

# Assembler

# 目次

- assembler とは
- hack assembly の復習(長い道のり)
- 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 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）

# アセンブラー図

## Assembly process

Assembly Language

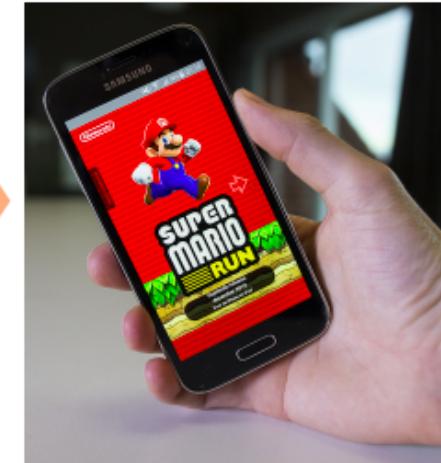
```
@i  
M=1 // i = 1  
@sum  
M=0 // sum = 0  
(LOOP)  
@i // if i>RAM[0]  
D=M // GOTP WRITE  
@R0  
D=D-M  
@WRITE  
D;JGT  
... // Etc.
```

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

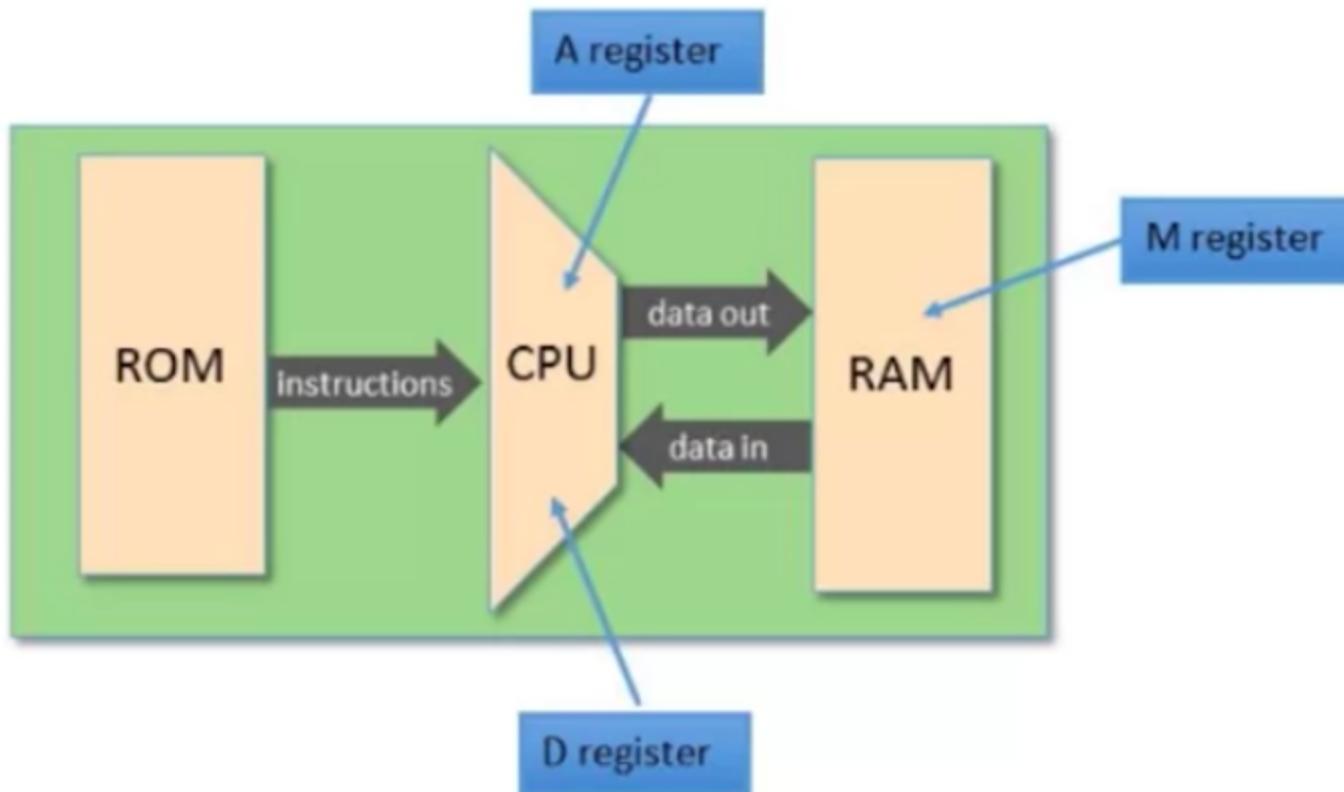
- クロスコンパイルするぜ
- アセンブラーをマシンコードで書かなくて良い(多分そういうこと)
- 別のコンピュータすでに実装されている高レベルの言語で書くことができる

## 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

# 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
(END)
@END
```

## 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

# A 命令

## Assembly 文法

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

# A 命令

## 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
```

## 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 effects:
  - RAM[A] becomes the selected RAM register
  - ROM[A] becomes the selected ROM register

### Example:

```
// Sets A to 17  
@17
```

# A 命令 シンボルでかくと binary 文法

---

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

## 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>		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	

<i>dest</i>	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

<i>jump</i>	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

## 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

## 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* jump 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。

<b>d1</b>	<b>d2</b>	<b>d3</b>	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* =  $0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D\&A, D|M$   
 $M, !M, -M, M+1, M-1, D+M, D-M, M-D, D\&M, D|M$

*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]

# 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:

<i>comp</i> =	0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, M, IM, -M, M+1,	D-1, M-1, A-1, D+A, D-A, A-D, D&A, D M
---------------	---	--

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

*jump* = null, JGT, DEQ, 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
```

# 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]
```

## 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

## 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

## 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
```

- 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

は、LOOP、STOP、END の3つのシンボルがある。

## 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
```

### 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

状況、レジスタ、メモリアドレスなど、定義済みのシンボル。

例では、R0 と R1 がそうである

## 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
```

### 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

## **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にする

## 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 対応するバイナリを揃う

- 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
...	...

- 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 | 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ーによって一位のメモリアドレスが与えられる。

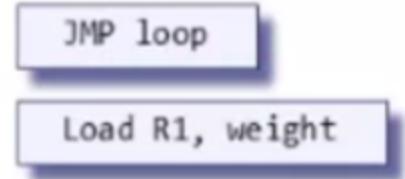
# シンボルどうする問題

## Symbols

---

- Used for:

- Labels



- Variables

# 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

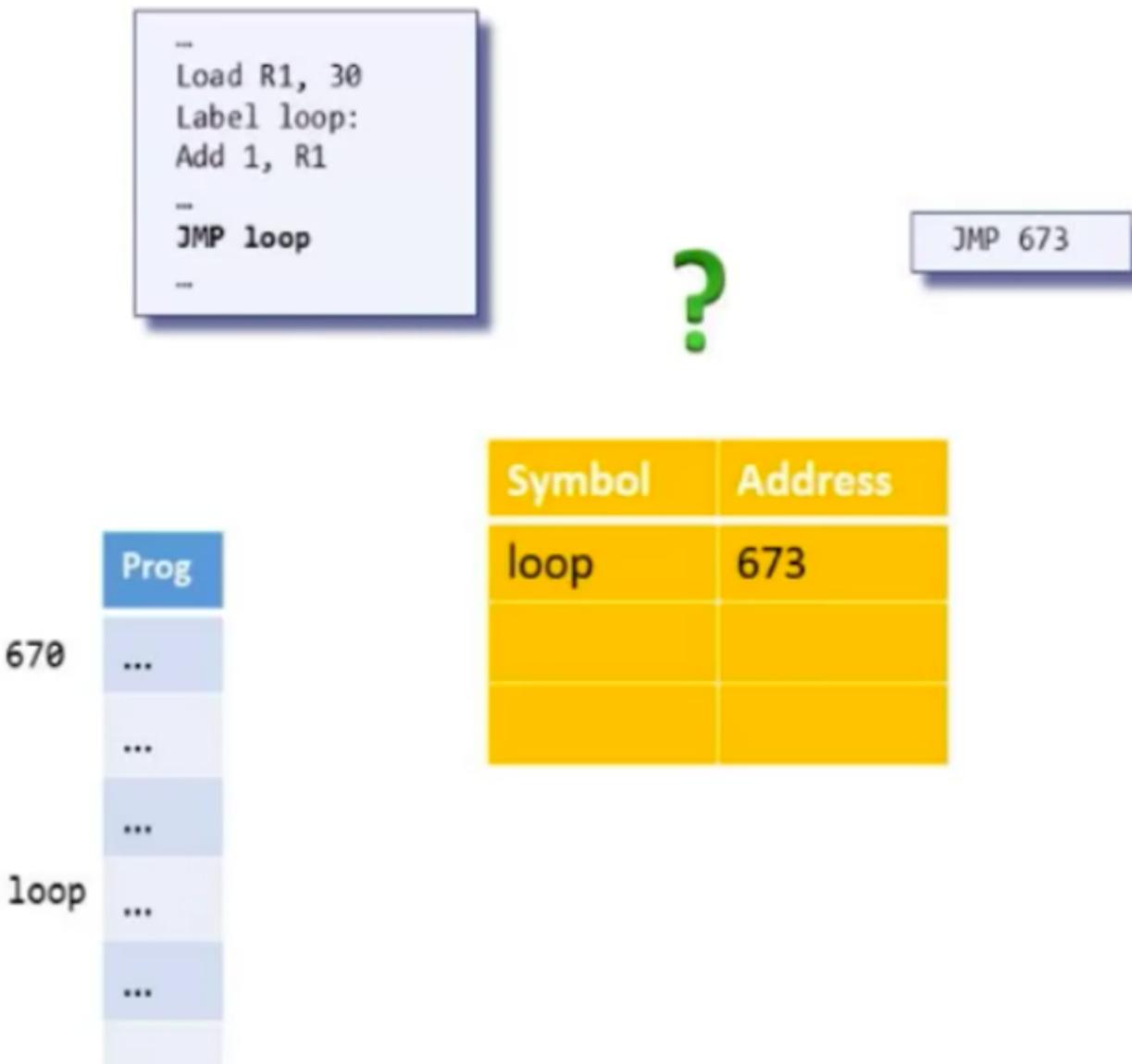
Load R1, weight

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
- 



# 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

- なんか難しそう

## 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
0000000000000000
1111010011010000
0000000000000000
1111110000010000
0000000000000001
1111000010001000
00000000000010010
1110001100000001
000000000010000
1111110000010000
0000000000000001
1111000010001000
00000000000010000
11111110111001000
000000000000000100
1110101010000111
```

Based on the syntax  
rules of:

- The source language
- The target language

# A 命令

---

## Hack language specification: A-instruction

---

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

## 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>		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	

<i>dest</i>	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

<i>jump</i>	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

# Hack language specification: symbols

---

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		

Label declaration:       $(label)$

Variable declaration:     $@variableName$

## アセンブラーで対処しないといけないこと

- インデントとかコメントとか無視しないといけなさそう
- 命令を対処する
  - A, C 命令
- シンボル
  - 変数、ラベル

シンブルの実装が難しそうなので、とりあえずシンボルなしのケースを考えます

# Handling programs without symbols

Assembly program (without symbols)

```
// Computes RAM[1] = 1 + ... + RAM[0]
@16
M=1    // i = 1
@17
M=0    // sum = 0

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

Assembler  
for symbol-less  
Hack programs

Hack machine code

```
000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000001000
1110101010000111
0000000000010001
1111110000010000
0000000000001000
```

## Challenges:

Handling...

- White space
- Instructions

- ただただ、コメントの empty line はアセンブラーで無視するようにしましょう

## Handling white space

Assembly program (without symbols)

```
// Computes RAM[1] = 1 + ... + RAM[0]
@16
M=1    // i = 1
@17
M=0    // sum = 0

@16    // if i>RAM[0] goto STOP
D=M
@0
D=D-M
@18
D;JGT
@16    // sum += i
D=M
@17
M=D+M
@16    // i++
M=M+1
@4     // goto LOOP
0;JMP
@17
D=M
@1
M=D    // RAM[1] = the sum
@22
0;JMP
```

Assembler  
for symbol-less  
Hack programs

### Challenges:

Handling...

- White space
- Instructions

### Handling white space:

Ignore it!

Hack machine code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000001000
1110101010001111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000000000
```

# Handling white space

Assembly program (without symbols)

```
@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
```

Assembler  
for symbol-less  
Hack programs

## Challenges:

Handling...

- ✓ White space
- Instructions

Hack machine code

```
000000000010000  
1110111111001000  
000000000010001  
1110101010001000  
000000000010000  
1111110000010000  
0000000000000000  
1111010011010000  
000000000010010  
1110001100000001  
000000000010000  
1111110000010000  
000000000010001  
1111000010001000  
000000000010000  
1111110111001000  
0000000000000100  
1110101010000111  
000000000010001  
1111110000010000  
0000000000000001  
1110001100001000  
00000000000010110
```

- 15 ビットで表現されるので、足りないところは 0 でうめる
- symbol の場合もありますが、それはまた後で考えましょう

## Translating A-instructions

---

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:

`0000000000010101`

Translation to binary:

- If *value* is a decimal constant

- jump が empty であれば ; は省略

## Translating C-instructions

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 $\geq$ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out $\neq$ 0 jump
JLE	1	1	0	if out $\leq$ 0 jump
JMP	1	1	1	Unconditional jump

symbol ない奴はこれで動く

## 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.

# 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
(END)
@END
```

## 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.

- プログラムの中で XXX が出てきたときには、メモリのアドレスに置き換える
- 命令番号を記録  
アドレスに置き換えていきましょう
- @LOOP -> @4 になる

ex.

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

### 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

Translating @variableSymbol :

- 23このpredefined symbolがありますか、これに対応する数字を二進数に変換しましょう

## 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
```

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

- ()で始まるで () で始まるのには理由があるらしい
  - @Loop とかを @4 と変換しなくてはいけない
- 

## Handling symbols that denote labels

---

### Assembly program

```

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

(LOOP)
4    @i // if i>RAM[0] goto STOP
5    D=M
6    @R0
7    D=D-M
8    @STOP
9    D;JGT
10   @i // sum += i
11   D=M
12   @sum
13   M=D+M
14   @i // i++
15   M=M+1
16   @LOOP // goto LOOP
17   0;JMP
(STOP)

```

### Label symbols

- Used to label destinations of goto commands
- Declared by the pseudo-command (xxx)
- This directive defines the symbol xxx to refer to the memory location holding the next instruction in the program

symbol	value
LOOP	4
STOP	18
END	22

- 値は 16 から始まります。
- 

## 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
@LOOP // goto LOOP
0;JMP
(STOP)
@sum
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
sum	17

### Translating @variableSymbol:

- If you see it for the first time,

# 1. ラベル変数を追加しましょう (ライン番号を数えながら、)

## Symbol table

Assembly program

```

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

(LOOP)
4    @i    // if i>RAM[0] goto STOP
5    D=M
6    @R0
7    D=D-M
8    @STOP
9    D;JGT
10   @i   // sum += i
11   D=M
12   @sum
13   M=D+M
14   @i   // i++
15   M=M+1
16   @LOOP // goto LOOP
17   0;JMP

```

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
END	22

### Initialization:

Add the pre-defined symbols

### First pass:

Add the label symbols

- この時点では 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
```

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

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,

## 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.

## Testing options

---

Use your assembler to translate `Xxx.asm`,  
generating the executable file `Xxx.hack`

Hardware simulator:

load `Xxx.hack` into the Hack Computer chip, then execute it

CPU Emulator:

load `Xxx.hack` into the supplied CPUEmulator, then execute it

Assembler:

use the supplied Assembler to translate `Xxx.asm`;  
Compare the resulting code to the binary code generated by  
*your* assembler.

# 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

# まとめ

- `hack.aasm` 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

- 最後にそれをファイルとして書き出しましょう

## 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
```

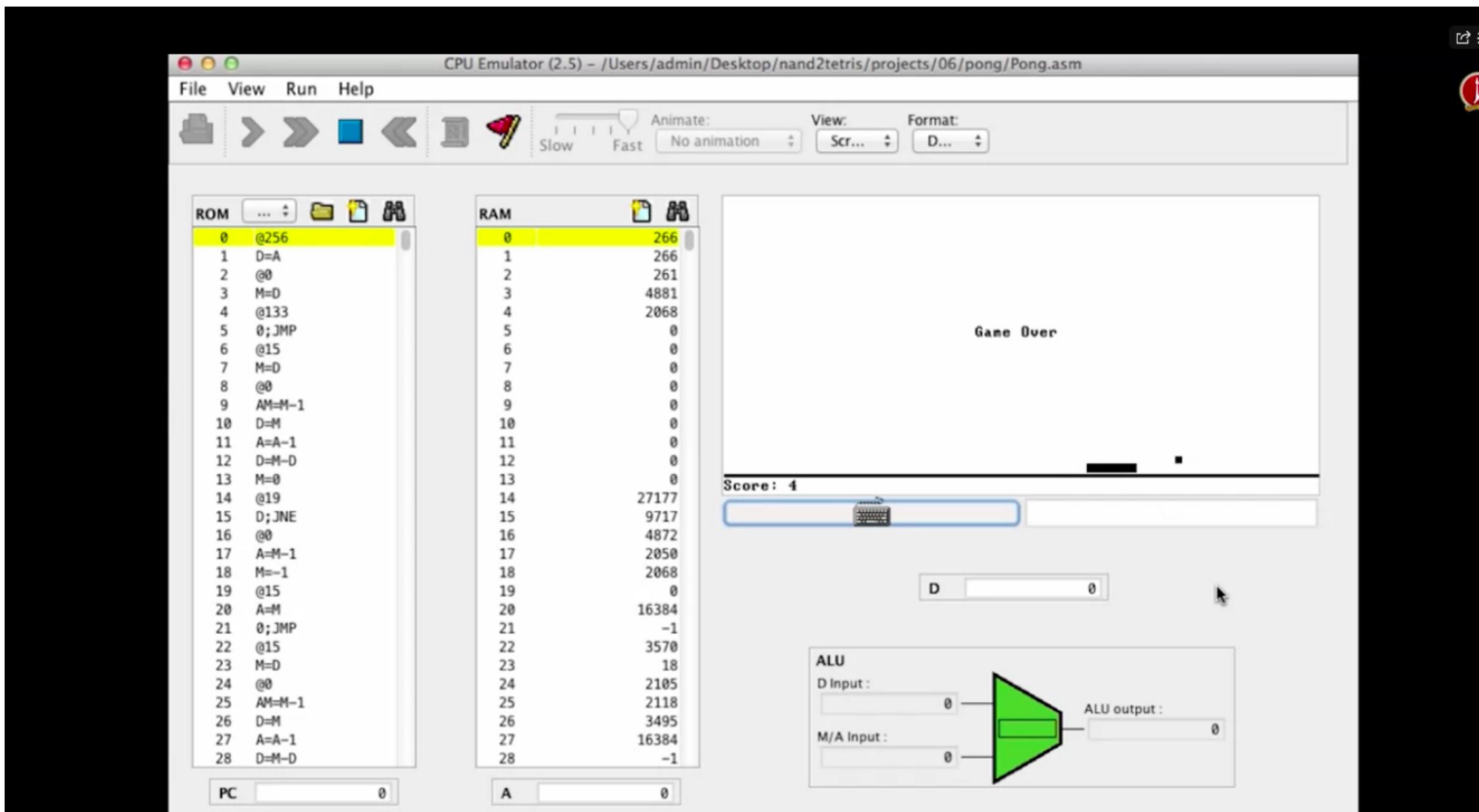
with  
symbols

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



