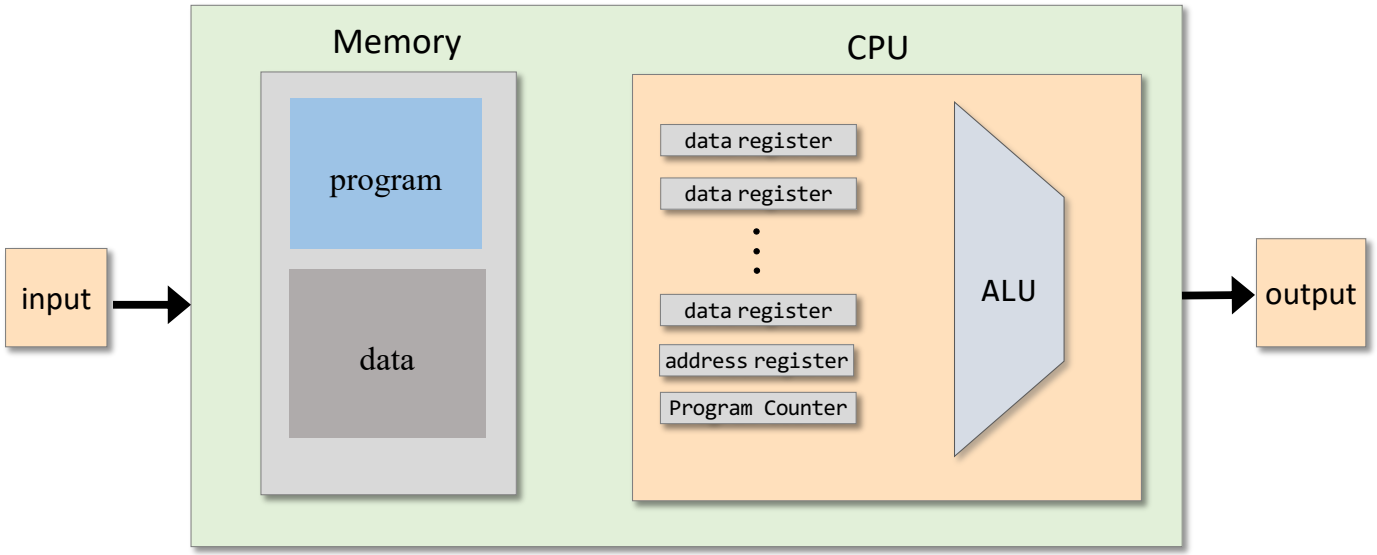
1. Langage assembleur



Caractéristiques de la Hack machine

- **Mémoire pour les données : RAM (16 bits)**
- **Mémoire pour les instructions : ROM (16 bits)**
- **Registres : A, D, M avec M=RAM[A]**
- **Opérations : ALU**
- **Compteur ordinal : PC**
- **Jeu d'instruction : A-instruction, C-instruction**
- **Un clavier et un écran**

Les instructions de la Hack

A instruction

Symbolic: `@`*xxx*

(*xxx* is a decimal value ranging from 0 to 32767, or a symbol bound to such a decimal value)

Binary: `0` *vvvvvvvvvvvvvvv*

(*vv ... v* = 15-bit value of *xxx*)

C instruction

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

(*comp* is mandatory.
If *dest* is empty, the `=` is omitted;
If *jump* is empty, the `;` is omitted)

Binary: `111`*accccccdddjjj*

comp		*c*	*c*	*c*	*c*	*c*	*c*
`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	*d*	*d*	*d*	Effect: store *comp* in:
`null`	0	0	0	the value is not stored
`M`	0	0	1	`RAM[A]`
`D`	0	1	0	`D` register (reg)
`DM`	0	1	1	`RAM[A]` and `D` reg
`A`	1	0	0	`A` reg
`AM`	1	0	1	`A` reg and `RAM[A]`
`AD`	1	1	0	`A` reg and `D` reg
`ADM`	1	1	1	`A` reg, `D` reg, and `RAM[A]`

jump	*j*	*j*	*j*	Effect:
`null`	0	0	0	no jump
`JGT`	0	0	1	if *comp* > 0 jump
`JEQ`	0	1	0	if *comp* = 0 jump
`JGE`	0	1	1	if *comp* ≥ 0 jump
`JLT`	1	0	0	if *comp* < 0 jump
`JNE`	1	0	1	if *comp* ≠ 0 jump
`JLE`	1	1	0	if *comp* ≤ 0 jump
`JMP`	1	1	1	unconditional jump

A-instruction

```
@value          // A ← value
```

où value est un nombre ou une référence symbolique à un nombre.

Entrer une constante	@17 // A=17
	D=A // D=17

Sélectionner une donnée dans la RAM	@17 // A=17
	D=M // D=RAM[17]

Sélectionner une instruction dans la ROM	@17 // A=17 JMP // PC=17

C-instructions

```
dest = x + y
dest = x - y
dest = x
dest = 0
dest = 1
dest = -1
```

x = {A, D, M}

y = {A, D, M , 1}

dest = {A, D, M, MD, A, AM, AD, AMD, null}

Exercices

- Set **D** to **A-1**
- Set both **A** and **D** to **A + 1**
- Set **D** to **19**
- Set both **A** and **D** to **A + D**
- Set **RAM[5034]** to **D - 1**
- Set **RAM[53]** to **171**
- Add **1** to **RAM[7]**, and store the result in **D**.

Etiquettes

(LABEL) crée une étiquette « numéro d'instruction »

@LABEL range dans le registre **A** la valeur de LABEL.
Si LABEL n'est pas
- une étiquette « numéro d'instruction »
- déjà défini

alors LABEL prend une valeur d'adresse dans la RAM.

symbol	*value*
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

Exercices

- `sum = 0`
- `j = j + 1`
- `q = sum + 12 – j`
- `arr[3] = -1`
- `arr[j] = 0`
- `arr[j] = 17`

Symbol table:

j	3012
sum	4500
q	3812
arr	20561

C-instructions (suite)

C-command: dest = comp ; jump // 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 = M , D , MD , A , AM , AD , AMD, or null

jump = JGT , JEQ , JGE , JLT , JNE , JLE , JMP, or null

Exercices

- goto 50
- if D==0 goto 112
- if D<9 goto 507
- if RAM[12] > 0 goto 50
- if sum>0 goto END
- if x[i]<=0 goto NEXT.

Symbol table:

sum	2200
x	4000
i	6151
END	50
NEXT	120

Attention : Les symboles représentent des adresses dans la RAM ou dans la ROM

Instruction if

High level:

```
if condition {
    code block 1}
else {
    code block 2}
code block 3
```

Hack:

```
    D ← not condition
    @IF_TRUE
    D;JEQ
    code block 2
    @END
    0;JMP
(IF_TRUE)
    code block 1
(END)
    code block 3
```

Hack convention:

- True is represented by -1
- False is represented by 0

Attention : (LABEL) est une étiquette égale au n° ligne de l'instruction suivante

Instruction while

High level:

```
while condition {
    code block 1
}
Code block 2
```

Hack:

```
(LOOP)
      D ← not condition)
      @END
      D;JEQ
      code block 1
      @LOOP
      0;JMP
(END)
      code block 2
```

Hack convention:

- True is represented by -1
- False is represented by 0

Exercice

Traduire le programme suivant pour une Hack Machine

C language code:

```
// Adds 1+...+100.
 into i = 1;
 into sum = 0;
 while (i <= 100){
    sum += i;
    i++;
 }
```

Hack assembly code:

```
// Adds 1+...+100.
       @i      // i refers to some RAM location
       M=1     // i=1
       @sum    // sum refers to some RAM location
       M=0     // sum=0
```

Le traducteur d'un programme assembleur

Symbolic low-level program

```
// Program: Sum1ToN (R0 represents N)
// Computes R1 = 1 + 2 + 3 + ... + R0
// Usage: put a value >= 1 in R0
   // i = 1
   @i
   M=1
   // sum = 0
   @sum
   M=0
(LOOP)
   // if(i > R0) goto STOP
   @i
   D=M
   @R0
   D=D-M
   @STOP
   D;JGT
   // sum = sum + i
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D
   // i = i + 1
   @i
   M=M+1
   // goto LOOP
   @LOOP
   0;JMP
   ...
```

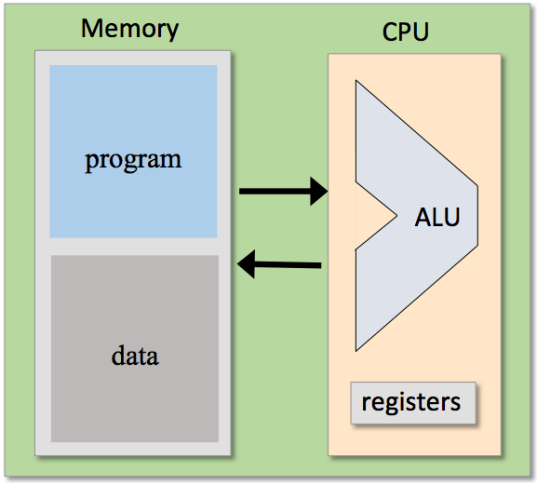
assembler

Binary code

```
0101111100111100
1010101010101010
1100000010101010
1011000010000001
0101111100111100
1010101010101010
1100000010101010
0101111100111100
1010101010101010
1100000010101010
1011000010000001
0101111100111100
1010101010101010
1100000010101010
0101111100111100
1010101010101010
1100000010101010
1011000010000001
0101111100111100
1010101010101010
1100000010101010
...
```

load and execute

Computer



L'assembleur ...

- relie la plate-forme matérielle et la hiérarchie logicielle.
- est l'échelon le plus bas de l'ensemble des traducteurs
- est un exemple simple de techniques clés de génie logiciel (analyse syntaxique, génération de code, tables de symboles, ...)

Traduction d'un programme

Symbolic code

```
// Computes R1=1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i>R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    // sum += i
    @i
    D=M
    @sum
    M=D+M
    // i++
    @i
    M=M+1
    @LOOP
    0;JMP
(STOP)
    @sum
    D=M
    ...
```

Translate

Besoin de gérer :

- Les espaces et commentaires
- Les instructions
- Les symboles

Assembleur avec symboles

Binary code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
...
```

Traitement des symboles

Symbolic code

```
// Computes R1=1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i>R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    // sum += i
    @i
    D=M
    @sum
    M=D+M
    // i++
    @i
    M=M+1
    @LOOP
    0;JMP
(STOP)
    @sum
    D=M
    ...
```

L'assembleur de la Hack utilise 23 symboles prédéfinis

symbol	*value*
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

Etiquettes

LOOP	4
STOP	18

Variables

i	16
sum	17

Ce programme utilise deux étiquettes LOOP, STOP et deux variable i, sum

Traitement des symboles

Symbolic code

```
// Computes R1=1 + ... + R0
    // i = 1
    @i
    M=1
    // sum = 0
    @sum
    M=0
(LOOP)
    // if i>R0 goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    // sum += i
    @i
    D=M
    @sum
    M=D+M
    // i++
    @i
    M=M+1
    @LOOP
    0;JMP
(STOP)
    @sum
    D=M
    ...
```

Symbol table

symbol	*value*
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
LOOP	4
STOP	18
i	16
sum	17

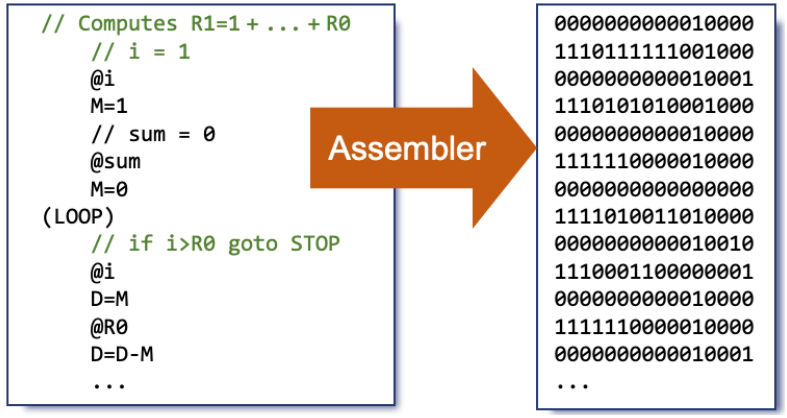
```
// Computes R1=1 + ... + R0
    // i = 1
    @16
    M=1
    // sum = 0
    @17
    M=0

    // if i>R0 goto STOP
    @16
    D=M
    @0
    D=D-M
    @18
    D;JGT
    // sum += i
    @16
    D=M
    @17
    M=D+M
    // i++
    @16
    M=M+1
    @4
    0;JMP
    @17
    D=M
    ...
```

no symbols

Assembleur

Entrée (Prog.asm) : un fichier texte contenant une séquence de lignes, chacune étant une chaîne représentant un commentaire, une instruction A, une instruction C ou une déclaration d'étiquette



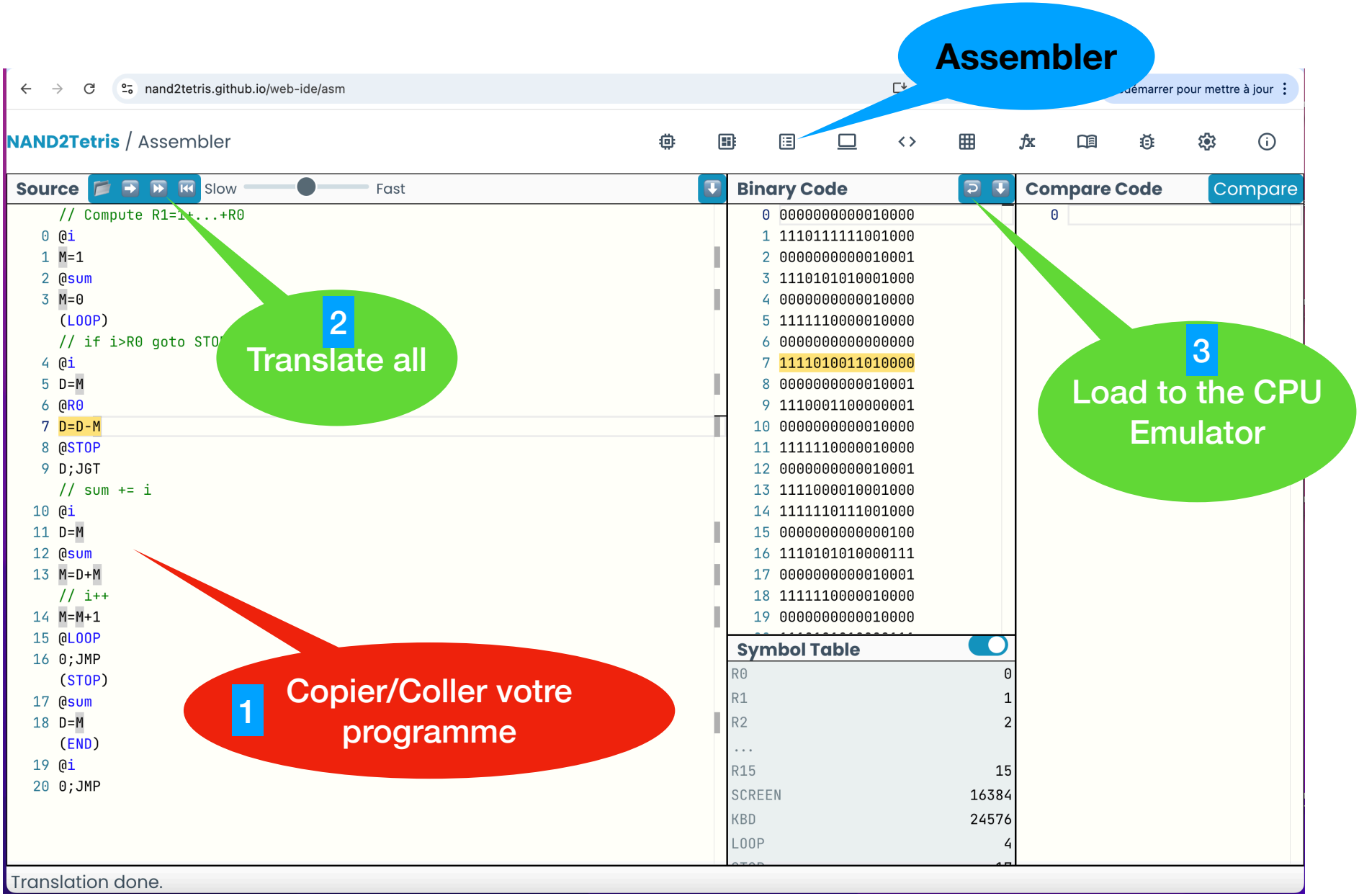
Sortie (Prog.hack) : un fichier texte contenant une séquence de lignes, chacune étant une chaîne de seize caractères 0 et 1

Utilisation : (si l'assembleur est implémenté en Java)
$ java HackAssembler Prog.asm

Action : Crée un fichier Prog.hack, contenant le programme Hack traduit.

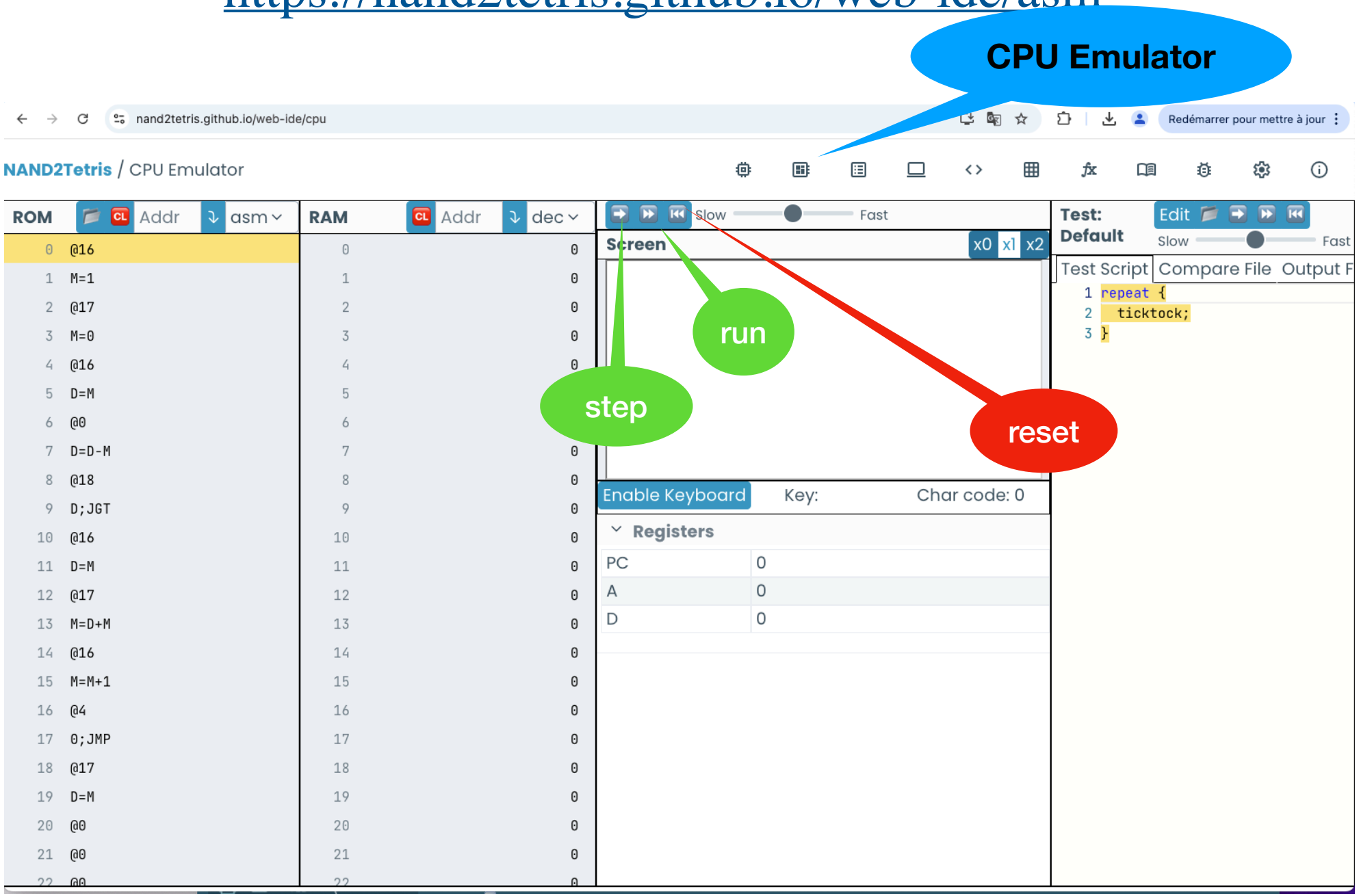
Utilisation du simulateur

https://nand2tetris.github.io/web-ide/asm



Utilisation du simulateur

https://nand2tetris.github.io/web-ide/asm



Démo !

Ecrire de l'assembleur

- **Ecrire en assembleur directement dans le simulateur**

  - comprendre l'assembleur

- **Ecrire en assembleur dans un fichier texte**

  - écrire un mini programme pour comprendre comment combiner les instruction

- **Ecrire un programme Python qui génère de l'assembleur**

  - évite les copier/coller

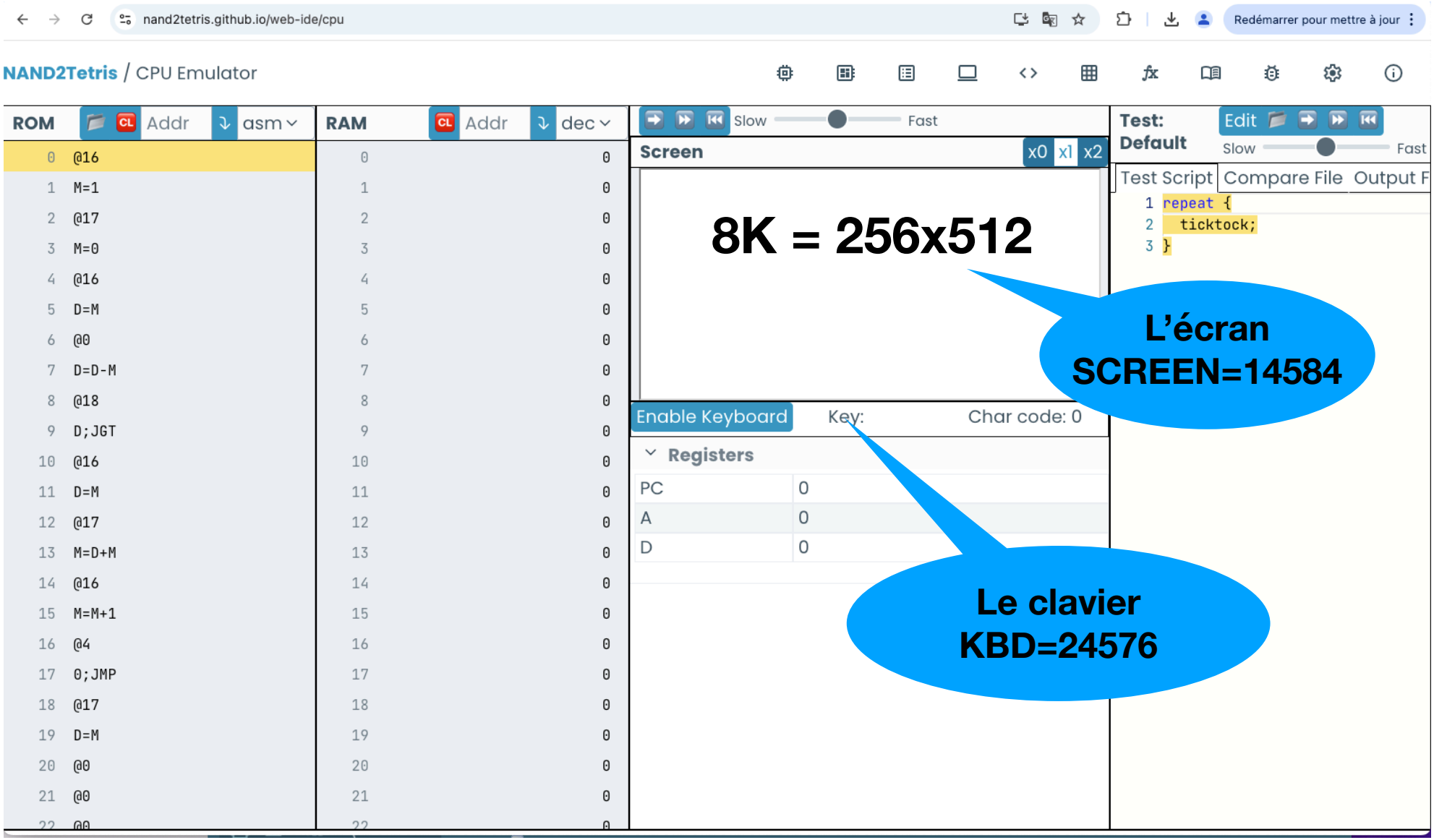
  - permet d'écrire de plus gros programme

- **Ecrire un compilateur en Python générant de l'assembleur**

  - objet du cours

Démo !

Le clavier et l'écran



Chaque ligne de l'écran est codé par 32 mots (16 bits)

Démo !

Quoi faire !

- **Ecrire en assembleur directement dans le simulateur**

  - comprendre l'assembleur

- **Ecrire en assembleur dans un fichier texte**

  - écrire un mini programme pour comprendre comment combiner les instruction

- **Ecrire un programme Python qui génère de l'assembleur**

  - évite les copier/coller

  - permet d'écrire de plus gros programme

- **Ecrire un compilateur en Python générant de l'assembleur**

  - objet du cours