

## 目次

- 一般的なアセンブラーの話と hack ではどうかという話(いらない気もするがあったのでやった)
- hack でどうするか、そのためには、hack assembly の理解をする必要があるなあと(長い道のり)
- hack assembler の話
- 課題の話

## アセンブラー

- 機械語のコードを実行できるコンピュータを作りました。
- アセンブリ言語でプログラミングをしてました、
- その隙間にアセンブラを作りましょ。
- 初めてのソフトウェアですよ（ついに）

## Assembly process

Assembly Language

```
@i  
M=1 // i = 1  
@sum  
M=0 // sum = 0  
(LOOP)  
@i // if i>RAM[0]  
D=M // GOTP WRITE  
@R0  
D=D-M  
@WRITE  
D;JGT  
... // Etc.
```

Machine Language

```
000000000010000  
1110111111001000  
0000000000010001  
1110101010001000  
0000000000010000  
1111110000010000  
0000000000000000  
1111010011010000  
0000000000010010  
1110001100000001  
0000000000010000  
1111110000010000  
0000000000010001  
...
```

assembler

run



アセンブラーをどこで実行するか

- クロスコンパイルするぜ
- アセンブラーをマシンコードで書かなくて良い(多分そういうこと)
- 別のコンピュータすでに実装されている高レベルの言語で書くことができる

一般的にアセンブリーをどのように読んでいくか

1. ファイルを上から下に読んでいくが、スペースとかコメントは無視するようとする

## step 2 要素ごとの分割

- string を要素ごとに分ける

### Basic Assembler Logic

---

Repeat:

- Read the next Assembly language command
- Break it into the different fields it is composed of

L	o	a	d	R	1	,	1	8
---	---	---	---	---	---	---	---	---

L	o	a	d
---	---	---	---

R	1
---	---

1	8
---	---

step 3 対応するハイブリーを探す

- それぞれの要素を読み込むぜ
- それに対応する binary に書き換える

## Basic Assembler Logic

---

Repeat:

- Read the next Assembly language command
- Break it into the different fields it is composed of
- Lookup the binary code for each field

L | o | a | d

R | 1

1 | 8

Command	Opcode
Add	10010
Load	11001
...	...

## step 4 — フラグ binary によじめて山下する

- これでプログラムは実行することができるようになる
- 

## Basic Assembler Logic

---

Repeat:

- Read the next Assembly language command
- Break it into the different fields it is composed of
- Lookup the binary code for each field
- Combine these codes into a single machine language command

1 | 1 | 0 | 0 | 1

0 | 1

0 | 0 | 0 | 1 | 0 | 0 | 1 | 0

1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0

## シンボル どうする問題

- ラベルシンボル
  - ユーザが定義するシンボルであり、プログラムの特定のアドレスにジャンプするための変数(goto に使われる)
- 変数シンボル
  - ユーザ定義したシンボルの中で、事前に言語に定義されておらず、かつ ラベルシンボルで定義されてない奴、それは変数として扱われ、このシンボルはアセンブラによって一位のメモリアドレスが与えられる。

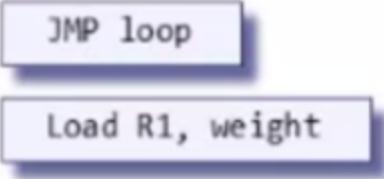
## シンボルどうする問題

### Symbols

---

- Used for:

- Labels



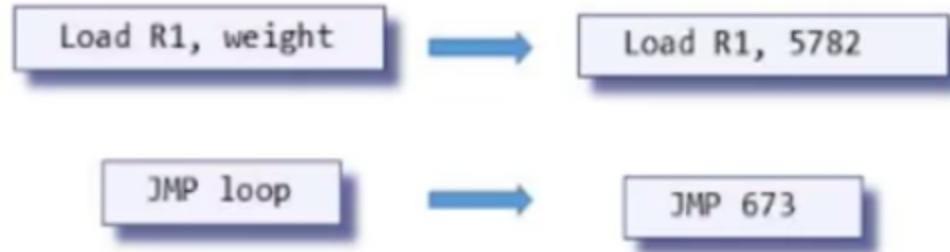
JMP loop

- Variables

Load R1, weight

# Symbol Table

---



	Symbol	Address
Prog	weight	5782
670	loop	673
	...	
	...	
loop	...	
	...	
	...	
	...	

## 変数シンボルのアロケーション

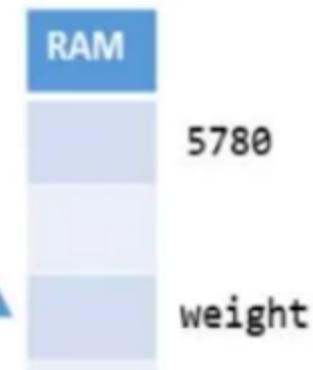
- 使えるメモリを探す
- その値を symboltable に保存していく

### Allocation of variables

Load R1, weight

Find an unallocated  
memory cell

Symbol	Address
weight	5782



- ラベルは実行するコマンドではない。
  - ループとかで呼ばれるので、アドレスを覚えている必要がある
- 

```
--  
Load R1, 30  
Label loop:  
Add 1, R1  
  
---  
JMP loop  
---
```

?

JMP 673

Prog	Symbol	Address
670	...	
...		
...		
loop	loop	673
...		
...		
...		

ジャンプが先呼ばれるとどうする問題

## Forward references

---

```
-- JGT cont  
-- Label cont:  
--
```

Symbol	Address

Possible Solutions:

- Leave blank until label appears, then fix
- In first pass just figure out all addresses

# The translator's challenge

## Hack assembly code

(source language)

```
// Computes RAM[1]=1+...+RAM[0]
@i
M=1    // i = 1
@sum
M=0    // sum = 0

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

Assembler

## Hack binary code

(target language)

```
000000000010000
1110111111001000
000000000010001
1110101010001000
000000000010000
1111110000010000
0000000000000000
1111010011010000
00000000000010010
1110001100000001
000000000010000
1111110000010000
000000000010001
1111000010001000
0000000000010000
1111110111001000
000000000000100
1110101010000111
...
```

Based on the syntax  
rules of:

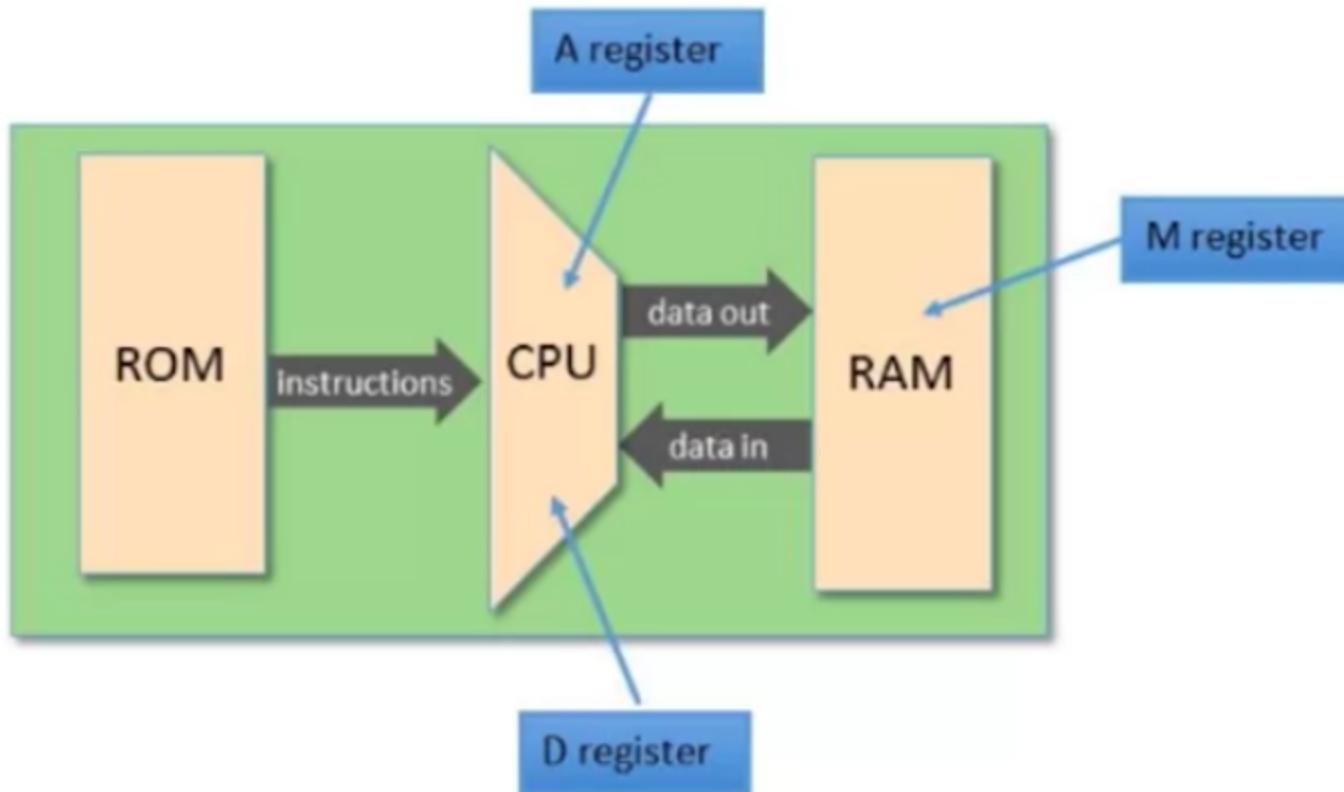
- The source language
- The target language

## 俺たちの Hack assembly どんなんだっけ

- 任意のコードを変換しないといけない（それはそうだがむずそう）
- 両方の文法を知らないといけないよね
- Hack aseembly は A 命令 C 命令 symbol によって構成されているぜ

# Hack computer: registers

---



The Hack machine language recognizes three registers:

- D holds a 16-bit value
- A holds a 16-bit value
- M represents the 16-bit RAM register addressed by A

A 命令

## Assembly 文法

- @で始まり、
  - 非負数の 10 進数で表されるか @20
  - シンボルで表される @tmp

A レジスタに 15 ビットの値を設定するために使える。つまり、特定の値を A レジスタに値を設定することができる

# できること

1. A レジスタを用いて定数を代入する方法(プログラムによって定数を代入する方法は A 命令以外できない)
2. メモリ操作に使う(A レジスタにメモリアドレスを指定することでその後に続く、C 命令に置いて、A レジスタでしたいしたメモリ位置にあるデータを操作することができる)
3. その後に移動するための C 命令を用いることで、jump するアドレスを指定することができる

メモリを操作する前には A 命令をする必要がありますよ

# 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

Usage example:

```
// Set RAM[100] to -1  
@100 // A=100
```

## A 命令

---

Symbolic syntax:

`@value`

Example:

`@21`

Where *value* is either

- a non-negative decimal constant or
- a symbol referring to such a constant

Binary syntax:

`0 valueInBinary`

Example:

`000000000010101`

## C 命令

基本的にやりたいこと

- 計算の結果をどこに保存するか、
- ジャンプするかどうかを決めるかどうか

c 命令と A 命令と一緒に用いることで、コンピュータで行う全ての命令を実行することができる

## C 命令

- destination、computation、およびjump の領域で構成されるよ
- assembly ではこんな感じで表現されるぜ

### The C-instruction

---

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

where:

*comp* =  $\theta, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D\&A, D|A$   
 $M, !M, -M, M+1, M-1, D+M, D-M, M-D, D\&M, D|M$

*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* ≠ 0) jump to execute  
the instruction in ROM[A]

dest のれい

## destの仕様

3つの d ビットを以下のように割り当てる。

d1	d2	d3	ニーモニック	保存先(計算された値を格納する場所)
0	0	0	null	値はどこにも格納されない
0	0	1	M	Memory[A] (メモリ中のアドレスがAの場所)
0	1	0	D	Dレジスタ
0	1	1	MD	Memory[A]とDレジスタ
1	0	0	A	Aレジスタ
1	0	1	AM	AレジスタとMemory[A]
1	1	0	AD	AレジスタとDレジスタ
1	1	1	AMD	AレジスタとMemory[A]とDレジスタ

## Comp 領域の仕様

- 7 ビットされるコマンドを実行するができるぜ
- destination、computation、および jump の領域で構成されるよ
- assembly ではこんな感じで表現されるぜ

### The C-instruction

---

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

where:

*comp* =  $\begin{matrix} 0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D\&A, D|A \\ M, \quad !M, \quad -M, \quad M+1, \quad M-1, \quad D+M, \quad D-M, \quad M-D, \quad D\&M, \quad D|M \end{matrix}$

*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]

## jump

3つの j ビットを以下のように割り当てる。

d1	d2	d3	ニーモニック	保存先(計算された値を格納する場所)
0	0	0	null	値はどこにも格納されない
0	0	1	M	Memory[A] (メモリ中のアドレスがAの場所)
0	1	0	D	Dレジスタ
0	1	1	MD	Memory[A]とDレジスタ
1	0	0	A	Aレジスタ
1	0	1	AM	AレジスタとMemory[A]
1	1	0	AD	AレジスタとDレジスタ
1	1	1	AMD	AレジスタとMemory[A]とDレジスタ

# The C-instruction

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

where:

*comp* =  $0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D\&A, D|A$   
 $M, \quad |M, \quad -M, \quad M+1, \quad M-1, \quad D+M, \quad D-M, \quad M-D, \quad D\&M, \quad D|M$

*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].

Example:

```
// If (D-1==0) jump to execute the instruction stored in ROM[56]
@56      // A=56
```

# 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].

Example:

```
// Set the D register to -1
D=-1
```

# 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].

## Example:

```
// Set RAM[300] to the value of the D register minus 1
@300    // A=300
M-D-1    // RAM[300]=D-1
```

# 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].

Example:

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

```
@56 // A=56
```

## C 命令

- 3つの要素で構成されている
- string processing ができるよ
- テーブルから対応している binary を見つけてこよう
- 高級言語で 0 と 1 のテキストファイルを吐き出して、それをパソコンに読んでもらおう

# C 命令

## Hack language specification: C-instruction

Symbolic syntax: *dest = comp ; jump*

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

<i>comp</i>		<i>c1</i>	<i>c2</i>	<i>c3</i>	<i>c4</i>	<i>c5</i>	<i>c6</i>
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						

<i>dest</i>	<i>d1</i>	<i>d2</i>	<i>d3</i>	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

<i>jump</i>	<i>j1</i>	<i>j2</i>	<i>j3</i>	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

## A レジスタの衝突

- A レジスタは c 命令においてデータメモリの位置を指定するのに使う(M と一緒に使う)
- A レジスタは c 命令によって命令メモリの位置を指定するのに使う(jump と一緒に使う)

jmp をする C 命令において M を参照するべきではない

# シンボル again

## シンボル

特定のアドレスにつけるラベルのこと。以下の3種類がある。

### 定義済みシンボル

- 仮想レジスタ: R0 ~ R15 はRAMアドレスの0から15を参照する。
- 定義済みポインタ: SP , LCL , ARG , THIS はRAMアドレスの0から4を参照する。
- 入出力ポインタ: SCREEN と KBD はRAMアドレスの16384(0x4000)と24576(0x6000)を参照する。

### ラベルシンボル。

ジャンプを伴うC命令飛び先となるアドレスにつけるラベル。 (xxx) と書く。

### 変数シンボル

アセンブラーによって一意のメモリアドレスが割り振られる。16(0x0010)から始まる。

# Handling symbols

## Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
@i
M=1    // i = 1
@sum
M=0    // sum = 0

(LOOP)
@i      // if i>RAM[0] goto STOP
D=M
@R0
D=D-M
@STOP
D;JGT
@i      // sum += i
D=M
@sum
M=D+M
@i      // i++
M=M+1
@LOOP // goto LOOP
0;JMP

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

- assembler ではこのアドレスの割り当てをやってあげないといけない

## Handling symbols

### Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
@i
M=1    // i = 1
@sum
M=0    // sum = 0

(LOOP)
@i      // if i>RAM[0] goto STOP
D=M
@R0
D=D-M
@STOP
D;JGT
@i      // sum += i
D=M
@sum
M=D+M
@i      // i++
M=M+1
@LOOP // goto LOOP
0;JMP
(STOP)
@sum
D=M
```

### Symbols:

- **variable symbols:**  
represent memory locations where the programmer wants to maintain values
- **label symbols:**  
represent destinations of goto instructions
- **pre-defined symbols:**  
represent special memory locations

# Handling symbols

---

## Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
@i
M=1    // i = 1
@sum
M=0    // sum = 0

(LOOP)
@i      // if i>RAM[0] goto STOP
D=M
@R0
D=D-M
@STOP
D;JGT
@i      // sum += i
D=M
@sum
M=D+M
@i      // i++
M=M+1
@LOOP // goto LOOP
0;JMP

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

## Symbols:

- **variable symbols:**  
represent memory locations where the programmer wants to maintain values
- **label symbols:**  
represent destinations of goto instructions
- **pre-defined symbols:**  
represent special memory locations

# Handling symbols

---

## Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
@i
M=1    // i = 1
@sum
M=0    // sum = 0

(LOOP)
@i      // if i>RAM[0] goto STOP
D=M
@R0
D=D-M
@STOP
D;JGT
@i      // sum += i
D=M
@sum
M=D+M
@i      // i++
M=M+1
@LOOP // goto LOOP
0;JMP

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

## Symbols:

- **variable symbols:**  
represent memory locations where the programmer wants to maintain values
- **label symbols:**  
represent destinations of goto instructions
- **pre-defined symbols:**  
represent special memory locations

- 定義された記号を、対応する値（10進数）に置き換える
- 10進数を2進数に置き換える（それはさっき話した）

## Handling pre-defined symbols

---

Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
@i
M=1    // i = 1
@sum
M=0    // sum = 0

(LOOP)
@i      // if i>RAM[0] goto STOP
D=M
@R0
D=D-M
@STOP
D;JGT
@i      // sum += i
D=M
@sum
M=D+M
@i      // i++
M=M+1
@LOOP // goto LOOP
0;JMP
(STOP)
@sum
D=M
@R1
M=D // RAM[1] = the sum
(END)
@END
```

The Hack language specification  
describes 23 *pre-defined symbols*:

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

Translating @preDefinedSymbol :  
Replace *preDefinedSymbol* with its value.

置き換えることになります

- 命令番号を記録
- empty line ignore
- 初めて見たらメモリをフリ当てよう
- そして、アドレスに置き換えていきましょう
- @LOOP -> @4 になる

## Handling symbols that denote variables

---

### Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
@i
M=1    // i = 1
@sum
M=0    // sum = 0

(LOOP)
@i      // if i>RAM[0] goto STOP
D=M
@R0
D=D-M
@STOP
D;JGT
@i      // sum += i
D=M
@sum
M=D+M
```

### Variable symbols

- Any symbol xxx appearing in an assembly program which is not pre-defined and is not defined elsewhere using the (xxx) directive is treated as a *variable*
- Each variable is assigned a unique memory address, starting at 16

<u>symbol</u>	<u>value</u>
i	16

で、変数はアドレス 16 以降のメモリに代入されます

- このようなシンボルが活躍するのは... A 命令のコンテキストだけです。
- プログラムの中でこの命令を初めて見たときに、この変数がプログラムの中に初めて現れたら、それをアドレス 16 以降のメモリアドレスに割り当てます

## Handling symbols that denote variables

### Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
@i
M=1    // i = 1
@sum
M=0    // sum = 0

(LOOP)
@i      // if i>RAM[0] goto STOP
D=M
@R0
D=D-M
@STOP
D;JGT
@i      // sum += i
D=M
@sum
M=D+M
@i      // i++
M=M+1
```

### Variable symbols

- Any symbol xxx appearing in an assembly program which is not pre-defined and is not defined elsewhere using the (XXX) directive is treated as a *variable*
- Each variable is assigned a unique memory address, starting at 16

symbol	value
i	16
sum	17

- pre defined のやつで埋めましょう

## Symbol table

---

Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
@i
M=1    // i = 1
@sum
M=0    // sum = 0

(LOOP)
@i    // if i>RAM[0] goto STOP
D=M
@R0
D=D-M
@STOP
D;JGT
@i    // sum += i
D=M
@sum
M=D+M
@i    // i++
M=M+1
@LOOP // goto LOOP
0;JMP
(STOP)
@sum
D=M
@R1
M=D // RAM[1] = the sum
(END)
```

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

**Initialization:**  
Add the pre-defined symbols

- ラベルを埋めましょう

## Symbol table

---

Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
@i
M=1    // i = 1
@sum
M=0    // sum = 0

(LOOP)
@i    // if i>RAM[0] goto STOP
D=M
@R0
D=D-M
@STOP
D;JGT
@i    // sum += i
D=M
@sum
M=D+M
@i    // i++
M=M+1
@LOOP // goto LOOP
0;JMP
(STOP)
@sum
D=M
@R1
M=D // RAM[1] = the sum
(END)
```

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

**Initialization:**

Add the pre-defined symbols

- この時点で symbol-table にない奴は symbol だ

## Symbol table

---

Assembly program

```
// Computes RAM[1] = 1 + ... + RAM[0]
@i
M=1    // i = 1
@sum
M=0    // sum = 0

(LOOP)
@i    // if i>RAM[0] goto STOP
D=M
@R0
D=D-M
@STOP
D;JGT
@i    // sum += i
D=M
@sum
M=D+M
@i    // i++
M=M+1
@LOOP // goto LOOP
0;JMP
(STOP)
@sum
D=M
@R1
M=D // RAM[1] = the sum
(END)
```

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

**Initialization:**

Add the pre-defined symbols

# The assembly process

---

## Initialization:

- Construct an empty symbol table
- Add the pre-defined symbols to the symbol table

## First pass:

Scan the entire program;

For each “instruction” of the form (xxx):

- Add the pair (xxx, *address*) to the symbol table,  
where *address* is the number of the instruction following (xxx)

## Second pass:

Set *n* to 16

Scan the entire program again; for each instruction:

- If the instruction is @*symbol*, look up *symbol* in the symbol table;
  - If (*symbol*, *value*) is found, use *value* to complete the instruction’s translation;
  - If not found:
    - Add (*symbol*, *n*) to the symbol table,
    - Use *n* to complete the instruction’s translation,
    - *n++*
- If the instruction is a C-instruction, complete the instruction’s translation
- Write the translated instruction to the output file

ヘア、xxx、アドレスをシンボルテーブルに追加していきます。

2.

## The assembly process

---

### Initialization:

- Construct an empty symbol table
- Add the pre-defined symbols to the symbol table

### First pass:

Scan the entire program;

For each “instruction” of the form (xxx):

- Add the pair  $(xxx, address)$  to the symbol table,  
where *address* is the number of the instruction following (xxx)

### Second pass:

Set *n* to 16

Scan the entire program again; for each instruction:

- If the instruction is  $@symbol$ , look up *symbol* in the symbol table;
  - If  $(symbol, value)$  is found, use *value* to complete the instruction’s translation;
  - If not found:
    - Add  $(symbol, n)$  to the symbol table,
    - Use *n* to complete the instruction’s translation,

- 入力を読み込んで、分割していこう
  - 指定されたファイルを読み込んでいこう
  - 読み込んだファイルの次のコマンドを毎回取得できるよう
  - 読み込んだコマンドをコンポーネントに分割していこう
- 

## Reading and Parsing Commands

---

- Start reading a file with a given name
  - E.g. Constructor for a **Parser** object that accepts a string specifying a file name.
  - Need to know how to read text files

## Reading and Parsing Commands

---

- Start reading a file with a given name
- Move to the next command in the file
  - Are we finished? **boolean hasMoreCommands();**
  - Get the next command. **void advance();**
  - Need to read one line at a time
  - Need to skip whitespace including comments

## Reading and Parsing Commands

---

- Start reading a file with a given name
- Move to the next command in the file
- Get the fields of the current command
  - Type of current command (A-Command, C-Command, or Label)
  - Easy access to the fields:

D = M+1; JGT

@sum



# C 命令のやり方

## Translating Mnemonic to Code: computation

Symbolic syntax:

*dest* = *comp* ; *jump*

Binary syntax:

1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3

<i>comp</i>		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						

## The Symbol Table

---

Symbol	Address
loop	73
sum	12

- Create a new empty table
- Add a (symbol,address) pair to the table
- Does the table contain a given symbol?
- What is the address associated with a given symbol? ↗

# symbol table

## Using the Symbol Table

---

- Create a new empty table
- Add all the pre-defined symbols to the table
- While reading the input, add labels and new variables to the table
- Whenever you see a “@xxx” command, where xxx is not a number, consult the table to replace the symbol xxx with its address.

## The overall assembly logic

### Assembly program

```
@16  
M=1  
@17  
M=0  
@16  
D=M  
@0  
D=D-M  
@18  
D;JGT  
@16  
D=M  
@17  
M=D+M  
@16  
M=M+1  
@4  
0;JMP  
@17  
D=M  
@1  
M=D  
@22  
0;JMP
```

### For each instruction

- Parse the instruction:  
break it into its underlying fields
- A-instruction:  
translate the decimal value into a binary value
- C-instruction:  
for each field in the instruction, generate the corresponding binary code;
- Assemble the translated binary codes into a complete 16-bit machine instruction
- Write the 16-bit instruction to the output file.

## Overall logic

---

- Initialization
  - Of Parser
  - Of Symbol Table
- First Pass: Read all commands, only paying attention to labels and updating the symbol table
- Restart reading and translating commands
- Main Loop:
  - Get the next Assembly Language Command and parse it
  - For A-commands: Translate symbols to binary addresses
  - For C-commands: get code for each part and put them together
  - Output the resulting machine language command

## Developing a Hack Assembler

---

### Contract

- Develop an *assembler* that translates Hack assembly programs into executable Hack binary code
- The source program is supplied in a text file named `Xxx.asm`
- The generated code is written into a text file named `Xxx.hack`
- Assumption: `Xxx.asm` is error-free

### Usage

```
prompt> java HackAssembler Xxx.asm
```

This command should create a new `Xxx.hack` file that can be executed as-is on the Hack computer.

# まとめ

- hack.txt ファイルが与えられる
- 上からそれぞれのラインを読んでいき
- それが A 命令 ならば - 十進数の値を二進数に変換する
- C 命令 ならば、それぞれの値を 3 つの component にわけて処理をしていこう

最後にこれら全てをまとめる - 各行は 16 個の 0 と 1 の文字列で構成される - 記号のない A 命令と C 命令だけで構成されたプログラムが完成される

- 最後にそれをファイルとして書き出しましょう

## project の概要

The assembler can be implemented in any high-level language

### Proposed software design

- **Parser**: unpacks each instruction into its underlying fields
- **Code**: translates each field into its corresponding binary value
- **SymbolTable**: manages the symbol table
- **Main**: initializes I/O files and drives the process.

# Proposed Implementation

---

## Staged development

- Develop a basic assembler that translates assembly programs without symbols
- Develop an ability to handle symbols
- Morph the basic assembler into an assembler that can translate any assembly program

## Supplied test programs

Add.asm

Max.asm

MaxL.asm

Rectangle.asm

RectangleL.asm

Pong.asm

PongL.asm

## Test program: Add

---

Add.asm

```
// Computes RAM[0] = 2 + 3  
  
@2  
D=A  
@3  
D=D+A  
@0  
M=D
```

### Basic test of handling:

- White space
- Instructions

# Test program: Max

Max.asm

```
// Computes RAM[2] = max(RAM[0],RAM[1])  
  
@R0  
D=M          // D = RAM[0]  
@R1  
D=D-M        // D = RAM[0] - RAM[1]  
@OUTPUT_RAM0  
D;JGT         // if D>0 goto output RAM[0]  
  
// Output RAM[1]  
@R1  
D=M  
@R2  
M=D          // RAM[2] = RAM[1]  
@END  
0;JMP  
  
(OUTPUT_RAM0)  
@R0  
D=M  
@R2  
M=D          // RAM[2] = RAM[0]  
  
(END)  
@END  
0;JMP
```

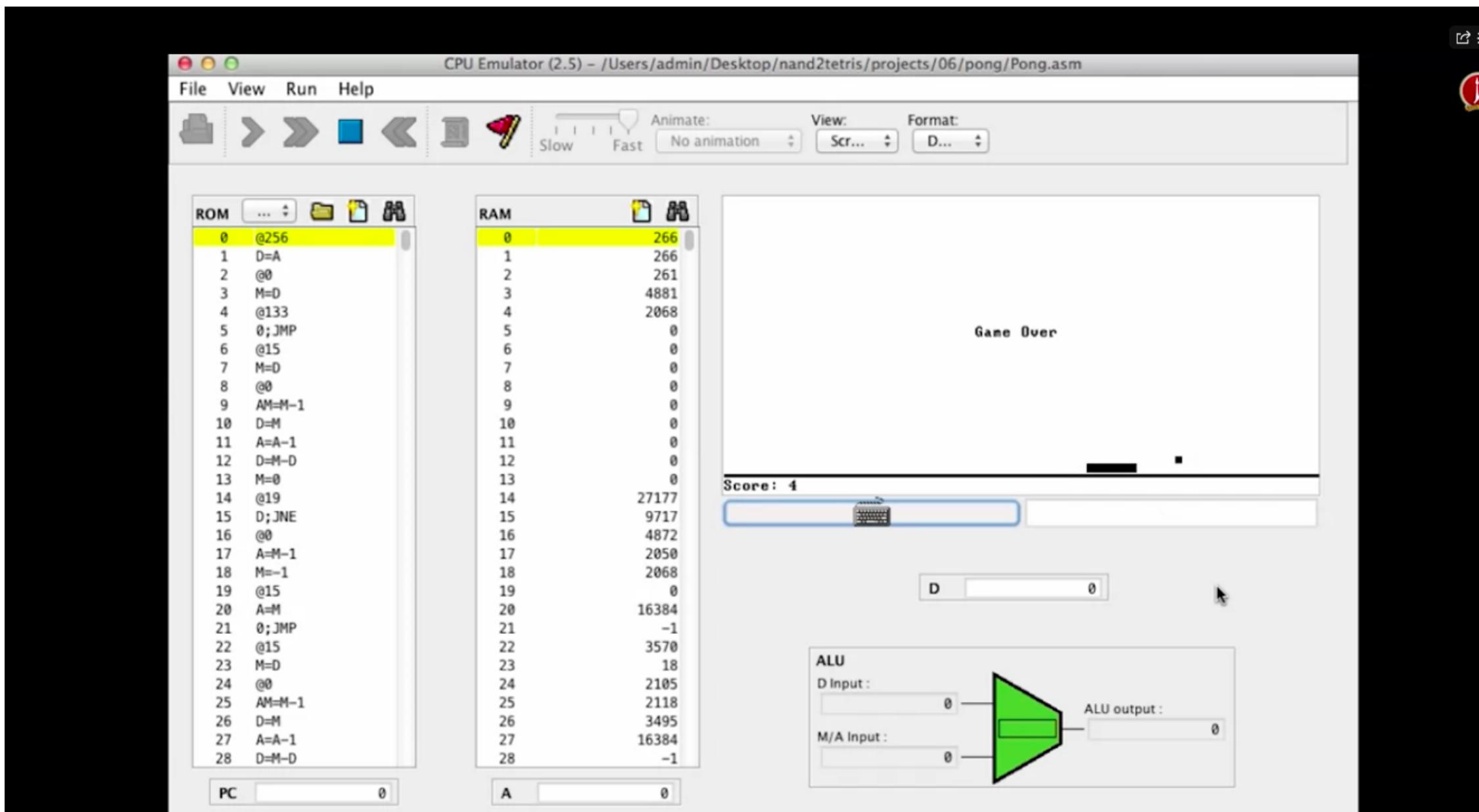
with  
symbols

MaxL.asm

```
// Symbol-less version  
  
@0  
D=M          // D = RAM[0]  
@1  
D=D-M        // D = RAM[0] - RAM[1]  
@12  
D;JGT         // if D>0 goto output RAM[0]  
  
// Output RAM[1]  
@1  
D=M  
@2  
M=D          // RAM[2] = RAM[1]  
@16  
0;JMP  
  
@0  
D=M  
@2  
M=D          // RAM[2] = RAM[0]  
  
@16  
0;JMP
```

without  
symbols

[https://youtu.be/0y8JPx0ZakY?list=PLrDd\\_kMiAuNmSb-CKWQqq9oBFN\\_KNMTal&t=1170](https://youtu.be/0y8JPx0ZakY?list=PLrDd_kMiAuNmSb-CKWQqq9oBFN_KNMTal&t=1170)



おしまい



