Unit 0.1

implementation ←

Don't worry about the "how"

only about the "what"

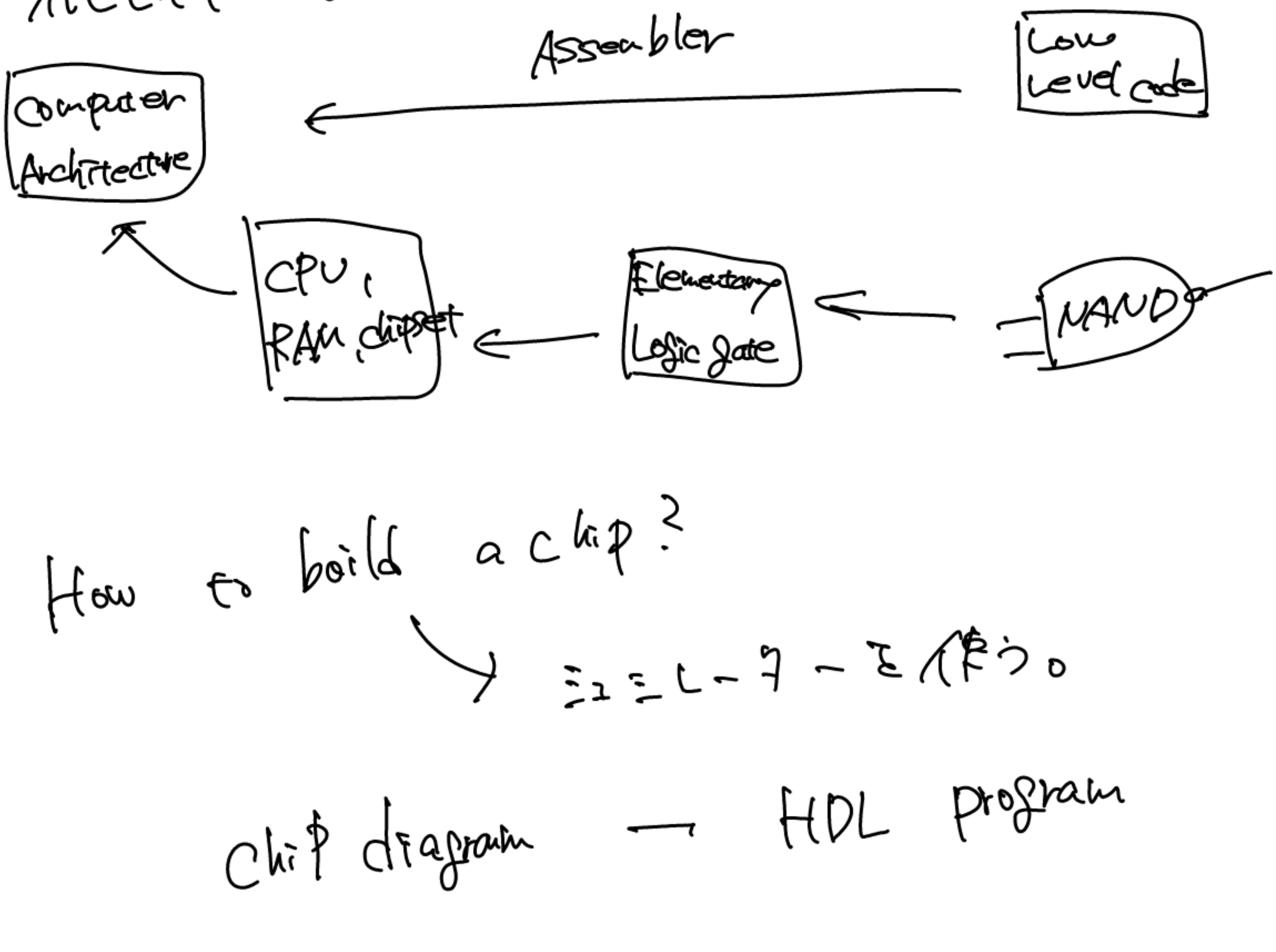
Abstraction

抽象化によって内部実装を知ることなく

コードを書いたりできるが詳細を知ることは

とても大事。

低レイヤーから高レイヤーに移りやりキュラム



How to build a chip?

↘ ハードウェア作る。

Chip diagram → HDL program

Module 1: Boolean functions and Gate Logic

Boolean operations

(x AND y), (x OR y)

x	y	AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

NOT(x)

x	NOT
0	1
1	0

Boolean functions は値が 0, 1 だけなので、
全ての combination を table にできる。

Truth Table

- (x AND (y OR z)) = (x AND y) OR (x AND z)
- (x OR (y AND z)) = (x OR y) AND (x OR z)

↑ Distributive Laws

- NOT (x AND y) = NOT(x) OR NOT(y)
- NOT (x OR y) = NOT(x) AND NOT(y)

← De Morgan Laws

それ以外にも Commutative Laws,

Associative Laws がある。

これらの identity は function を Simple にするのに

使える。例えば…

NOT (NOT(x) AND NOT(x OR y)) =

NOT(NOT(x) AND (NOT(x) AND NOT(y))) =

NOT (NOT(x) AND NOT(x)) AND NOT(y)) =) Idempotence

NOT (NOT(x) AND NOT(y)) =

NOT (NOT(x)) OR NOT(NOT(y)) =) Double Negation

x OR y

Can we make Boolean Expression from Truth Table ?

1. Truth Table で f = 1 の Row を見る
2. シンプルに AND でつなげた式を書く (その Row だけで動く)
3. 他の値が 1 の Row も同様に式を作る
4. 作った式を OR でつなぐ
5. Identity を使って Function を短く

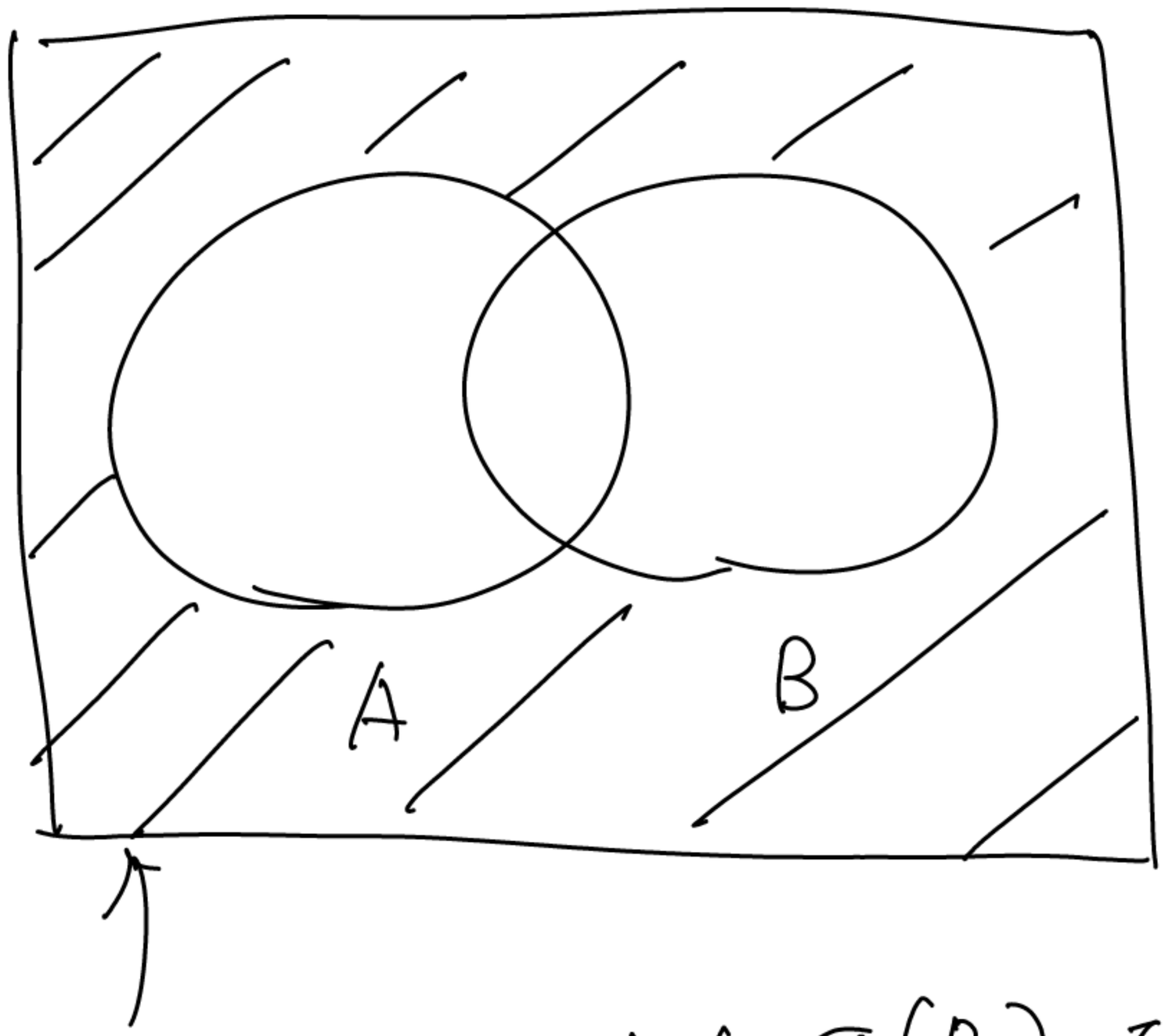
↑

簡単じゃない, NP-Hard

Theorem

Any Boolean function can be represented using an expression containing AND and NOT

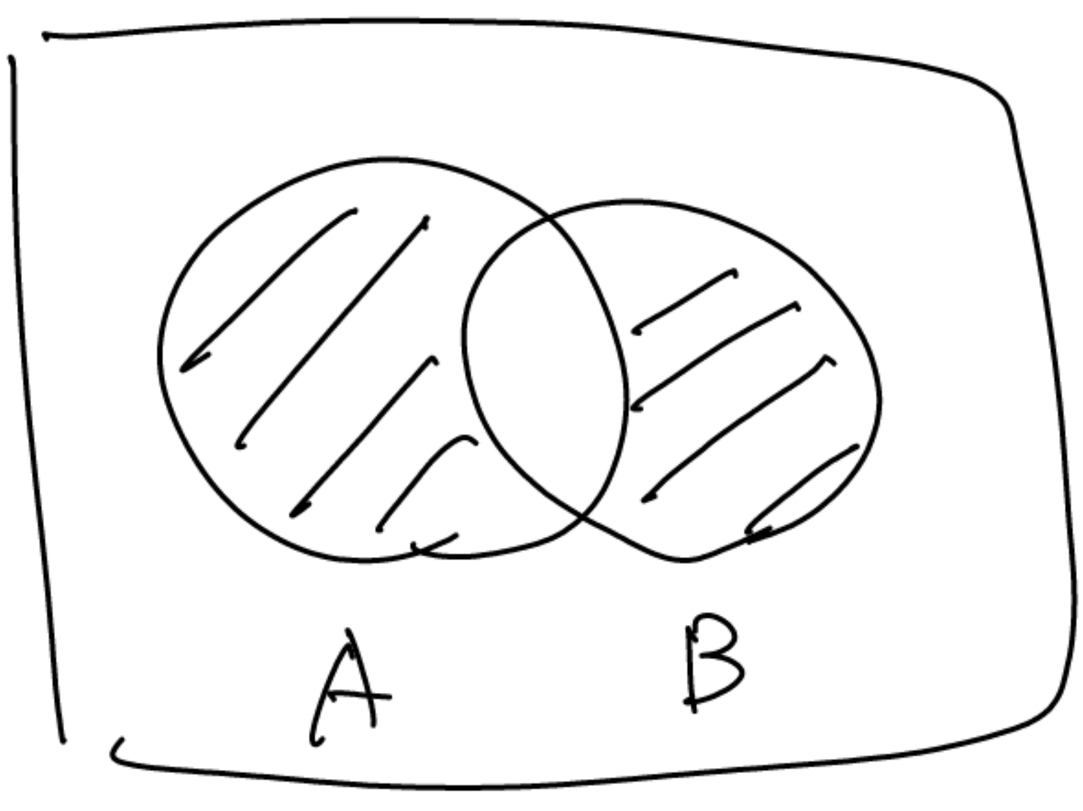
$$(x \text{ OR } y) = \text{NOT}(\text{NOT}(x) \text{ AND } \text{NOT}(y))$$



NOT (A) And NOT (B) Just..

A ⊼ B aOR if NOT (NOT (A) and NOT (B))

Xor



NAND

x	y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

$(x \text{ NAND } y) = \text{NOT}(x \text{ AND } y)$

定理 1:
Any Boolean function can be represented using an expression containig NAND operations.

Proof:

1) $\text{NOT}(x) = (x \text{ NAND } x)$

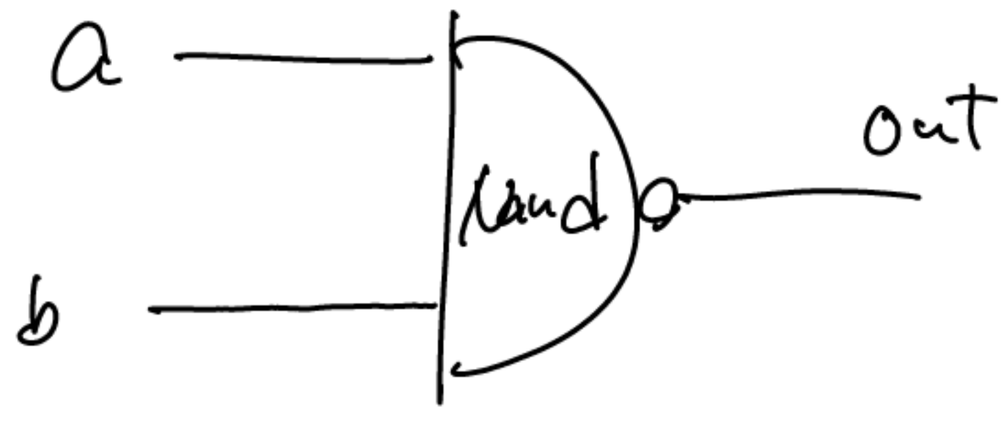
2) $(x \text{ AND } y) = \text{NOT}((x \text{ NAND } y))$

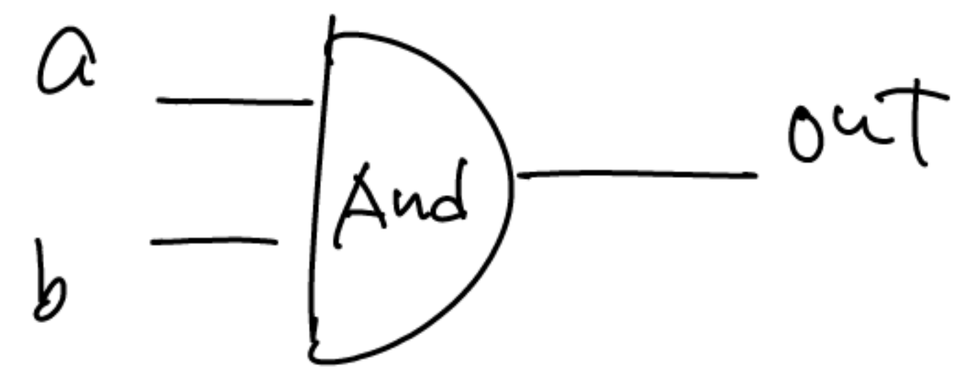
NAND 美しいな〜〜

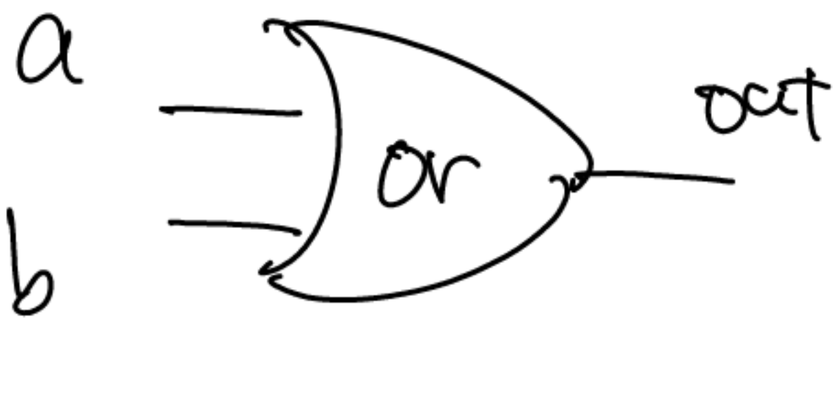
Logic gates

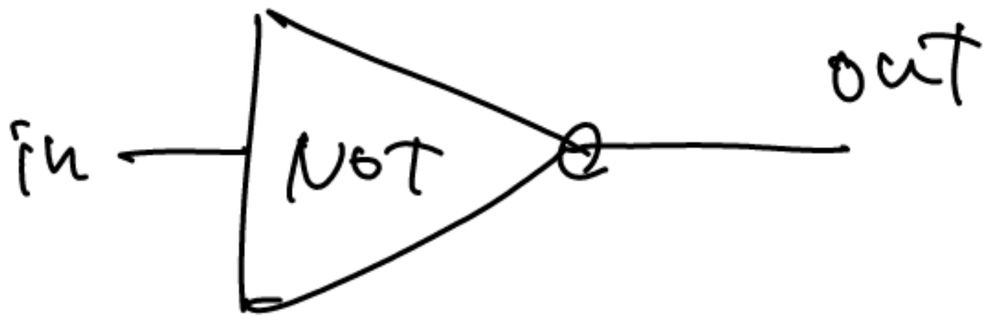
- Elementary (Nand, And, Or) ← Primitive なゲート
- Composite (Max, Adder ...)
  ↑ elementary を組み合わせたもの

gate diagram

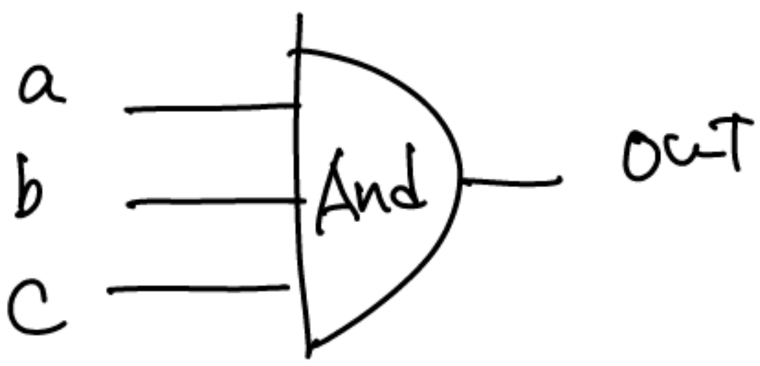


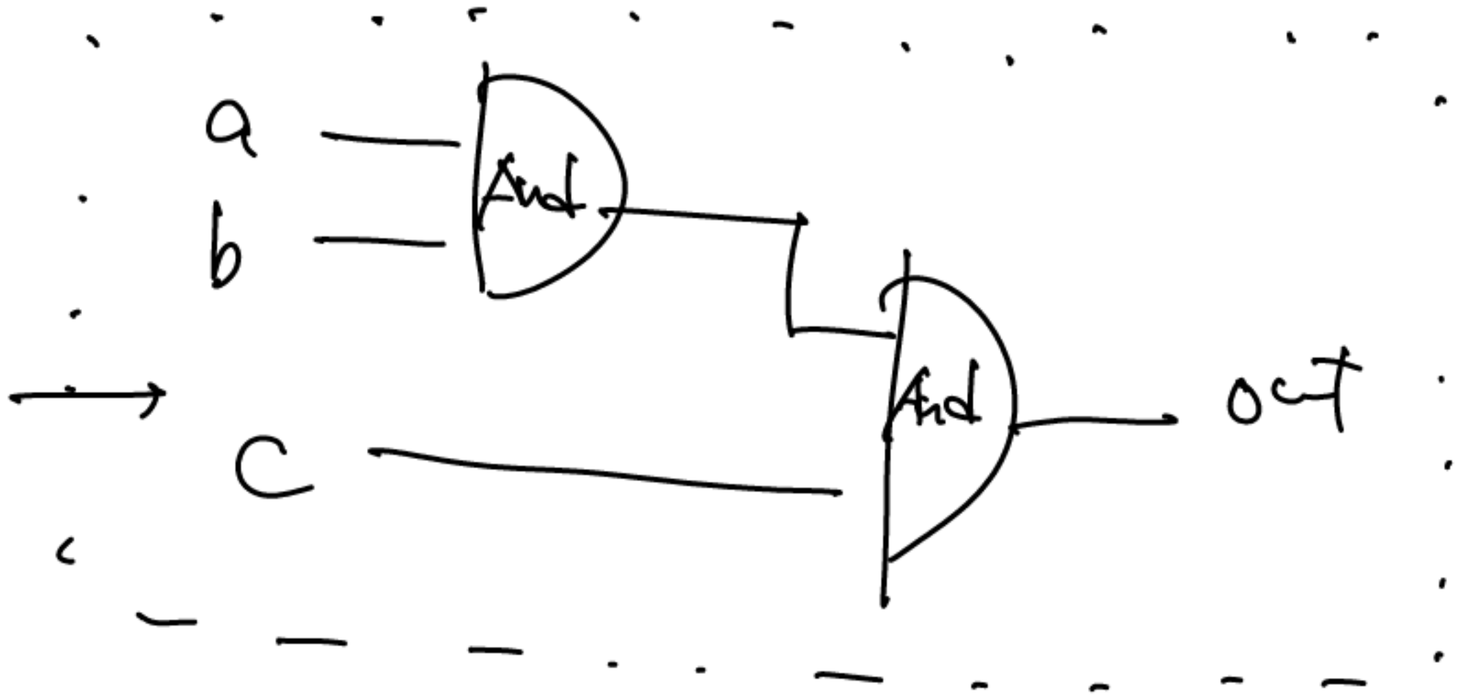






Composite gates



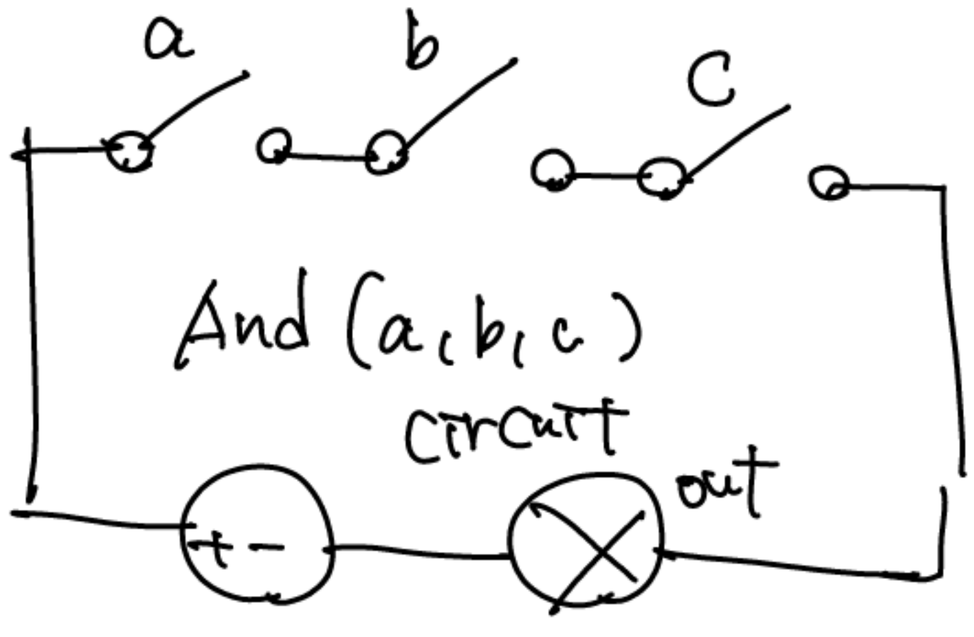


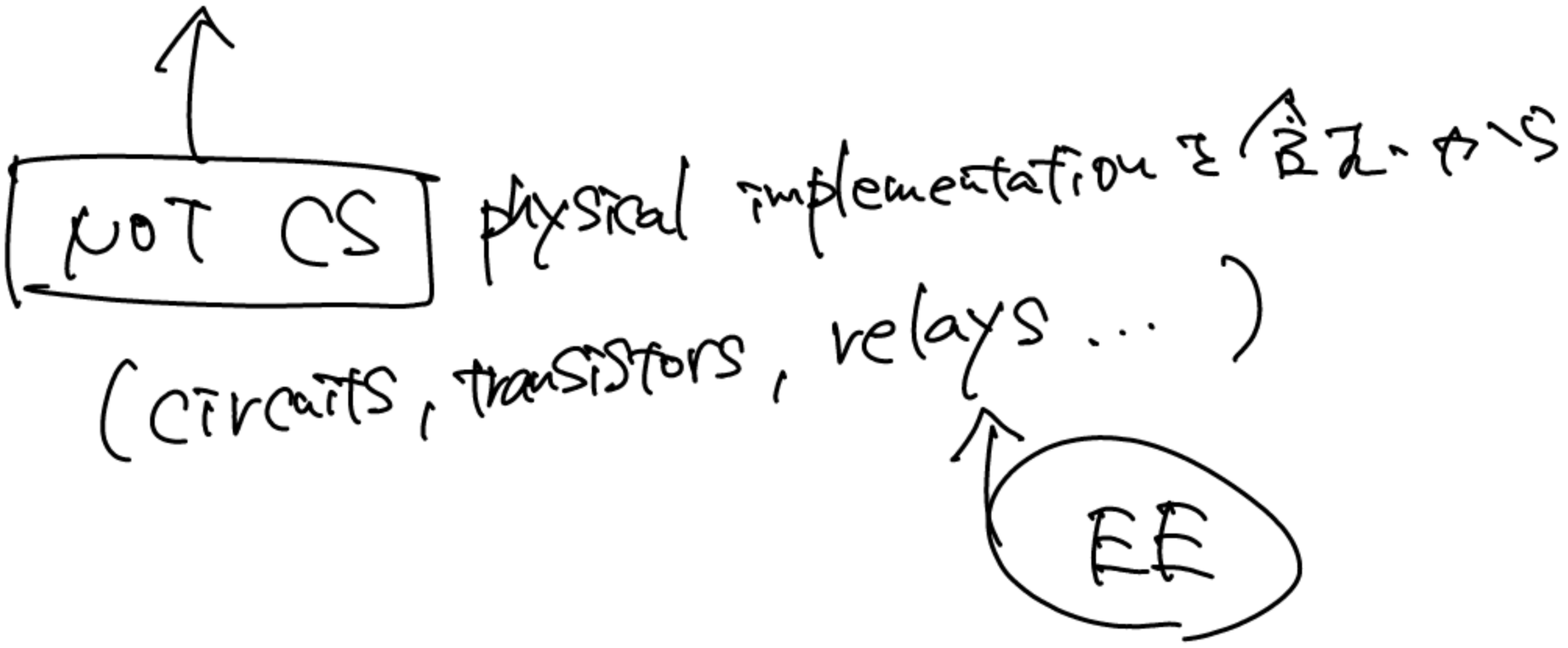
ANDを2つ組み合わせることで
引数を3つ取れるようにしている.

Interface

↑ implementation

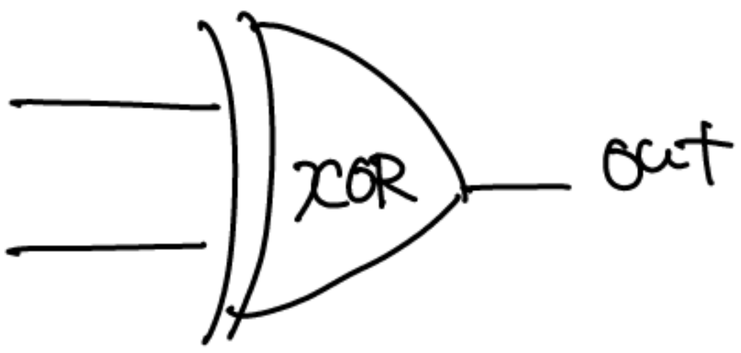
Circuit Implementation





Hardware Description Language

最初に詳細を書いた Requirement が一番重要



a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

diagram, Truth Table

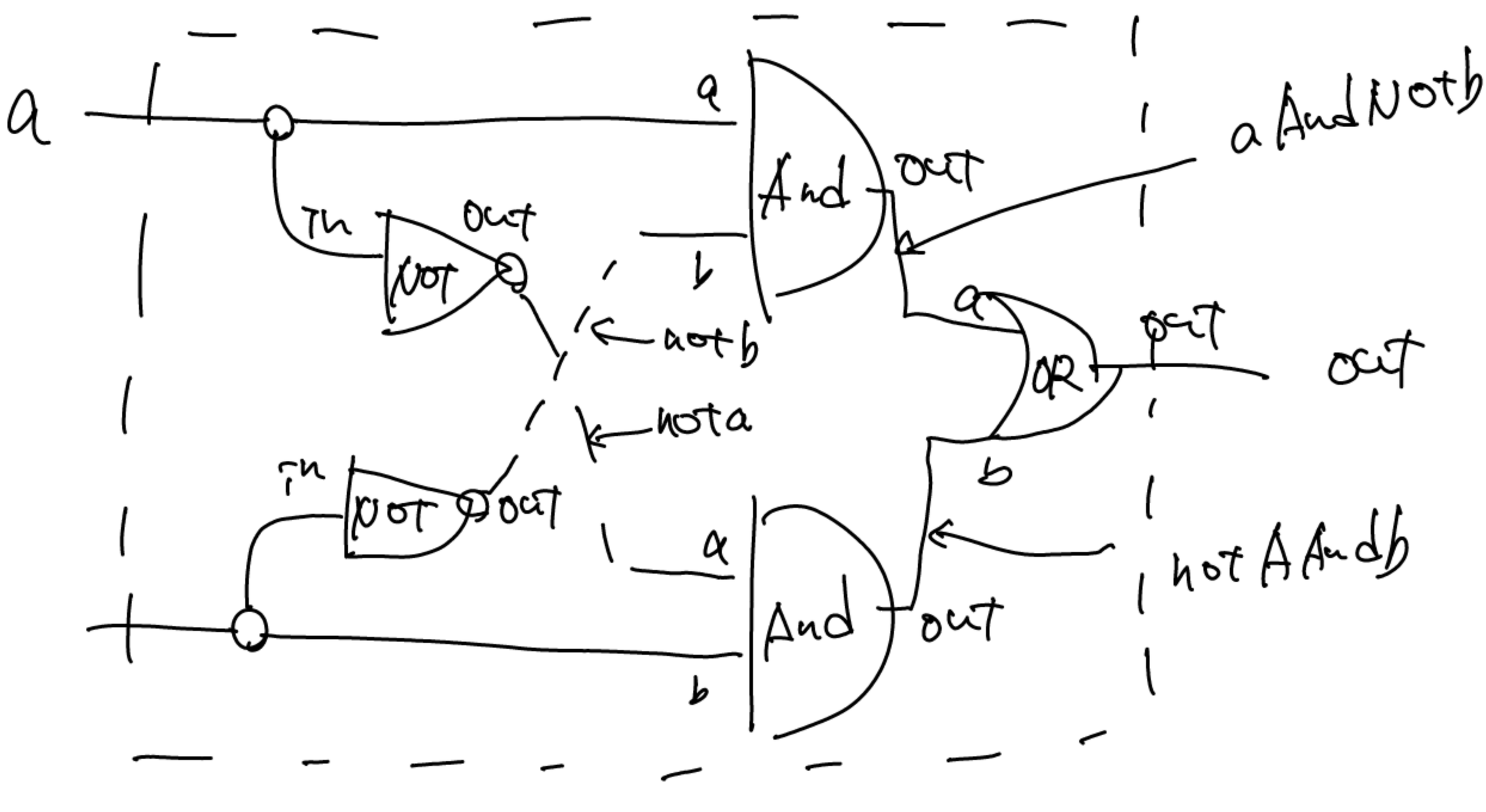
↓

General Idea

out = 1 when a and Not(b)
Or
b and Not(a)

↓

gate diagram to HDL



HDL

CHIP Xor {

IN a, b

OUT out

} interface

PARTS:

Not (in=a, out=nota);

Not (in=b, out=notb);

⋮

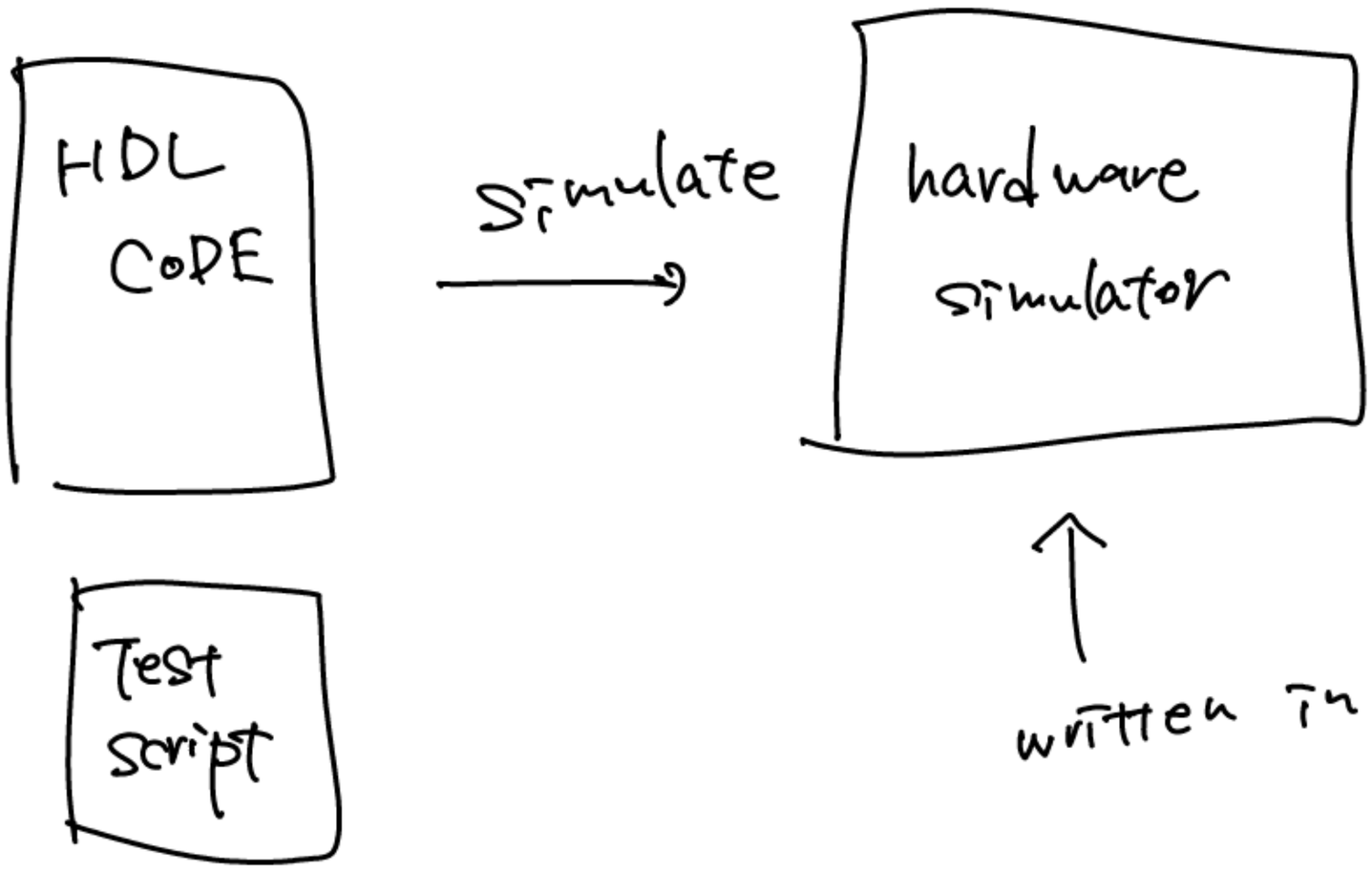
} implementation

Diagram & Text 는 다른 표현 방식

- HDL is functional / declarative language

↳ VHDL
Verilog
) 90% in industry use

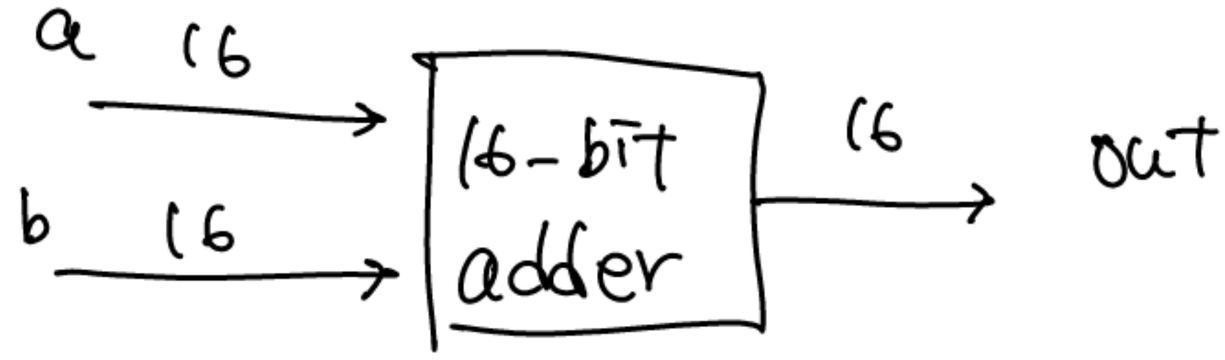
Hardware Simulation



1. Load HDL code on simulator
2. Pass inputs such as a, b
3. Check output

.TST file : unit test 하기 위한

Multi-Bit Buses



16 bitsを扱うためには array を使う.

buses

In first[16], second[16]

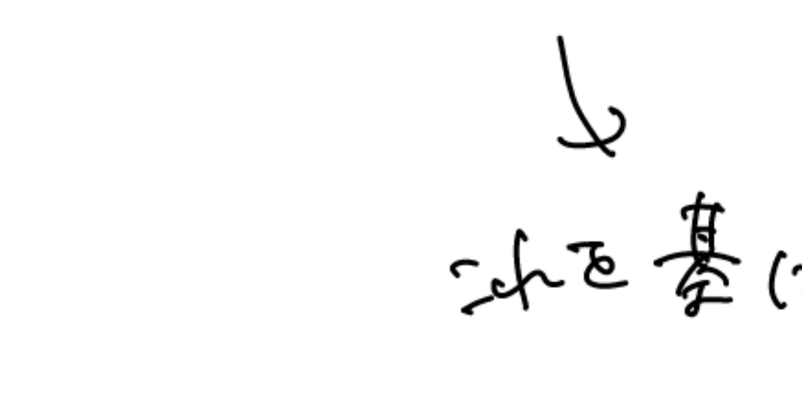
Project 1

build 15 gates using NAND

↓

これを基にコンピューターを作る

Multiplexor



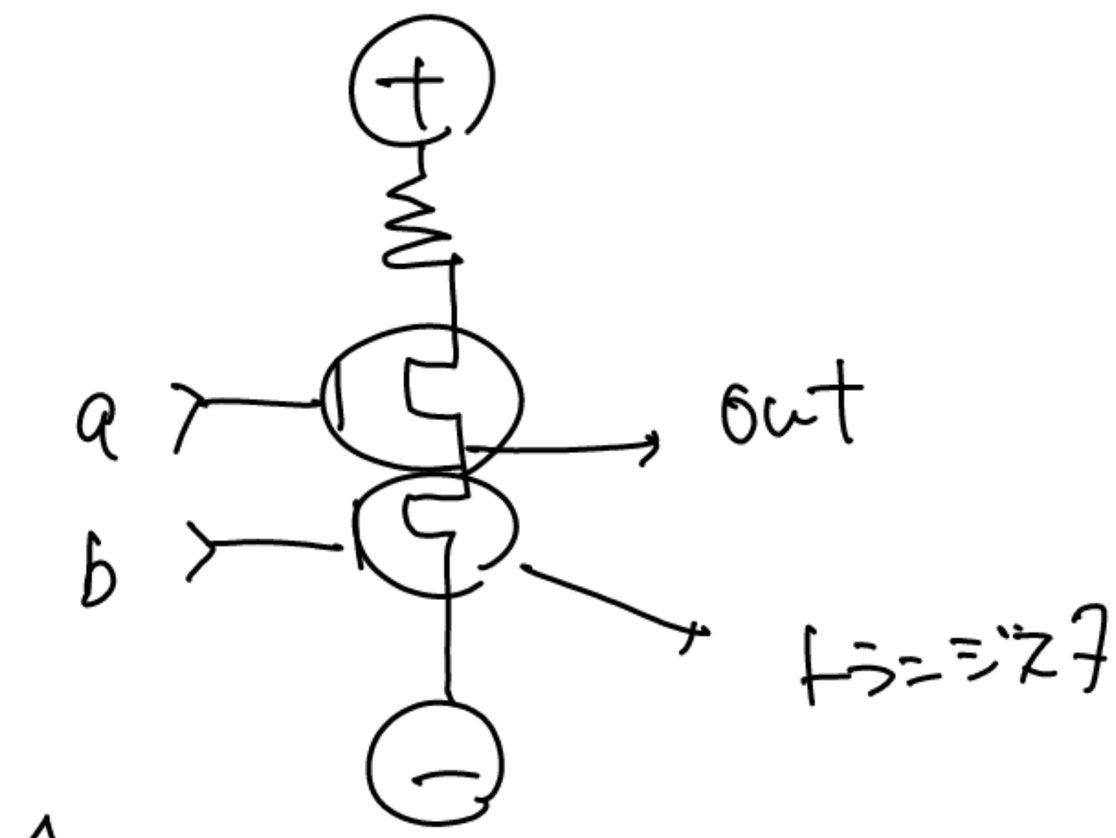

```
if (sel == 0)
    out = a
else
    out = b
```

Demultiplexor

- inverse of Mux

```
if (sel == 0)
    {a, b} = {in, 0}
else
    {a, b} = {0, in}
```

NMOS impl. NAND) Detail info



↑ もし a, b ともに high Voltage で、

トランジスタがつながる

結果 ⊕ と ⊖ を合わせて 0 が out となる

それ以外は ⊕ だけなので 1

Module 2 : Boolean Arithmetic and the ALU

Binary Numbers

'0' '1' で number を表現

Binary → Decimal

$b_n b_{n-1} b_{n-2} \cdots b_1 b_0$

$= \sum_i b_i \cdot 2^i$

Decimal → Binary

2の累乗で最大となる値を見ていく

87 decimal = 64 + 16 + 4 + 2 + 1

= 0 1 0 1 0 1 1 1 (binary)

32 8

Binary Addition

addition のみ実装できれば Subtraction は簡単

Multiplication, Division はハードウェアではなく

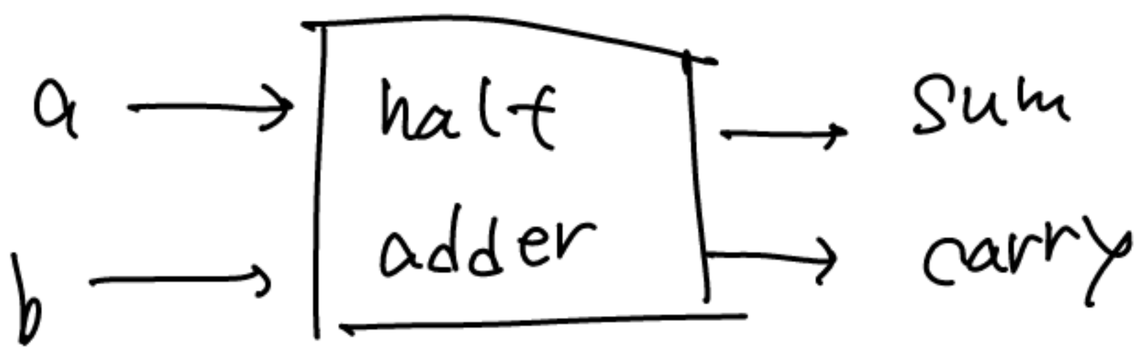
ソフトウェアで実装する (このコース-子のほうが簡単)

```
  1 1 1 1 1
  0 1 1 1 1 0 1
  0 0 1 1 0 1 1 0
-------------------
  1 0 1 1 0 0 1 1
```

どうやって 1+1 の繰り上げを行う?

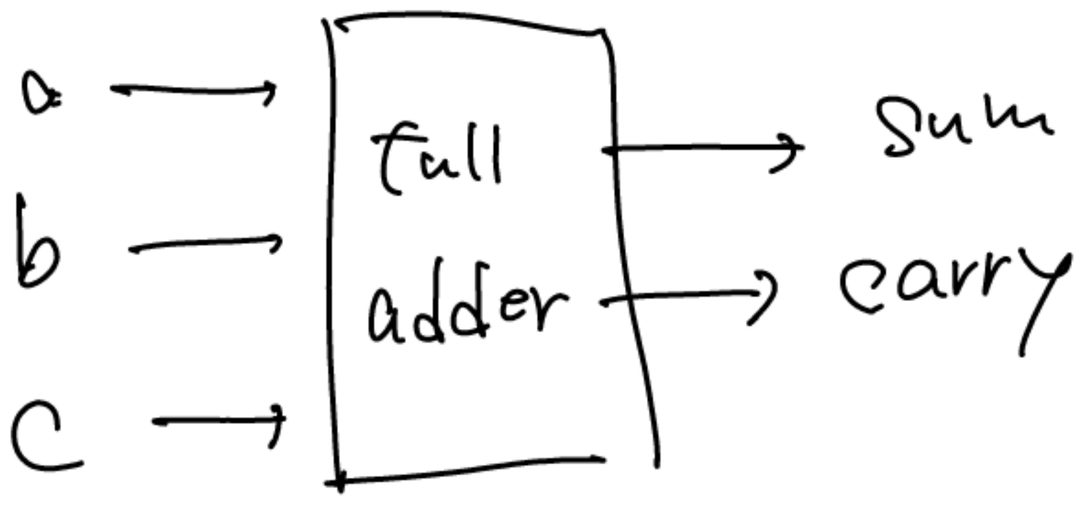
Halt Adder Chip とは

a	b	Sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Full Adder

Half Adder だけでは繰り上がらんで数に
対応できないので この Chip が必要 (引数と3つとる



Negative Numbers

これまで Positive Number だけを扱ってきたが、
Negative Number をどう扱うか。

Sign bit

最初の bit を −/+ として扱う。
それ以外は全て Positive Number とする。

↑ Not popular !

* − 0 が存在する
* Implementations が2つになる。

2's complement

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8 (8)
1001	-7 (9)
1001	-6 (10)
1011	-5 (11)
1100	-4 (12)
1101	-3 (13)
1110	-2 (14)
1111	-1 (15)

$0 \ldots 2^{n-1} - 1$

$-1 \ldots -2^{n-1}$

Addition in 2's complement

$$\begin{array}{r} -2 \\ +\ -3 \\ \hline -5 \end{array} \qquad \begin{array}{r} 14 \\ +\ 13 \\ \hline 11 \end{array} \qquad \begin{array}{r} 1110 \\ +\ 1101 \\ \hline 11011 \end{array}$$

$11011 = 27$

$1011 = 11$

11 in 2's complement of -5

$$\begin{array}{r} 7 \\ +\ -5 \\ \hline 2 \end{array} \qquad \begin{array}{r} 7 \\ +\ 11 \\ \hline \end{array} \qquad \begin{array}{r} 0111 \\ +\ 1011 \\ \hline 10010 \end{array}$$

0010

Computing $-x$

input : x

output : $-x$

というやつてやばい

$$2^n - x = 1 + (2^n - 1) - x$$

(例)

input: 4

1111

$-$

0100

1011 ← flip bits

$+\quad 1$

1100

↑

$-4\ (12)$

(5-1)

in 4-bit binary

-7 is $2^4 - 7 = 9$ (1001)

-5 is $2^4 - 5 = 11$ (1011)

x から $-x$ を binary で求めよう

1. flip all the bits of x

2. add 1

(5-1)

in 4-bit binary

3 (0011)

↓ flip

1100

↓ +1

1101 ⟶ $2^4 - 3$

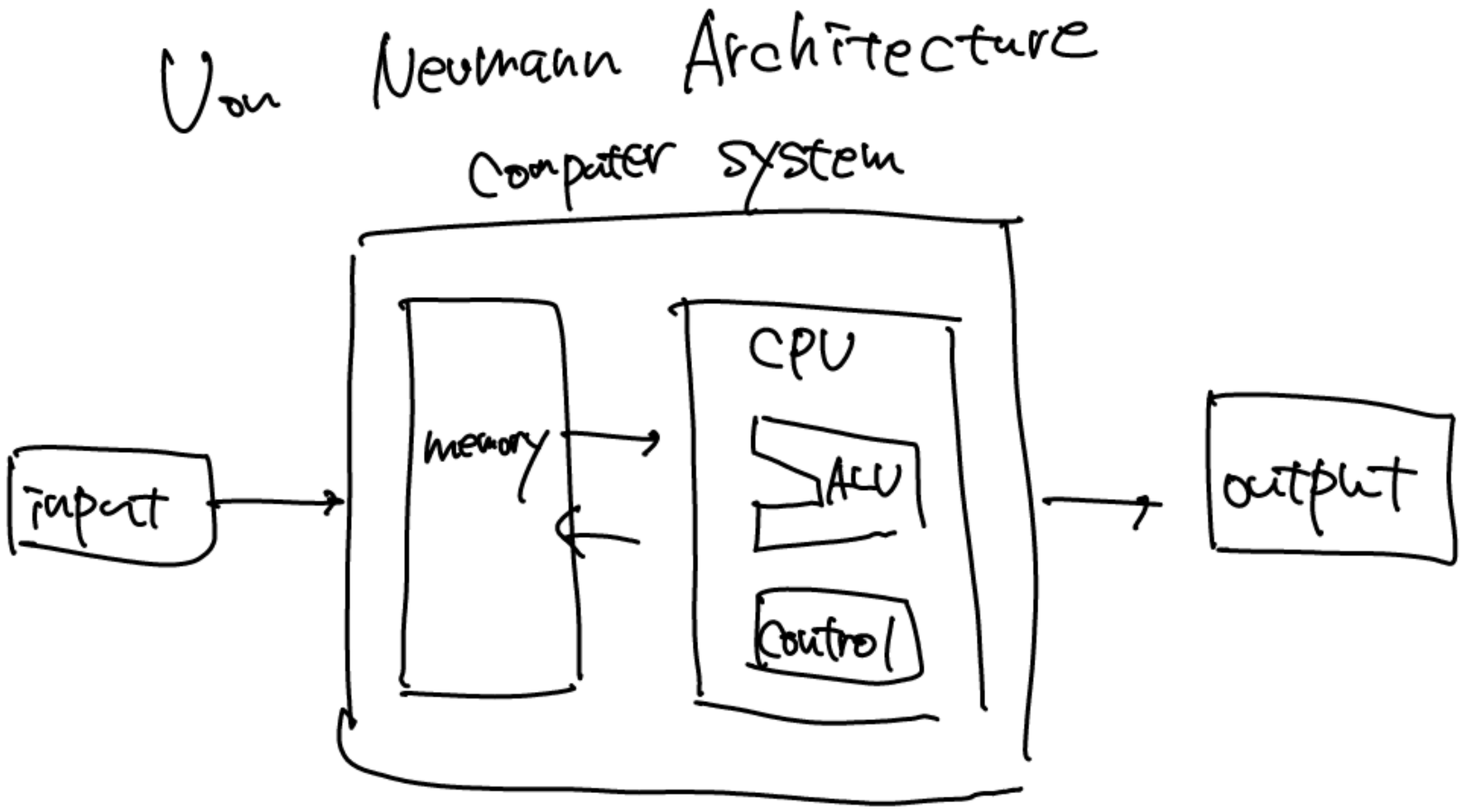
(13)

input = 6

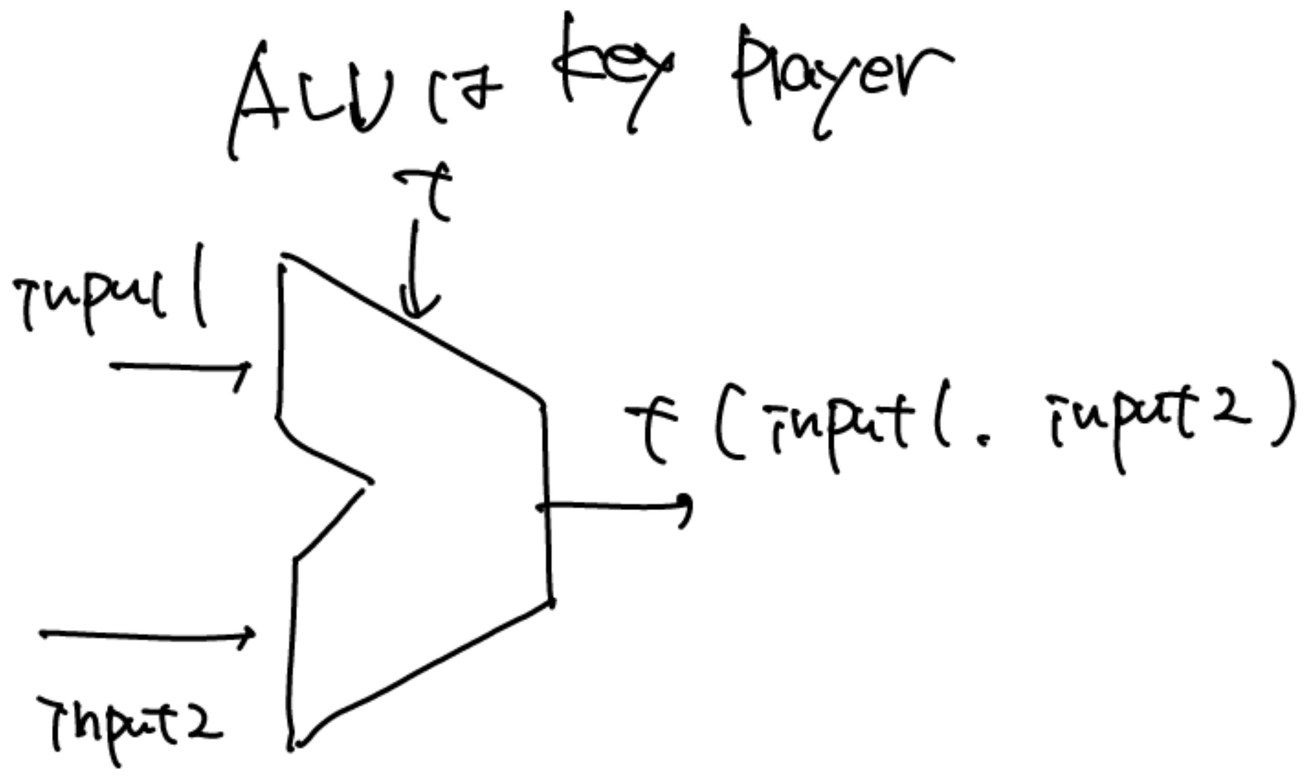
$$
\begin{array}{r}
1\,1\,1\,1 \\
-\;0\,1\,1\,0 \\
\hline
1\,0\,0\,1 \\
+\qquad\;\;1 \\
\hline
1\,0\,1\,0
\end{array}
$$

Arithmetic Logic Unit (ALU)

Von Neumann Architecture



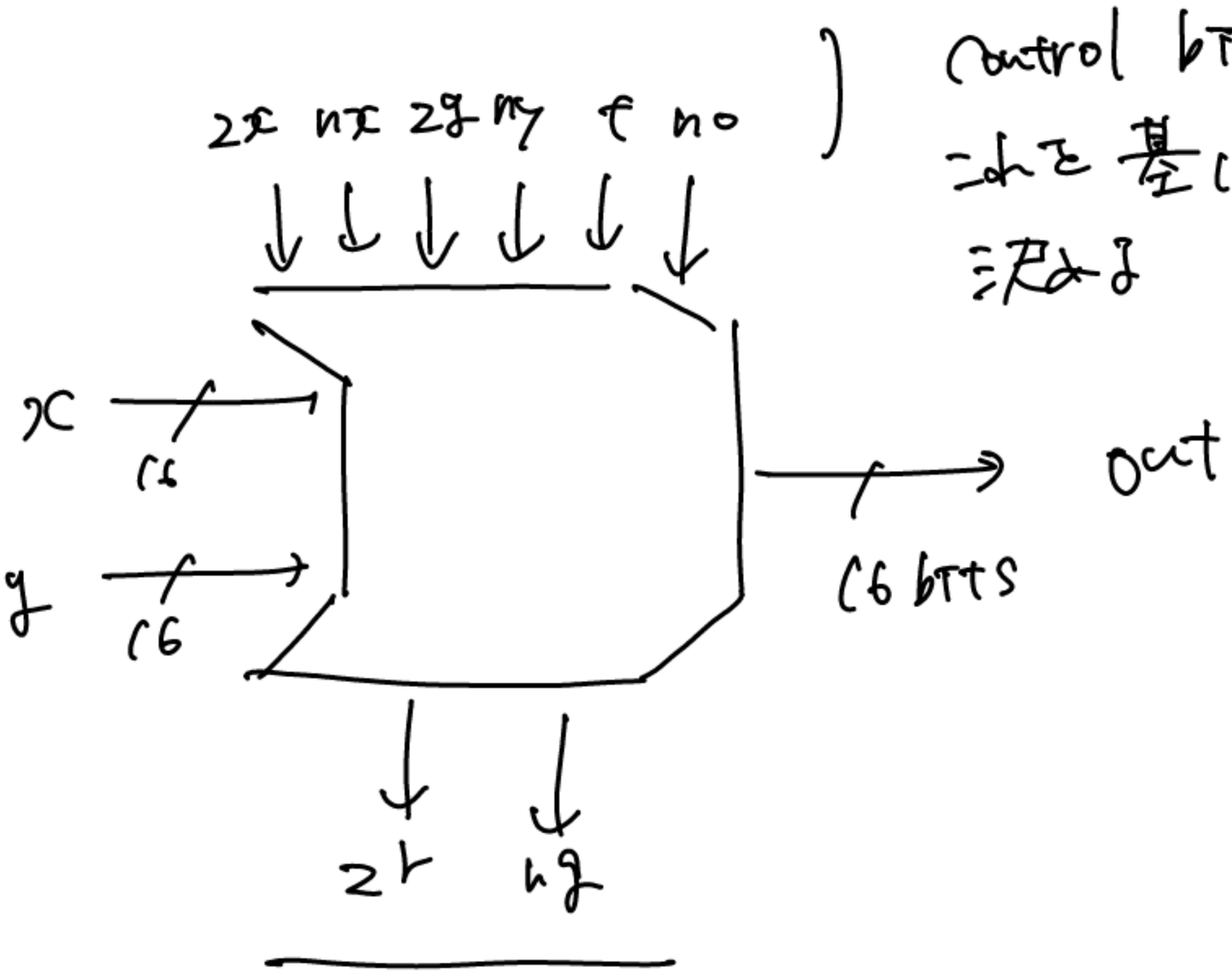
ALU は key player



ALUは f, input1, input2 を引数にとる

f は predefine された関数

- Arithmetic operations : integer add ...
- logical operations : And, Or ...



control bits
これを基にどのオペレーションを行うか
決める

If out = 0 then zr = 1, else zr = 0

If out < 0 then ng = 1, else ng = 0

zr, ng は Coputer Architecture とほぼ同じように利用する

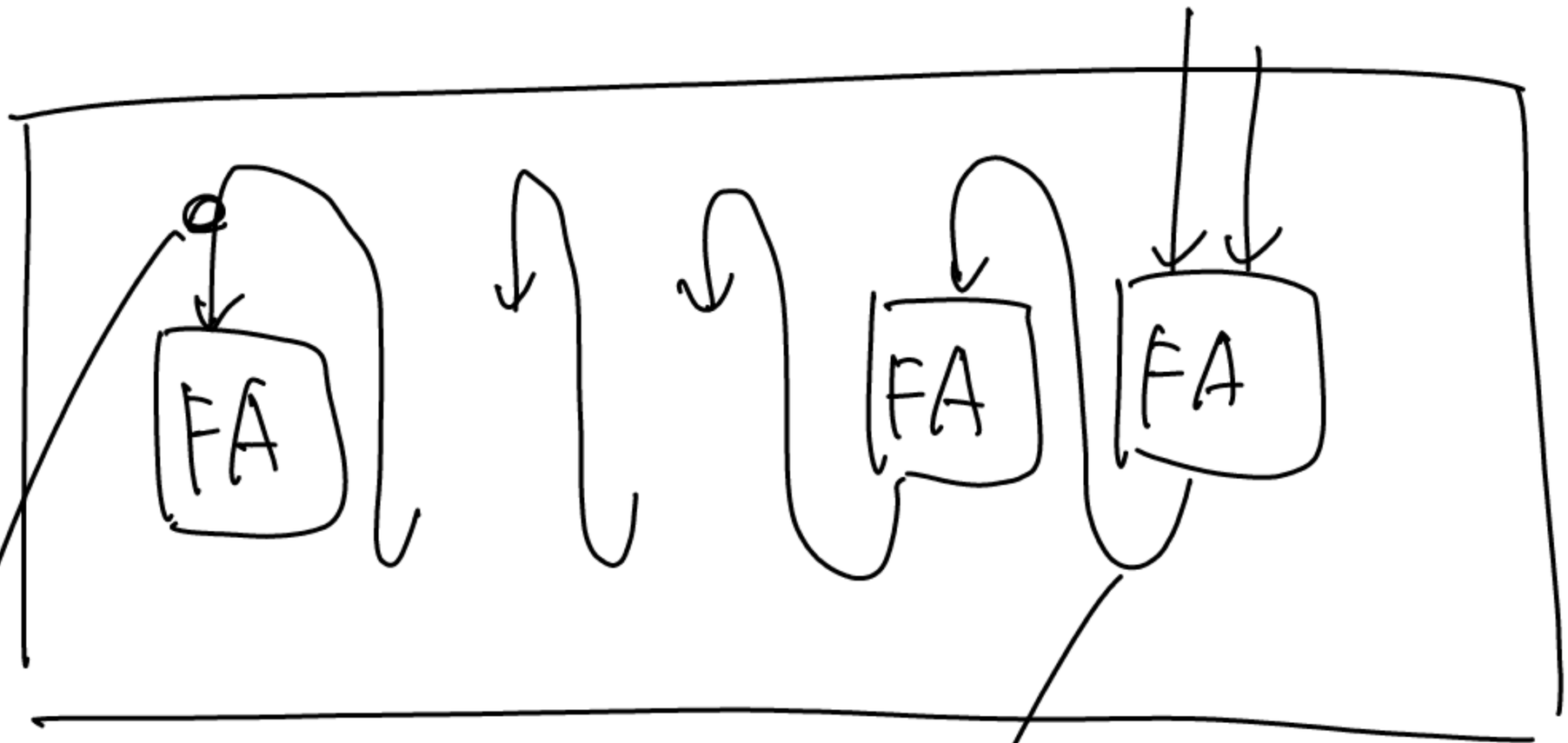
Perspective

- Are chips standard in industry.

  Yes but not for ALU
  これはニーズによる

- ADDER

  - 改善できる



carryが全てのfull Adderで行なわれる

かわりにCarryの一部だけ
を見る方法
Carry look ahead
を利用して改善できる

大きいチップには向てる
影響するので良くない

Half Adder

a	b	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Memory module 3

コンピューターで
どうやって時間を考えるか

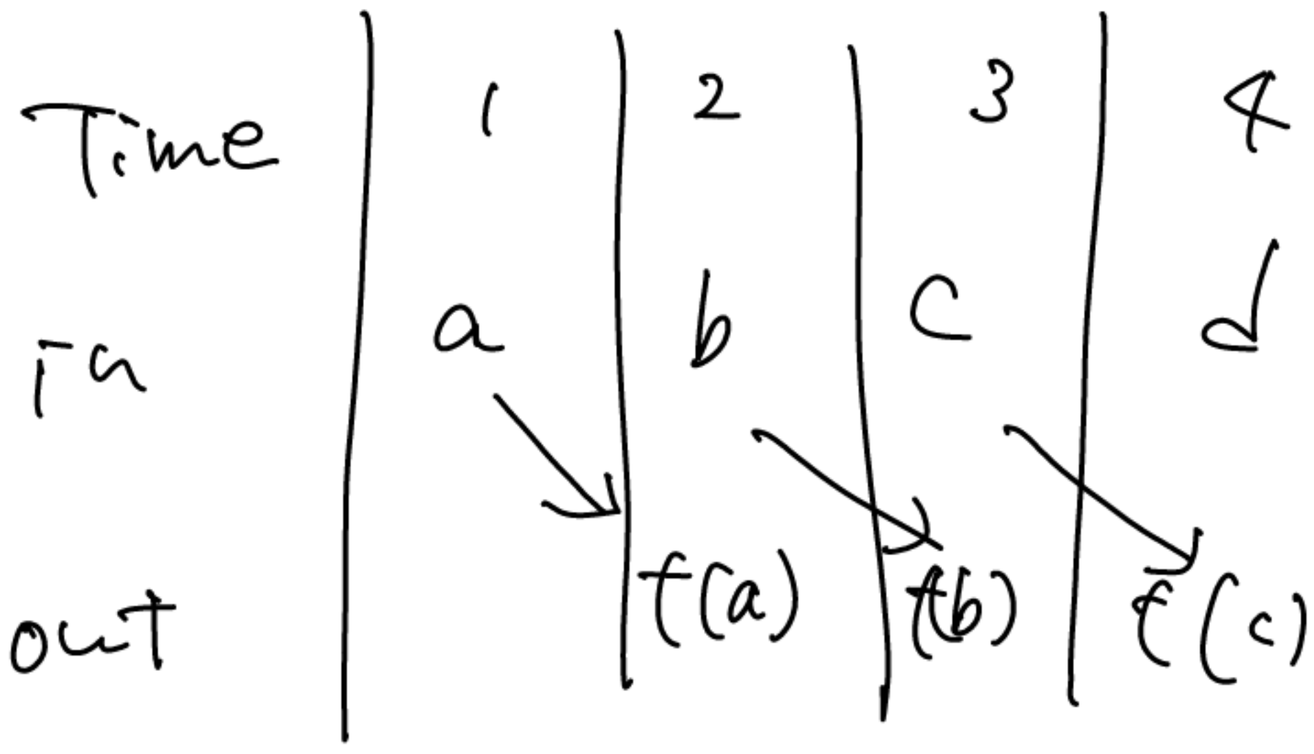
↓

Continuous には
依存せずに
1つずつステップごと
依存する

Sequential Logic

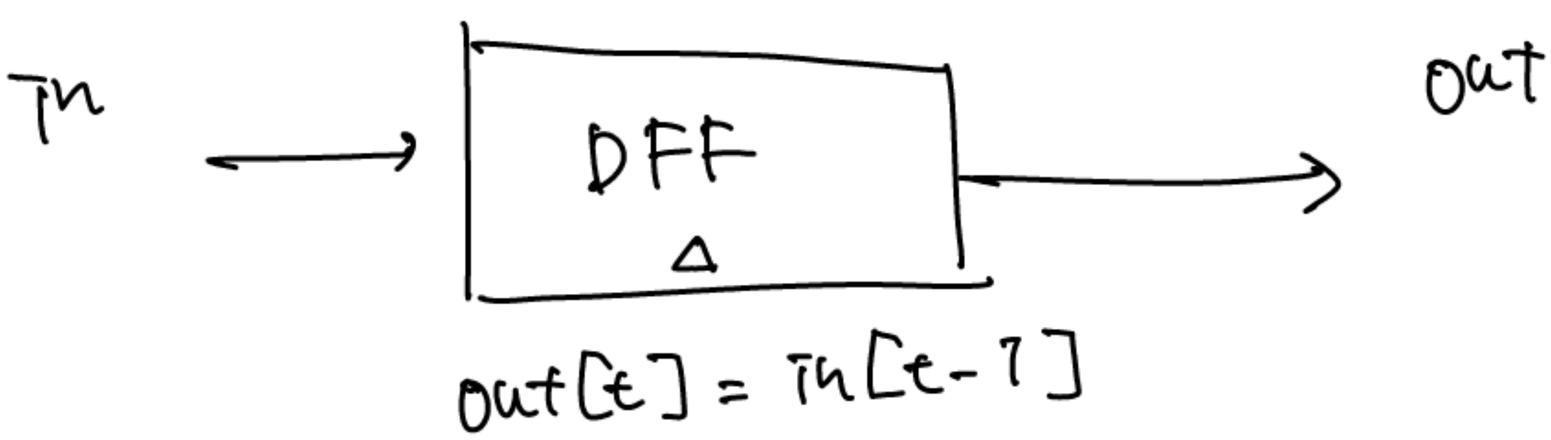
- Use the same hardware
- State

Out[t] = function (t-1)

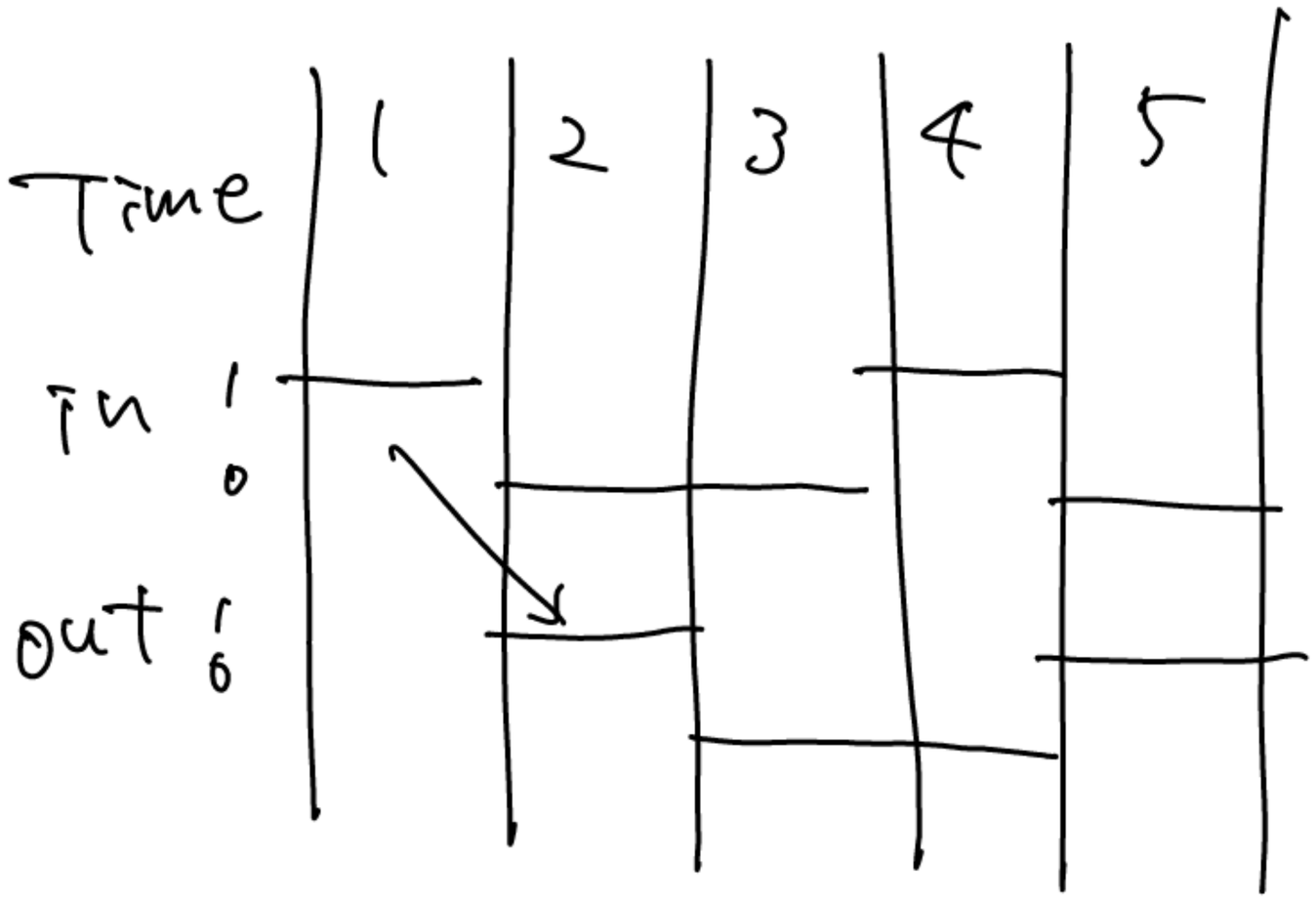


State [t] = function (state [t-1])

"Clocked Data flip Flop"

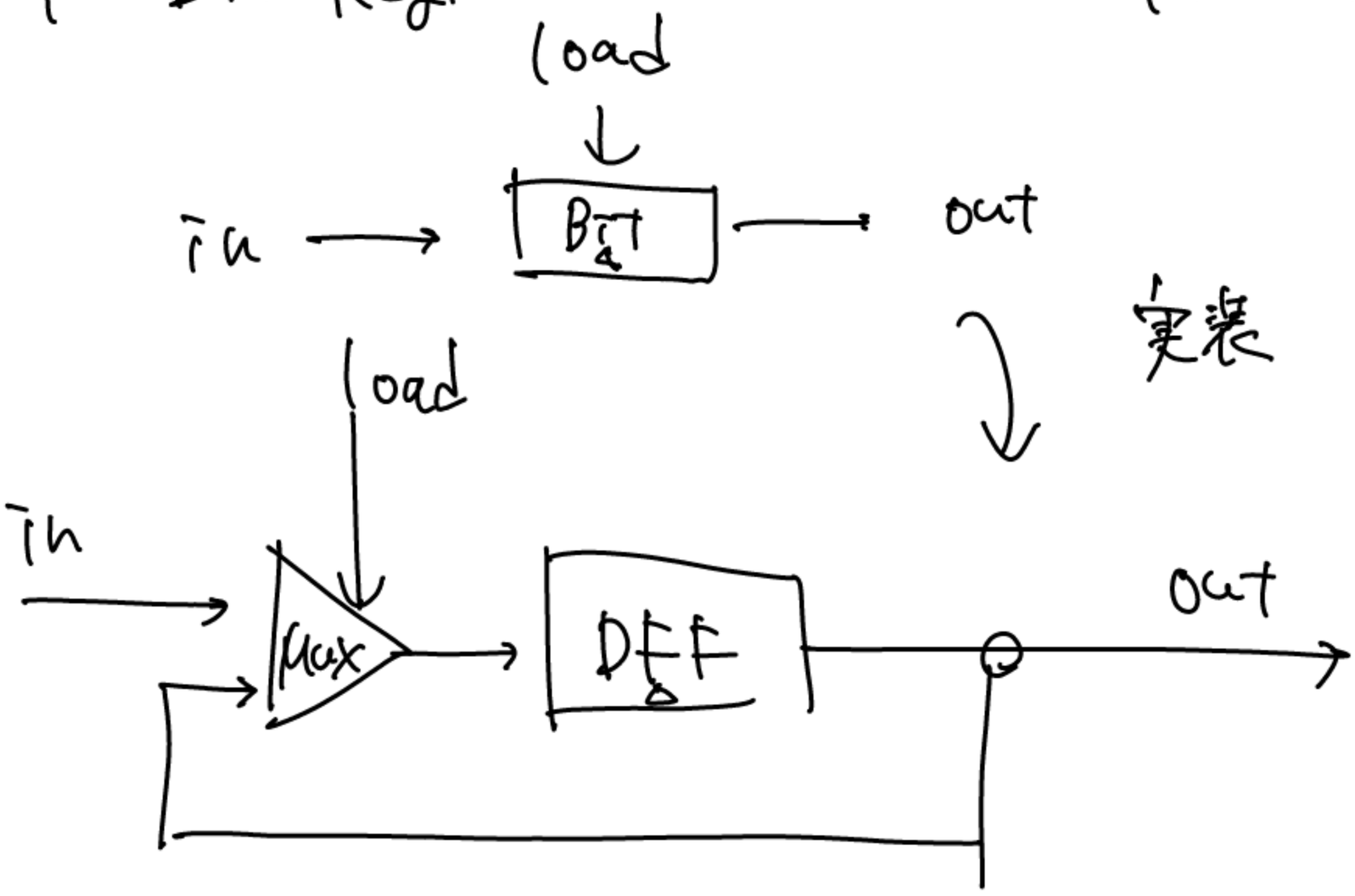


out[t] = in[t-1]



Prevn State & Output of

1- Bit Register ⟶ 1 bit メモリ



Memory

main memory : RAM

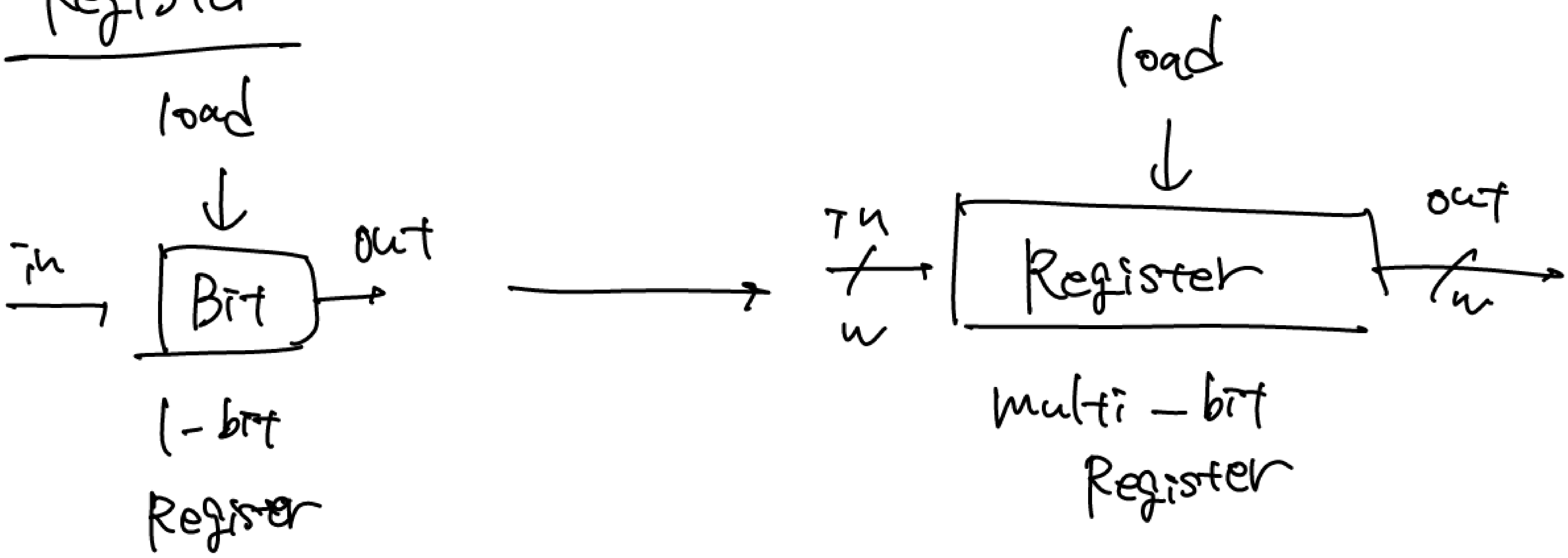
secondary : disks ...

Volatile / non-volatile ?

RAM stores

- Data
- Instructions

Register

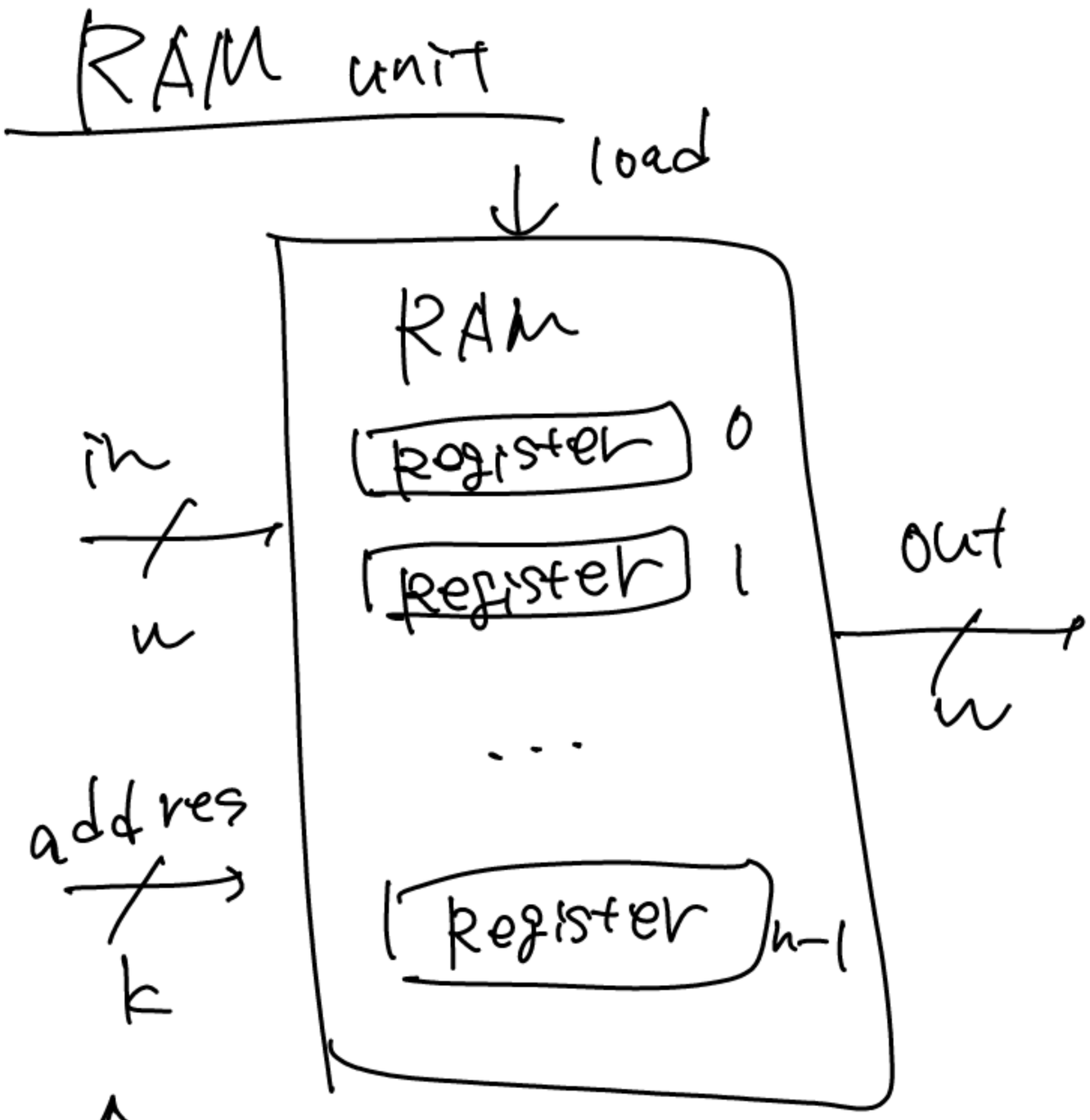


Registerを読み込むには

→ outを確認
(Register's state)

書き込むには

set in = v ← stateをvにする
set load = 1



どうやって RAMのデータを読むか？

↳ 1. address で registerを指定 (i)

2. register (i) の Stateを outする

どうやって RAMにデータを書き込むか？

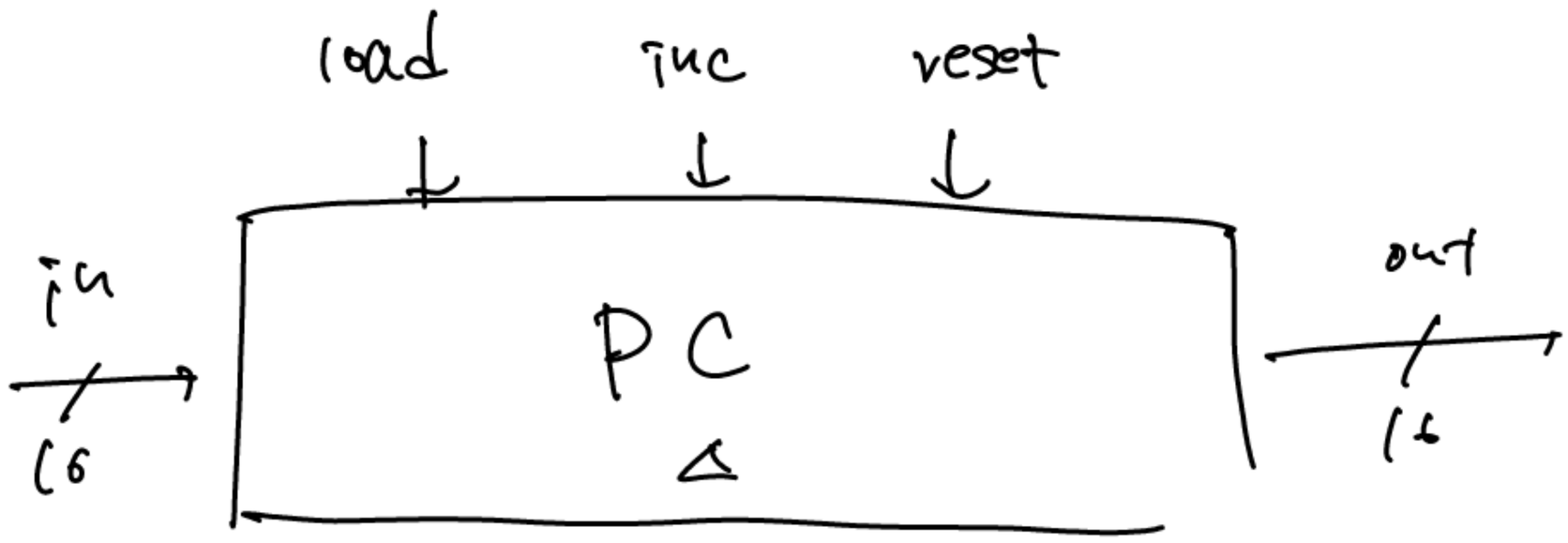
1. address で registerを指定 (i)

2. load = 1

3. in = v にて入力値 Registerの Stateを更新.

Counter

- increment number PC++
- set value PC = v
- reset PC = 0



if reset == 1

out[t+1] = 0

else if load == 1

out[t+1] = in[t]

else if inc[t] == 1

out[t+1] = out[t] + 1

else

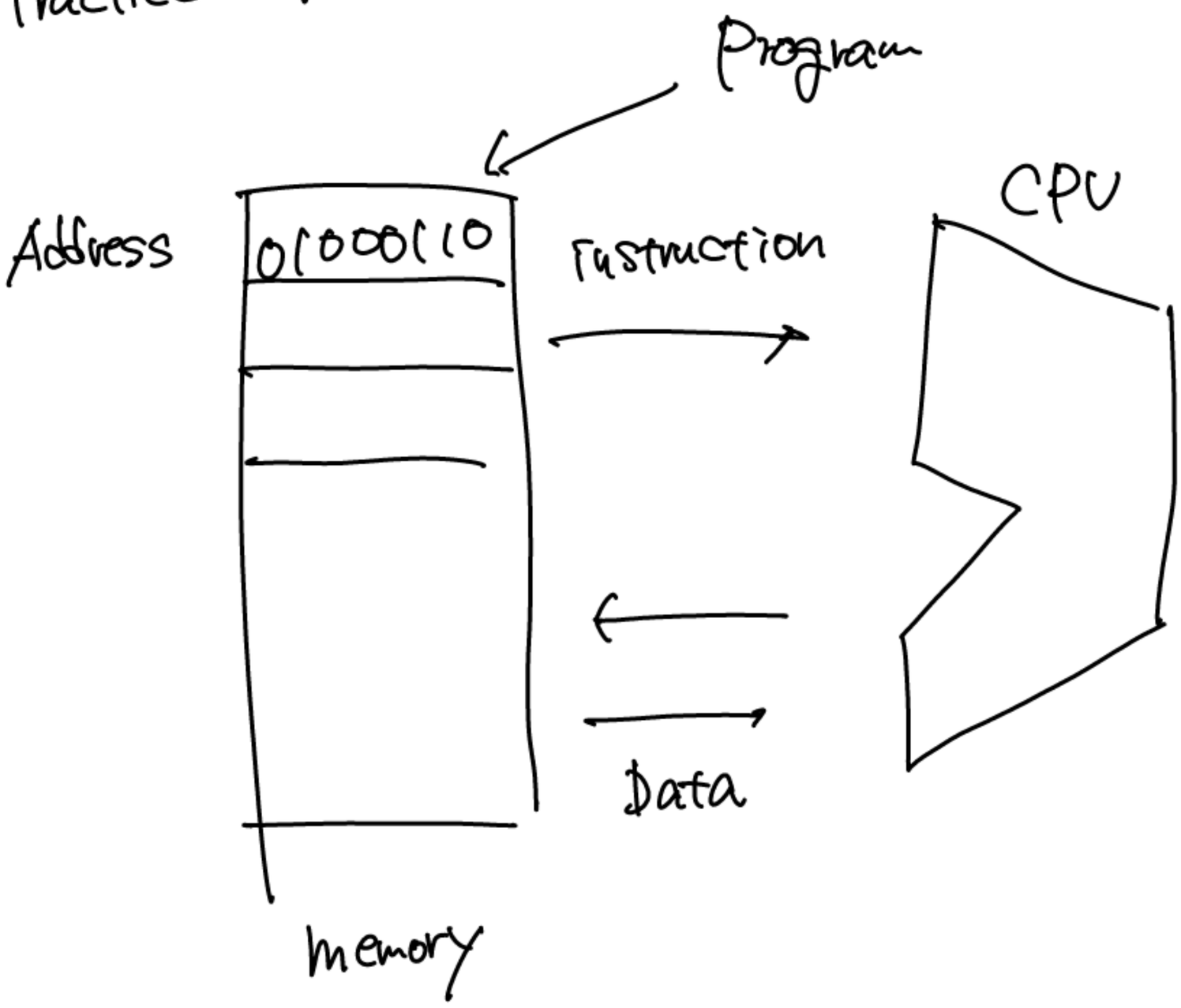
out[t+1] = out(t)

Machine Languages

ミニブル machine が 様々な存在 とできるのは Remarkable

Theory : Turing

Practice : Neuman



Instruction

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

ADD R3 R2

(registar)

assembly language

translate

machine language

ミニブル 言語

人間 が 理解 できる

Elements

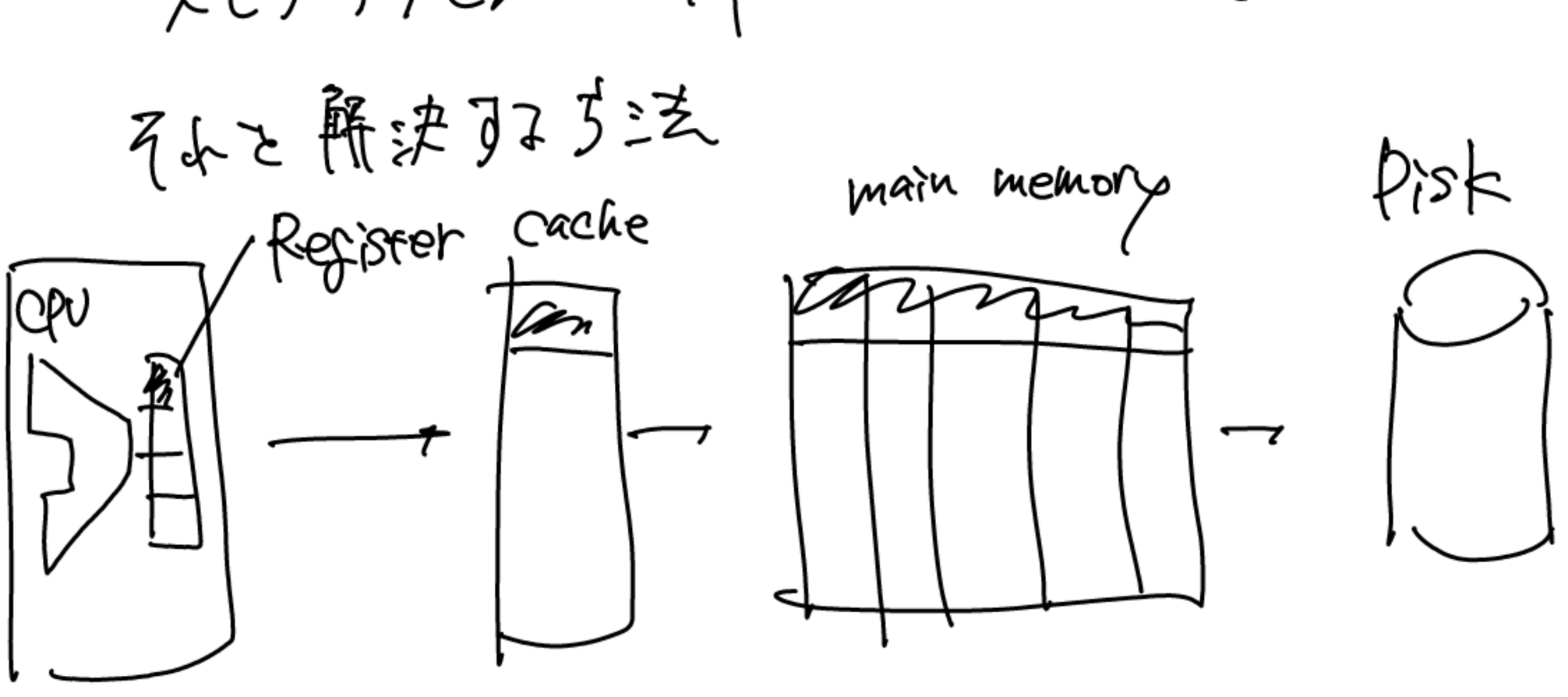
Machine operations

- ハードウェアに実装されている
  - Arithmetic operations : add, subtract
  - Logical : And, Or
  - Flow Control : goto
- Differences between machine languages
  - Richness of the set of operations.
  - Data types (divison?)
    (float, int?)

Memory Hierarchy

メモリアクセスが expensive なので（時間がかかる）

それを解決する方法



Registers について 取り組む

CPUは一般的に Registers を保持している。

Add R1, R2

Register 1 の値を Register 2 に 足してR

Store R1, @A

Register 1 の値を うばして A に 引き渡す。

Flow Control

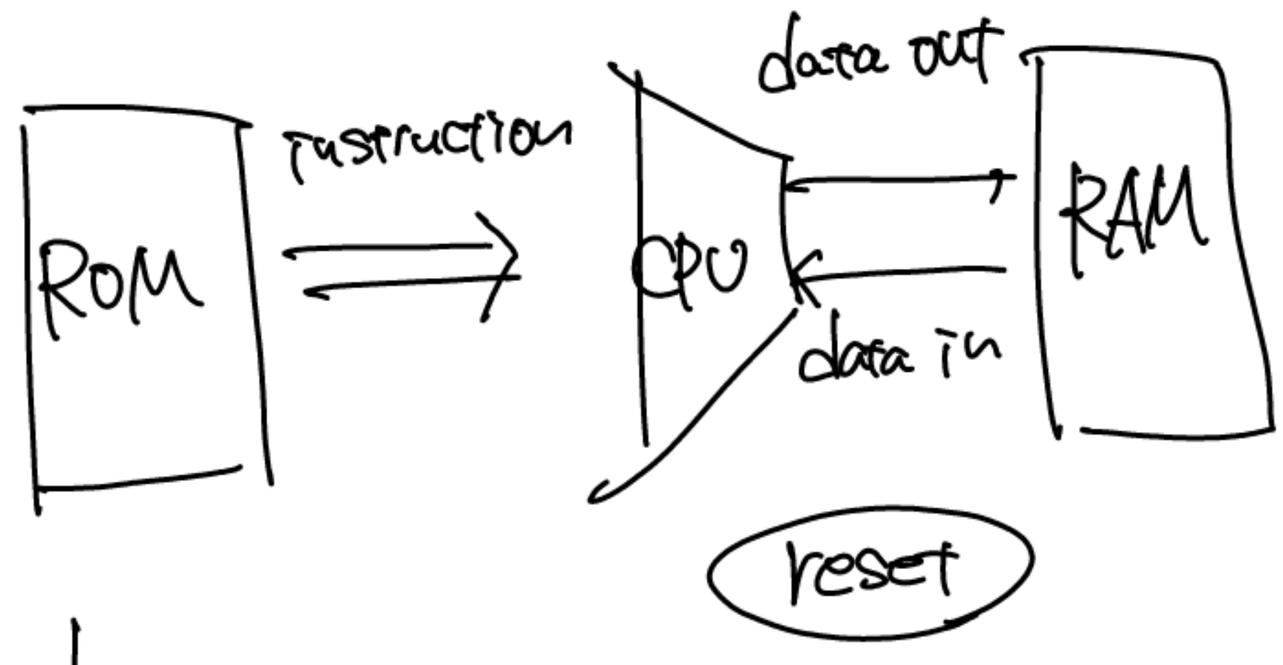
- unconditional Jump
- Conditional Jump

loop とかで 必要

16-bit machine のオルトの要素から構成される

- Data memory (RAM)
- Instruction memory (ROM)
- CPU (Performs 16-bit instructions)
- Instruction bus / data bus / address buses

Hack Computer : Software



Control

1. ROM is loaded
2. Push reset button
3. Start running programs

(a) 1

M = A - 1 ; JEQ

1. Set A register to 1.
2. Set M register to 0. (A-1)
3. JEQ checks computation is 0 or not
4. It's true, so the next instruction is value stored in A register, which is 1. *1

Hack machine language

- A-instructions
- C-instructions

Hack program = sequences of instructions

machine language 은 2가지 종류

- Binary Code
- Symbolic language (need to be translated to binary code)
  - ⓐ value

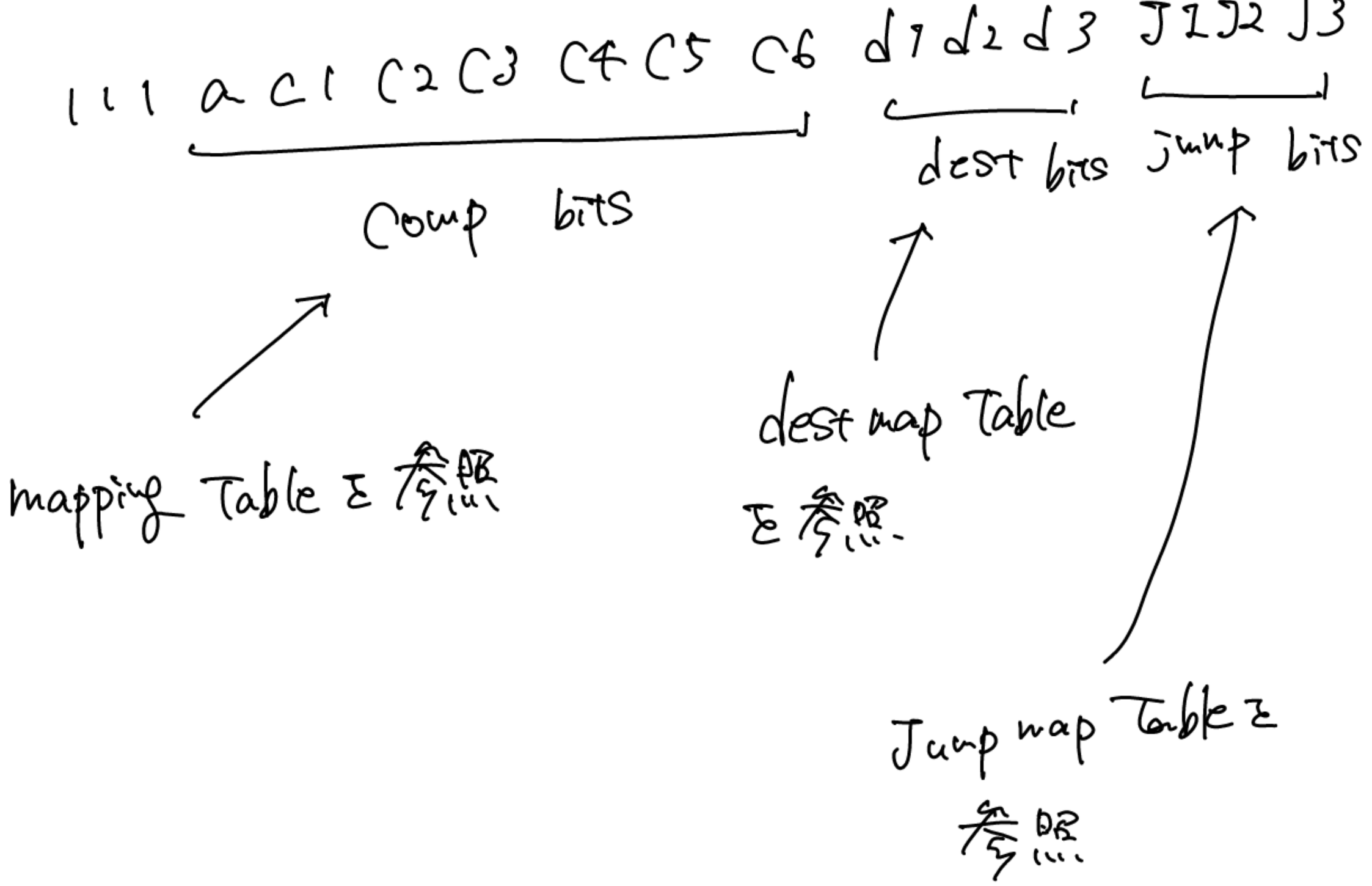
A-instruction (16진수)

Symbolic	ⓐ value	ⓐ 21
Binary	0 value 15 bit	⓪ 0000000000010101 21

C_instruction

dest = comp ; jump

Binary Syntax



Input / output

Peripheral I/O devices

- キーボード
- マウス
- スクリーン

} get data from users / display data to users

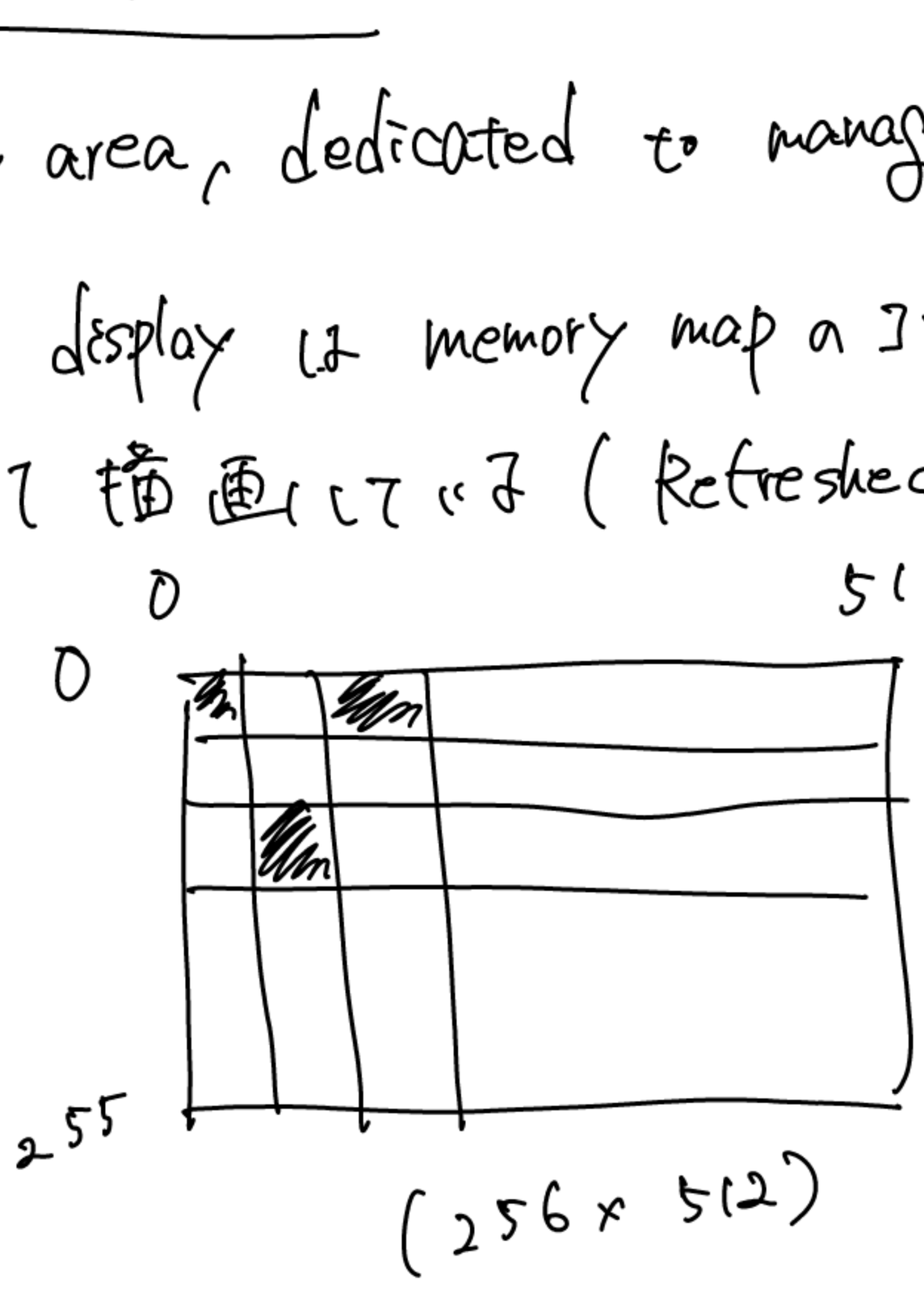
BitsでどうやってこれらのデバイスをTalkするか

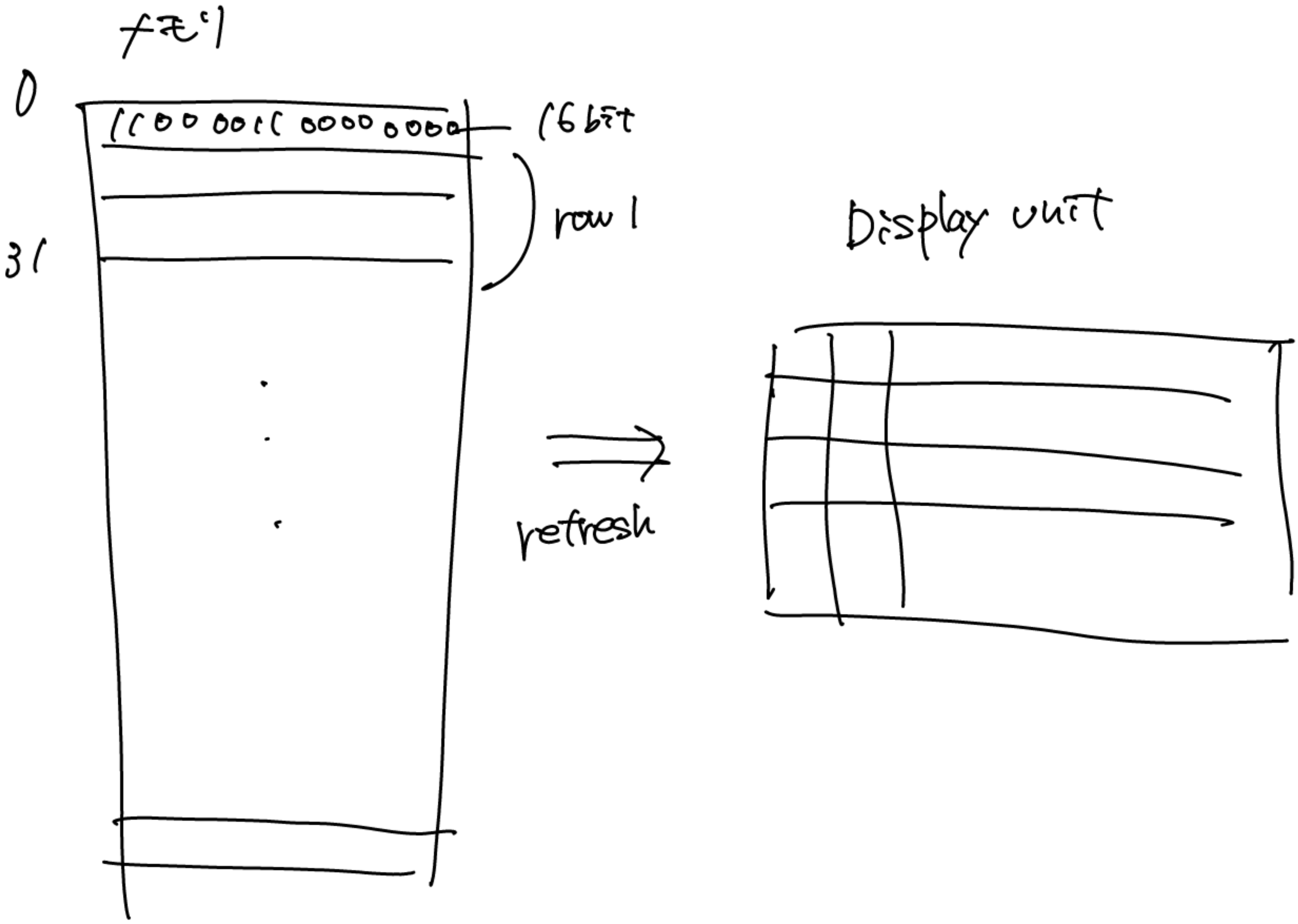
(Part 2では高級言語での扱いを紹介する)

Screen memory map

memory area, dedicated to manage a display unit

physical display は memory map のコピーを
連続して描画(refreshed)





- メモリを 2D の Display unit に マップする必要がある
- Display unit の (x, y) = 1 なら 対象の x, y を点ける

1. word = Screen[32*row + col/16]
   - 16bit
   - これで対象の Pixel のアドレスが含まれる
   - 行アドレス
2. Set (col%16)th bit of word to 0or1
3. Commit word to RAM

keyboard memory map

キーボードと接続すると screen memory map と同様に
keyboard memory map を介してコンピュータと
やり取りする (16bit)

keyboard から入力された scan code が
memory map に表示される

In hack, RAM[24576] が memory map

Working with registers and memory

```
// D = 10
@10        ← 一度 A register にアサインする
D = A        必要がある

// D++
D = D + 1

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

// RAM[17] = 10
@10
D = A
@17
M = D
```

Branching

goto

ⓐ Declaration を使う.

Variables

sigle register

@ {variable name}

↑

find availabe memory register and use it as "name" variable

アドレスを指定しないで

アドレスサイズなどに依存しないようにする。

Hack computer has three 16-bit registers

D register : data register, which stores a 16-bit value.

A register : serves as address register and data register

Set 17 to RAM[100]

@17 (A works as data register)

D=A

@100 (A works as address register).

M=D

```
Mult    R2 : R0 * R1

Sum = 0

n = R1

i = 0

Loop:

        if  i >= n:

            stop

        sum += R0

        i++

Stop:

        R2 = Sum

        i = 0

            4 < 2

2 - 0

2 - 1

2 - 2
```

Fill

- Listen keyboard
- blacken / whiten changing values of display memory map, which requires to use pointers.

D = D + M (hack)

Add D, Adder (typical machine language)

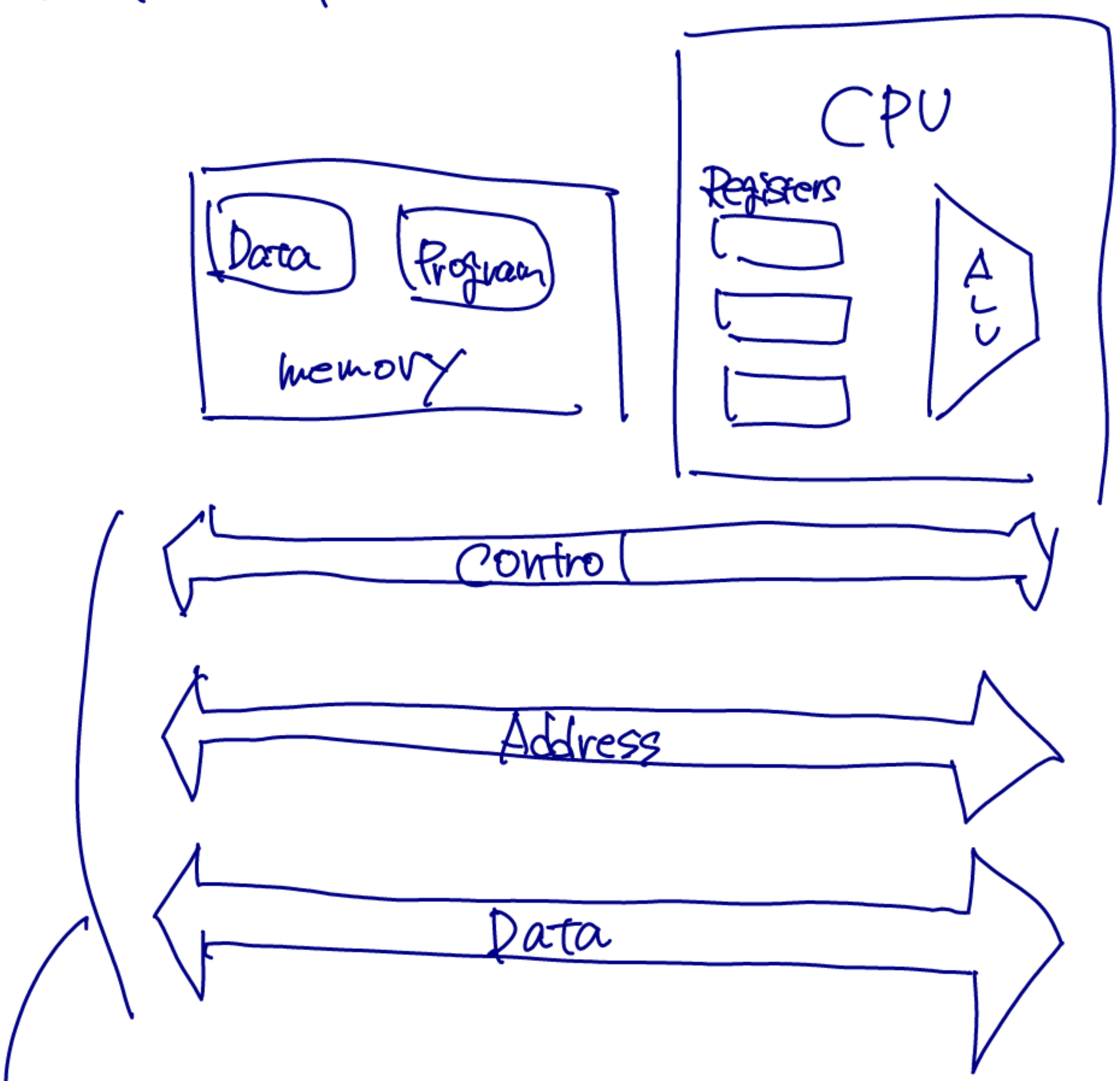
Hackは16-bitアーキテクチャ 一般的なマシンは

32bit, 64bitなのでオペレーションコードとDataを

保持するのに充分なスペースをもつ

Von Neumann Architecture

Same hardware can run many different software programs

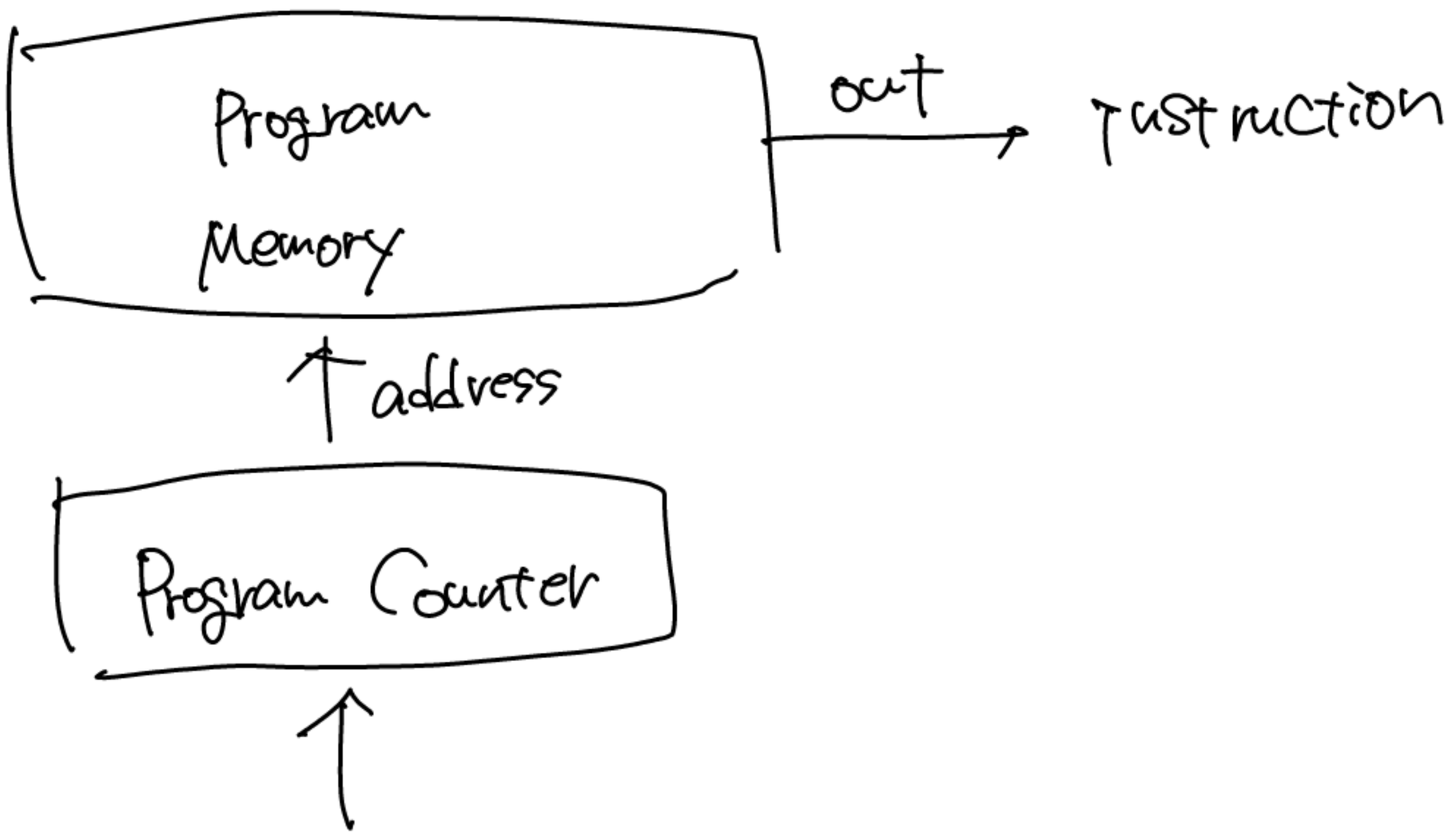


つのタイプの情報が存在する

(Bus)

The Fetch - Execute Cycle

- Fetch instruction from the program memory



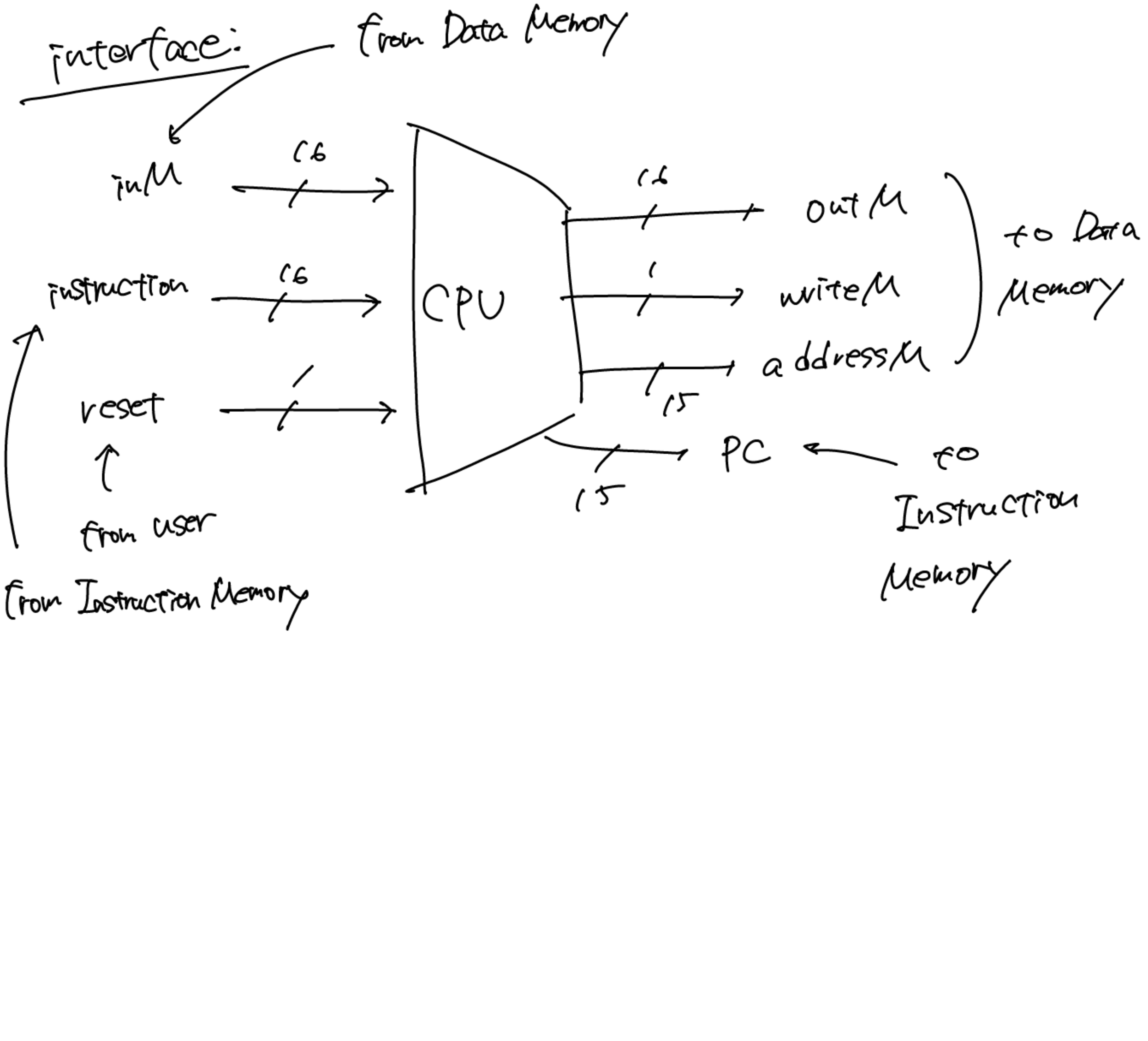
- Executing

The instruction code specifies "what to do"

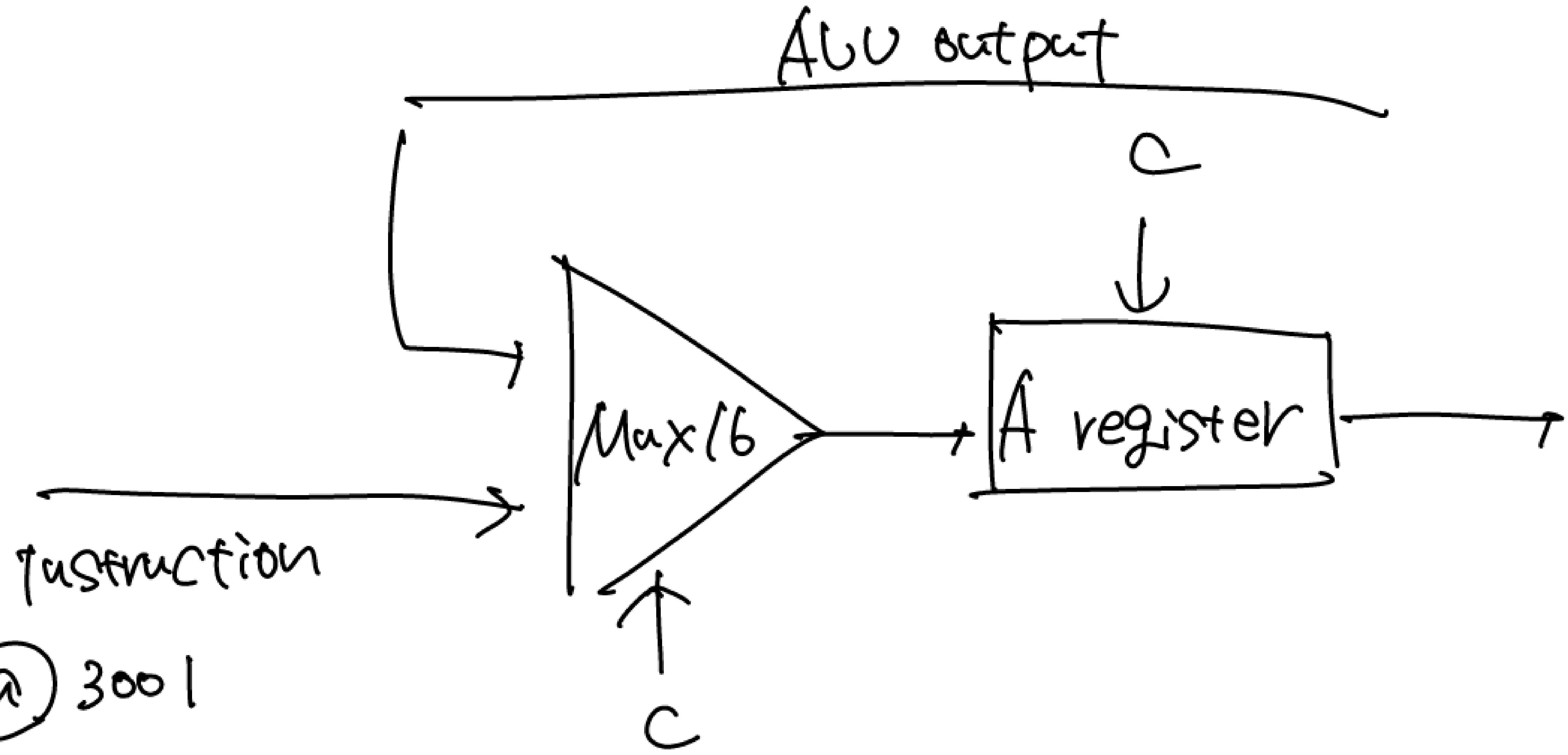
- operation
- what memory to access
- if / where jump

The Hack CPU:

- Execute the current instruction
- Figure out the next instruction



Instruction handling



@3001

0000010111011001

op-code

- Decodes the instruction into op-code + 15 bit

- A instruction では 15bit value を A register に Store

C instruction の場合

1110011111010111

op-code

- Decodes into
  - op-code
  - ALU code bits
  - Dest
  - Jump bits

ALU operation

input は D, A, inM から取る
(register)

operation は instruction の Control bits を
参照する

output は D, A, outM に流す

どこに行くかは instruction の destination bits を
参照する

Program Counter

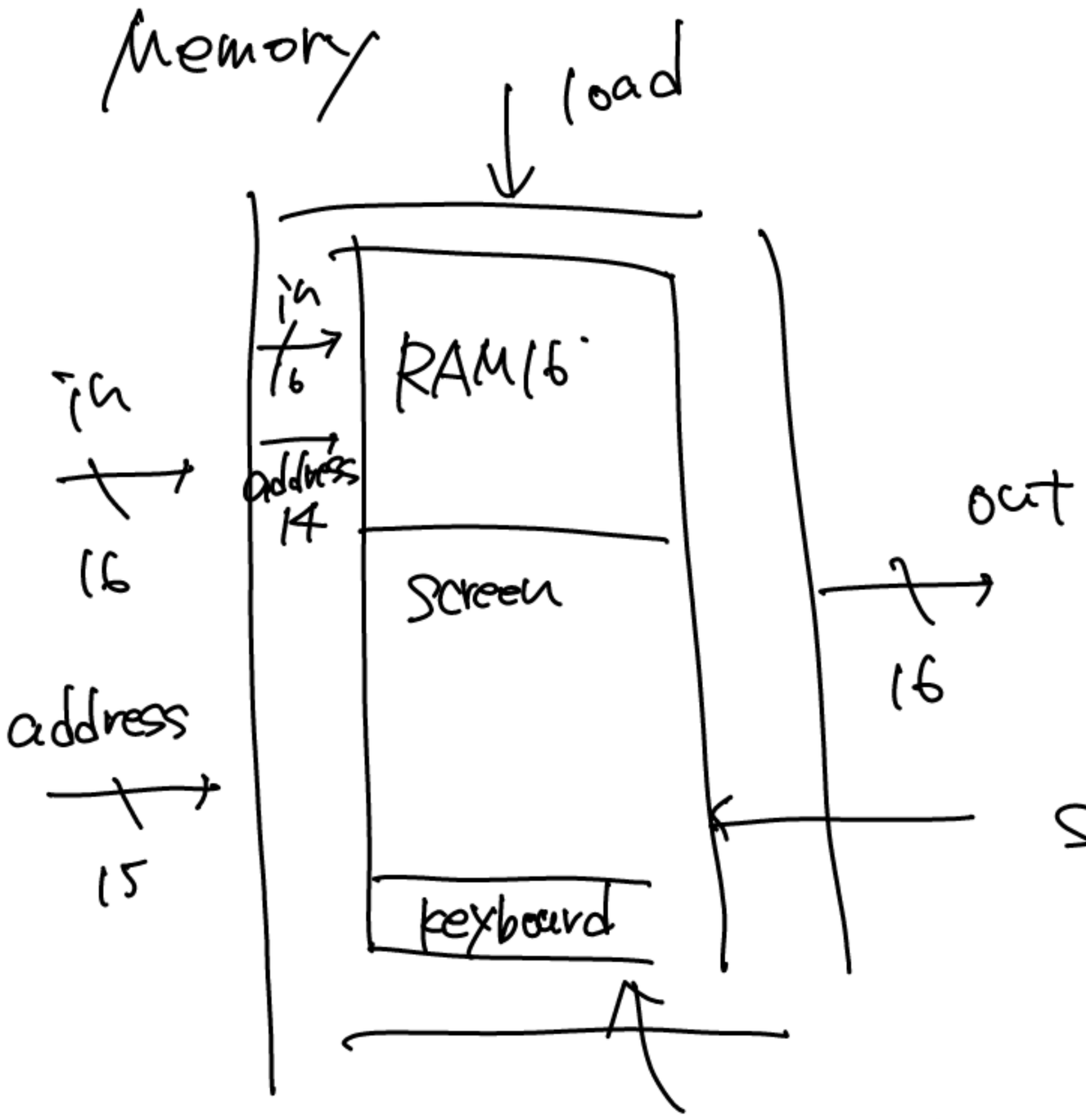
- Emits the address of the next instruction

  start 又は restart → PC = 0

  no jump → PC++

  goto → PC = A

  conditional goto → if true PC = A else PC++



Rom 32k

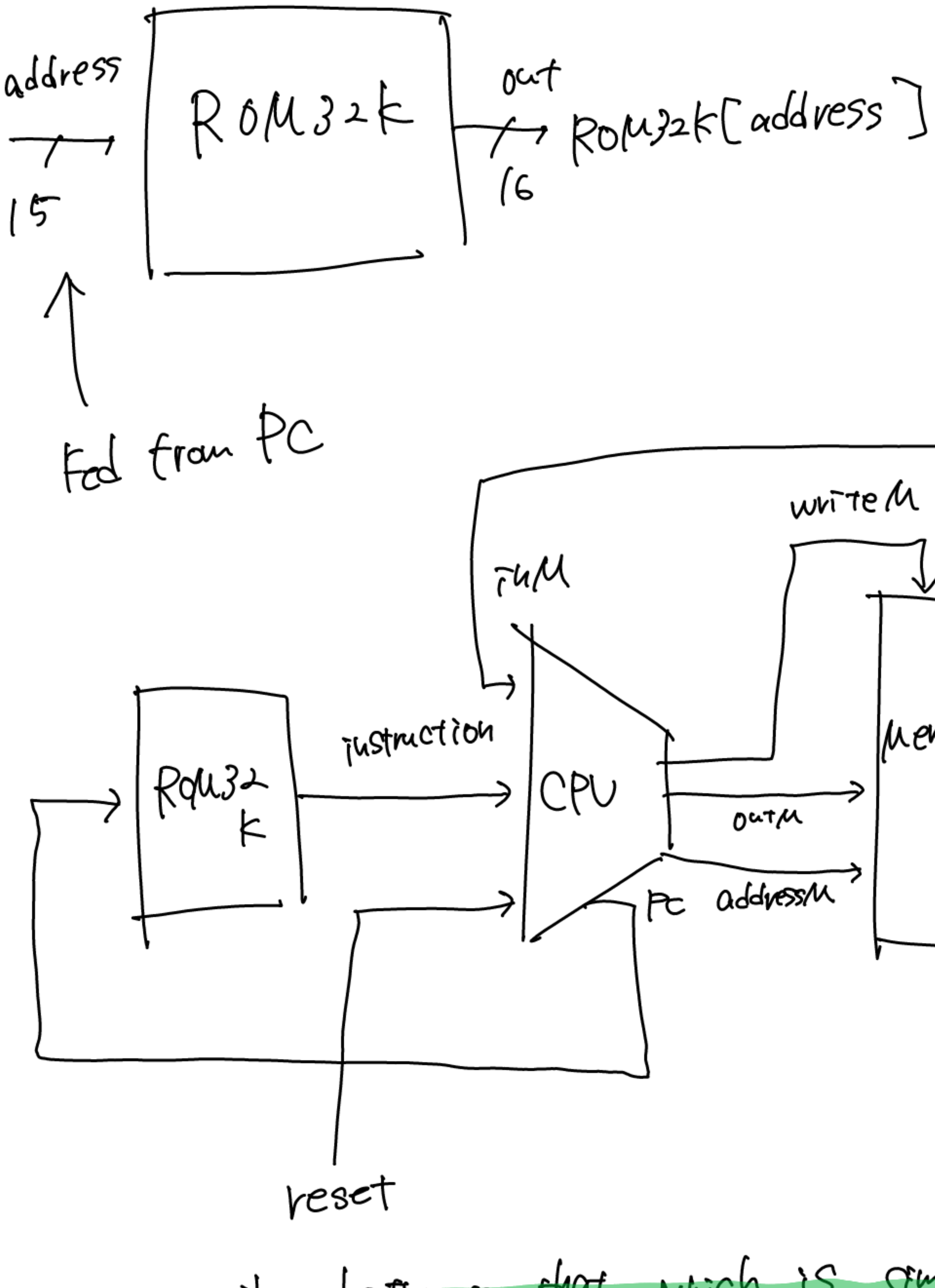
instruction memory.

↑

load → press start → program starts

どうやってロードする？

- Hardware に実装：ブートローダーに記述
- Hardware simulation: programs are stored in text. programs loading is emulated by the built-in ROM clip.



We ascribe beauty to that which is simple.

Ralph Waldo Emerson

0 - 16383

14 13

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

↑ 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1

18364

00 RAM

01 RAM

10 screen

11 KBD

screen

16384 - 24575

↓

8191 addresses

2^12

RAM

0 — 16383

2^14

Module 6: Assembler

アセンブラ、symbolic machine language を
0と1のバイナリ表現に置き換える
プログラムのこと

*.asm ——→ *.hack

This is software!

The First layer above the hardware.

Basic logic

1. read Assembly language command
   into an array
2. Break it into the different fields

3. Lookup the binary code for each
4. Combine codes
5. output in specified format

Symbols

Lables : JMP loop

Variables : Load R1, weight

どうやって アセンブルする？

↓

アセンブラは names と アドレスに 値を 探える

Symbol	Address
weight	5782
loop	673

Table を 持って Symbol と アドレスを 管理.

↳ どうやって？

↓

Allocation

↙ これら 以前に Symbol が すでに allocate されて いない か 確認

1. unallocated memory cell を 探す

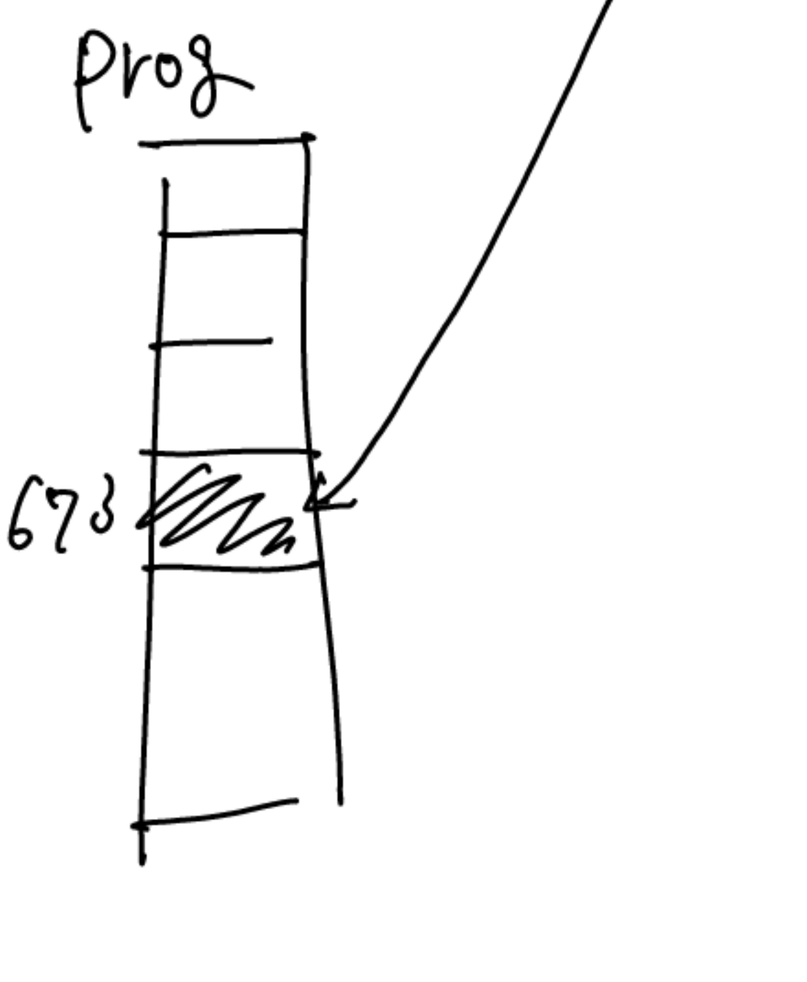
2. Table に 記録

Lables の役割

実行時の Program ラベルを Table に登録する

Lable loop:

Add 1, R1



Symbol	Address
loop	673

Forward references

Program label 宣言の前に実行されていたら

どうするか？

—

JGT count

—

Label count

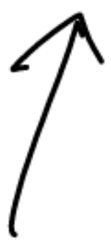
—

Possible Solutions

- Leave blank until label appears, then fix

これは複雑に
なりやすい

- 2回パスして最後の1つで
全てのアドレスアロケーションを完了させる



こっちがシンプル

Handling Instructions

A - instruction

@ value

0 value In Binary

- Value to decimal constant (15-bit)
- If value is symbol, later

C - instruction

dest = comp ; jump

例)

MD = D+1 null

dest comp Jump

1. 111 を記述
2. comp を参照
3. dest を参照
4. Jump を参照

Handling symbols

- variable symbol
- label symbol (destination of goto)
- pre-defined symbol

ⓐ preDefined Symbol

定数に置き換える，Assembler と同時

label symbol

(xxx) pseudo-command

を見つけて Table に追加

Symbol	value
LOOP	4
STOP	18
END	22

ⓐ label Symbol を Table の値に置き換える

Variable symbol

@xxx is not defined as pre-defined or label

↓

@xxx is a variable

Each variable is assigned a unique address

starting at 16 why? explain later, consider it arbitry

symbol	value
i	16
sum	17

@variableName ⟶ @variableValue

変数名から

Aインストラクションにする

Symbol Table の作り方

0. pre defined symbol を登録 (予約語)

FirstPass 1. label symbol を登録

(xxx) declaration を lookup する

Second pass 2. Symbol Table に 使われる variable を

Symbol Table に 追加

Reading and Parsing

no need to understand meanings !!

- 打ち込んだTokenに従うだけ

Mnemonic to Code

no need to worry about how
mnemonic fields were obtained !!

Symbol Table

No need to worry about the
what these symbols mean

Proposed software architecture

- Parser
  unpacks instructions into fields
- Code
  Translate field into binary value
- SymbolTable
  manages the symbol table
- Main
  initializes the I/O files and drives process

∅ // abc

r (a) 2

L D = A

) 3

head = 0 ⟵ pos = 6

Perspective

Macro - command

D = M [100] ⟶ { @100 / D = M

actual

Jump Loop ⟶ { @Loop / JMP

アセンブラを更新することで（マクロ展開）

マクロを定義して最終成果物であるBinaryは

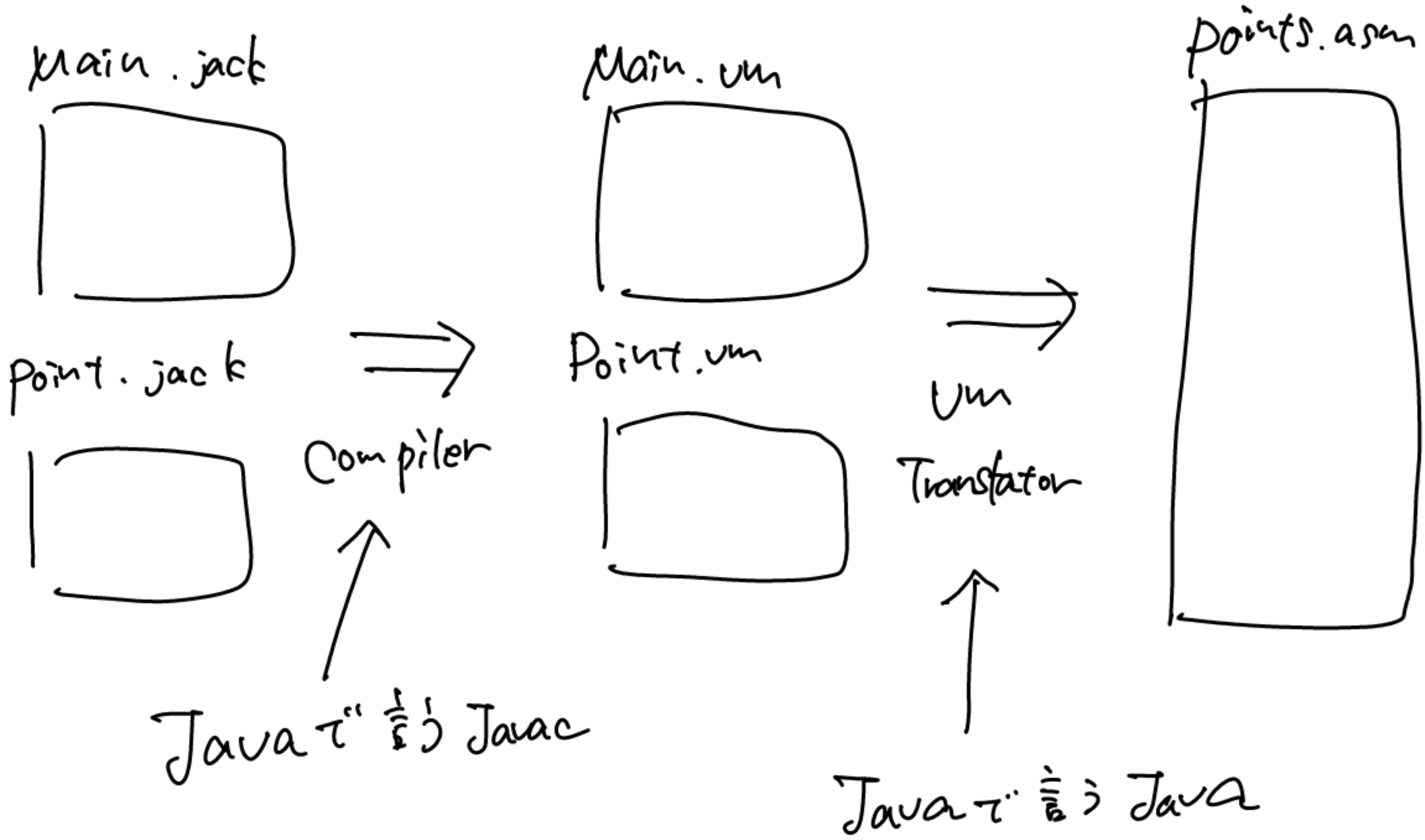
同一だけどアセンブラの表現をシンプル、理解しやすくできる。

How the first assembler written without high-level programming language?

Start writing assembler in high level language
↳ translate to machine language by hand
(only first time)
↓ so that
the assembler translates
other programs.

Module 1 : Virtual Machine 1 : Stack Arithmetic

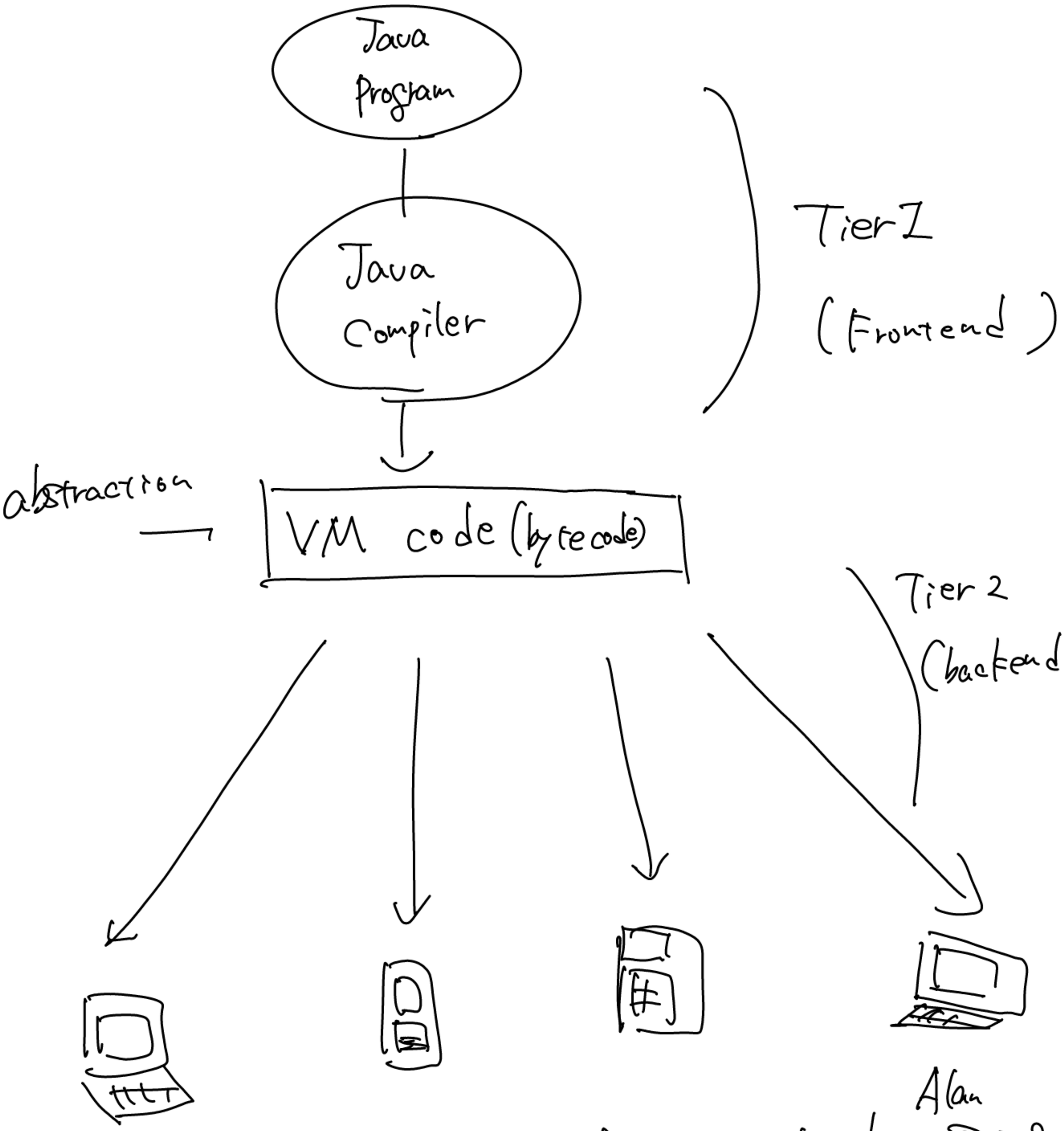
high level to low level



Part 2 is demanding than Part 1.

more

"Write once, run anywhere" — 90 years old idea



"We can only see a short distance ahead, but we can see plenty there that needs to be done"

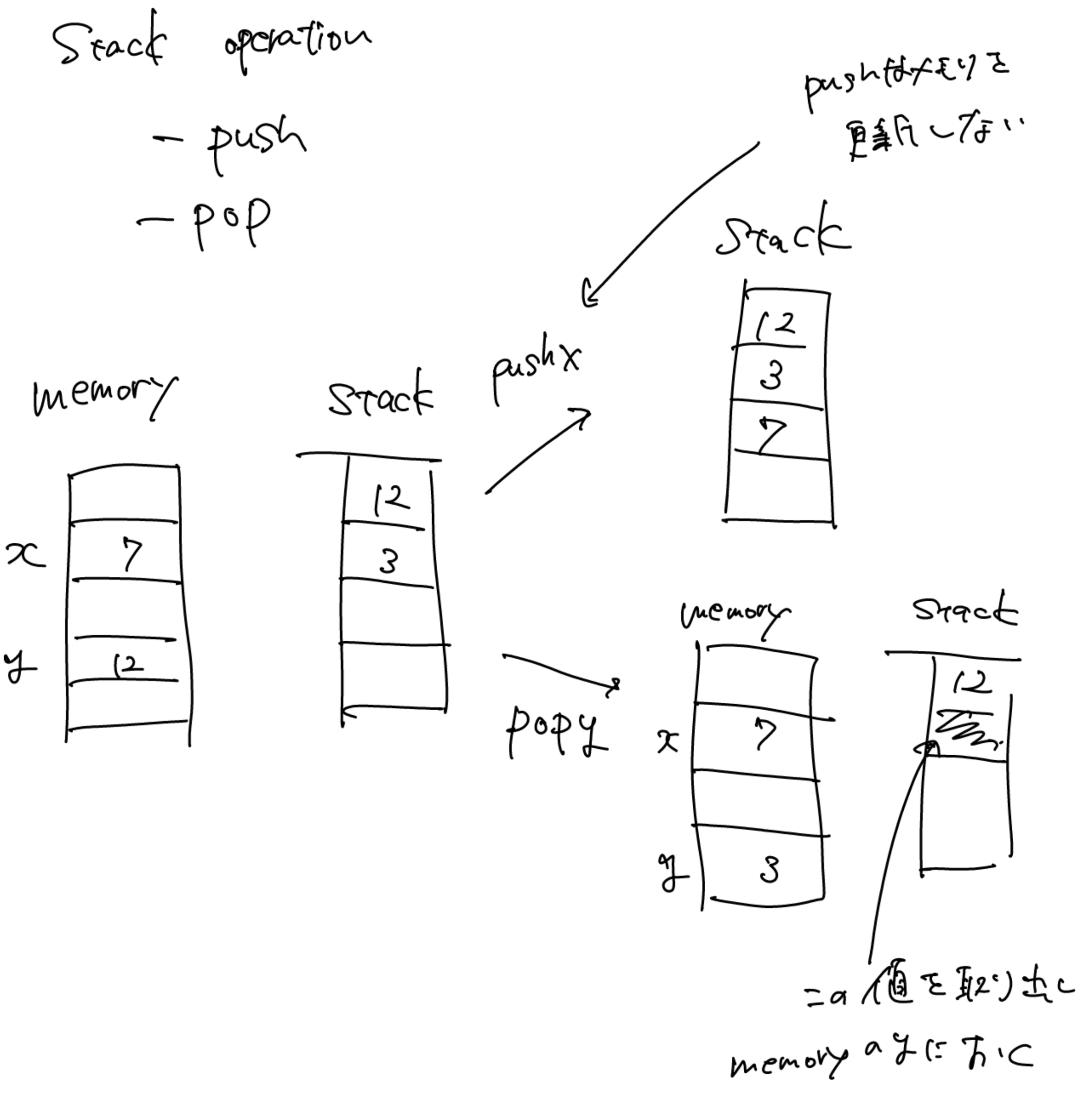
Alan Turing

Stack machine

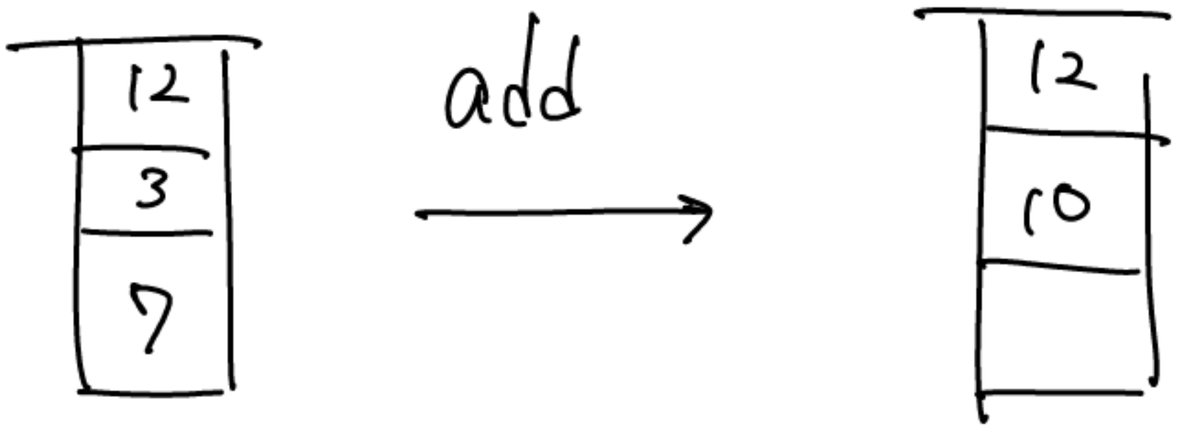
VMコードを Compiler と VM implementation の
どちらに記述するかというバランスを考えたとき
いい バランスと考えられているのが Stack machine

Stack operation

- push
- pop



Stack arithmatic

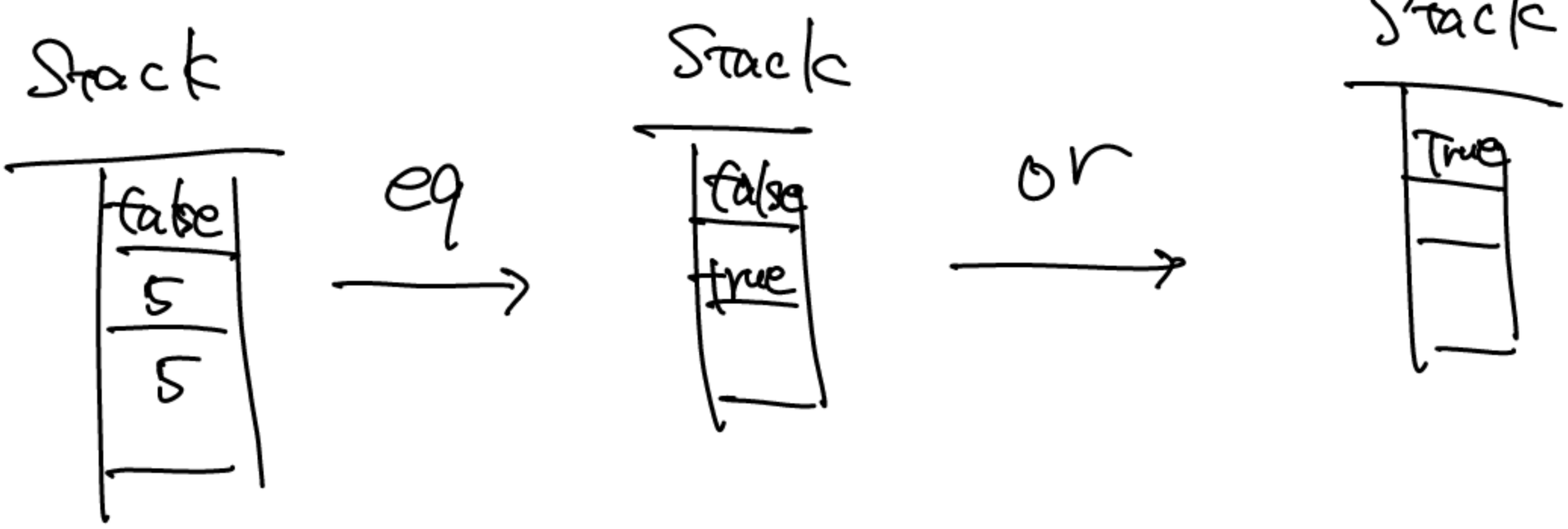


要素を上から2つ取り出して合わせてpushする

一番上を負の値に書き換える

Applying a function f on the stack:

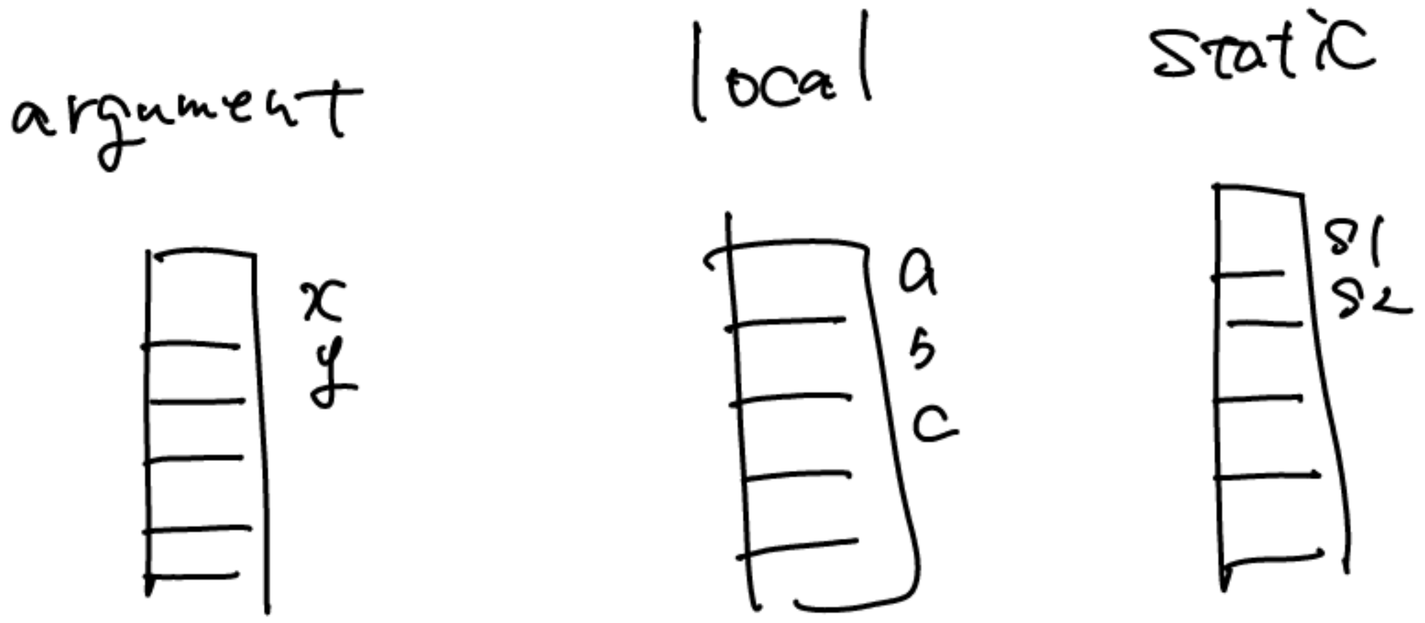
- pops arguments from the stack
- compute f on the arguments
- push the result onto the stack



Variable kinds

変数は static, local, argument という
種類がありそれぞれの種類を VM code 上で
表現するためにメモリのセグメンテーションを使う。

Virtual memory segments



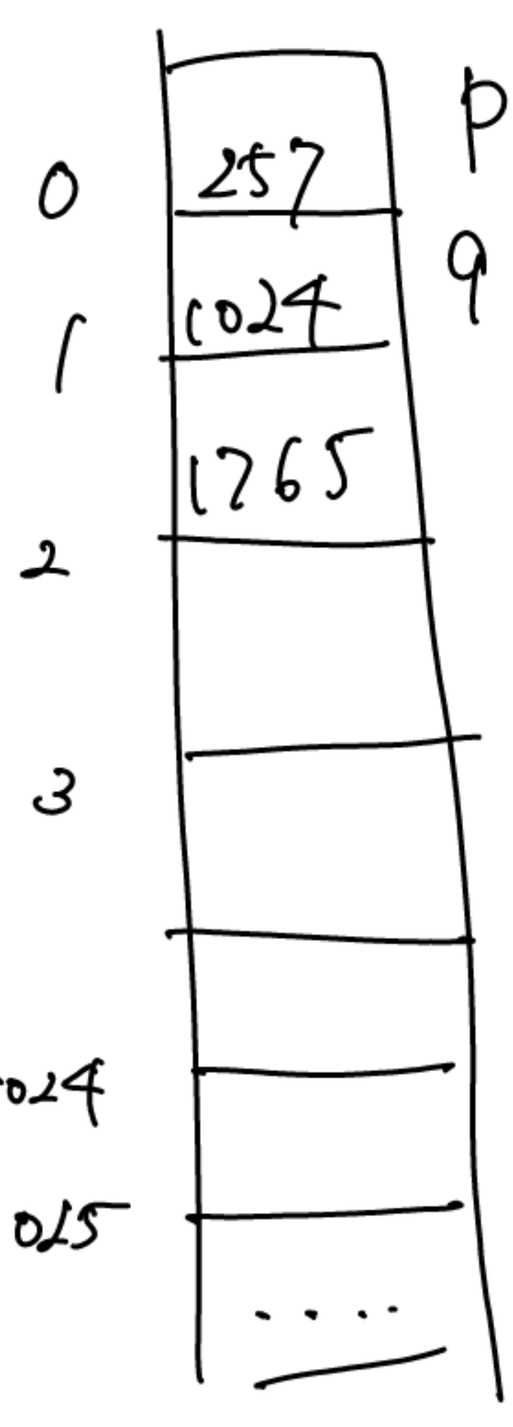
- push constant 17 ← segment と定数
- pop local 2

D = *P ⟶ ⓐ P / A = M / D = M

in hack

P-- ⟵ pointer をデクリメント

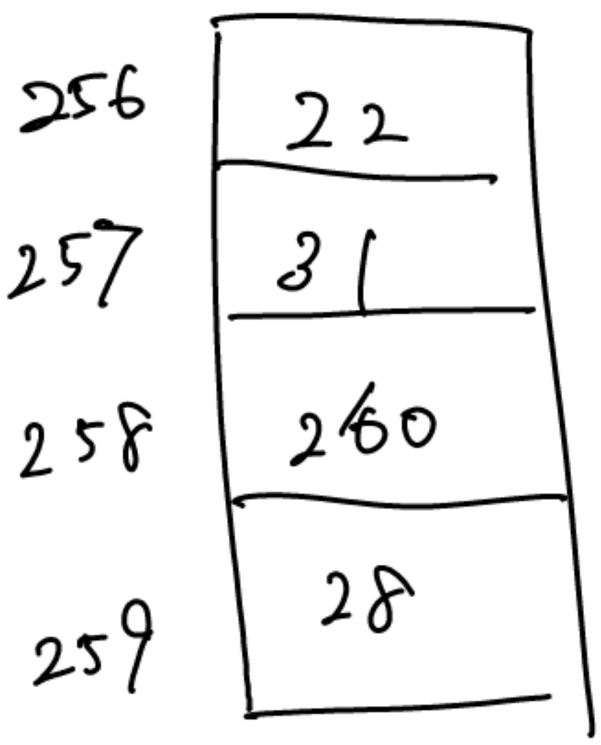
RAM



pとqはポインタ，

参照すべきデータのアドレスを

格納している



p1 : 256

p1 = 259

*p1 = *p2 + 3 // 295 ← 31

p2 = p1 - 2 // 257

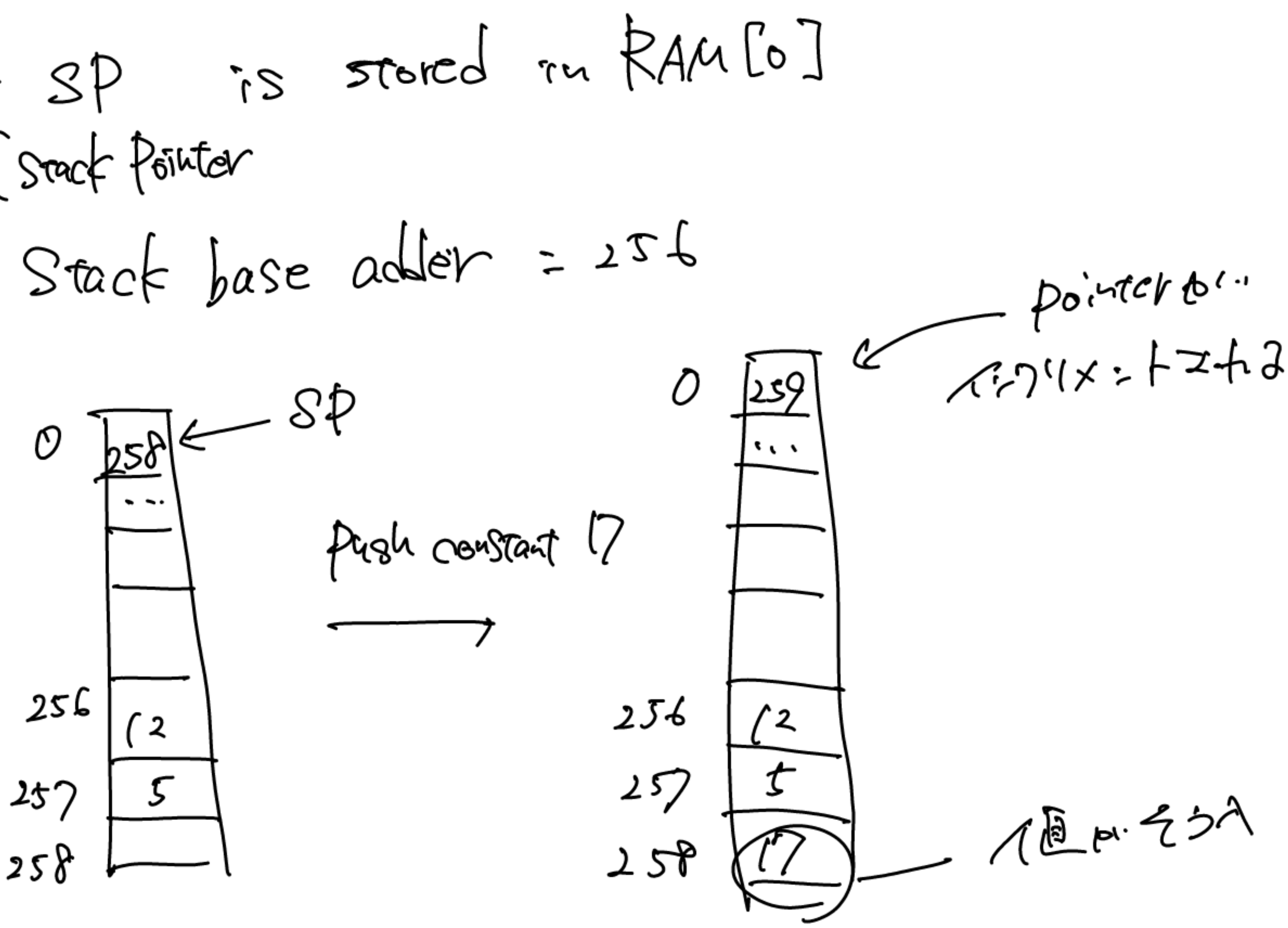
p1 -- // 258

$$*\underline{(p2+7)} = 260 + 31$$

258

Stack Machine Implementation

- SP is stored in RAM[0]
  (Stack Pointer
- Stack base adder = 256



↓

*SP = 17

SP++

@17 // D=17

D=A

@SP // *SP=D

A=M

M=D

@SP // SP++

M=M+1

Implementing local (segment)

SP pointer points to stack's top value

LCL, a pointer of local, points to base address

オフセットを使わずにデータにアクセスする

addr = LCL + 2, SP--, *addr = *SP

↳ Hack Assemblyでどう書くか

@LCL
D = M + 2
@addr
A = D
@SP // SP--
M = M - 1
D = A // *SP
@addr // *addr = *SP
A = M
M = D

Implementing local, argument, this, that

同じように LCL, ARG, THIS, THAT という4つの

ポインタを RAM[i] から保持する

implementing constant

push constant i ⇒ *SP = i, SP++

Implementig static

- static variables は すべてのメソッドから 見えるように するべき

↳ つまり global な空間に 置く必要がある

RAM[16] ~ RAM[255] を使う.

例えば"

pop static 5 ⇒ ⓐ foo.5 ← 実体は 任意の RAM を参照する

M = D

temp segment

- Sometimes need to store temporary values

Mapped on RAM 5 to 12.

(localと一緒の実装)

ここがポインタ参照しなくなっただけ

push temp i ⇒ addr = (5) + i, *SP = *addr,

SP++

pointer segment

コンパイラ実装しないとなぜこのセグメントが

必要か理解しずらい

→ keep track of the addresses of the "this" and "that" segments using the "pointer" segment

ここでは 0/1 しか使わない

0 → THISにアクセス

1 → THATにアクセス

push pointer 0/1

pop pointer 0/1

push pointer 0/1 ⇒ *SP = THIS/THAT , SP++

pop pointer 0/1 ⇒ SP--; THIS/THAT = *SP

VM Translator Proposed Implementation

usage

" java VMTranslator myProg.vm "

→ generate myProg.asm

- Parser : parse VM commands into its lexical elements
- CodeWriter : writes assembly code from parsed command
- Main :

1. parse
2. pass parsed elements to code writer
3. output

Perspective

VMはいつから?

1970年代 パーソナルコンピューターが普及

(Apple, IBM) ← different CPU
Chipset
OS

当時 Pascalが主なプログラミング言語

Pascal compilerがそれぞれに必要

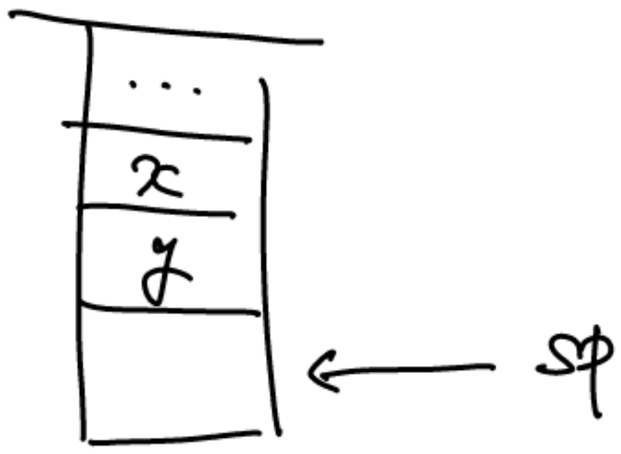
Apple, IBMは競合なのでPortabilityを提供することに興味はない状況

↓

P-code (初期のVM framework)
中間言語

and (x And y)

@SP
M = M - 1
D = M
) D = y



@SP
M = M - 1
A = M
M = D & M
) *SP / x = y and x ← D | M := C | S

@SP
M = M + 1
) SP++

OR

not

@SP
M = M - 1
A = M
M = !M

add (x+y)

@SP
M=M-1
D=M
@SP
M=M-1
A=M
M=D+M
@SP
M=M+1

sub (x-y)

@SP
M=M-1
D=M
@SP
M=M-1
A=M
M=M-D
@SP
M=M+1

neg (-y)

@SP
M=M-1
A=M
M=-M
@SP
M=M+1

eg

@SP
M=M-1
D=M
@SP
A=M
M=M-D
@TRUE_i
D;JEQ
@False_i
D;JMP
(TRUE_i)
@SP
M=M-1
A=M
M=-1
@SP
M=M+1
@RETURN_call_i
(RETURN_CALL_i)

D=x-y
(False_i)
@SP
M=M-1
A=M
M=0
@SP
M=M+1
@RETURN_call_i
(RETURN_CALL_i)

Jump 先は戻るアドレス

push local 3

@LCL

A = A + 3 (?)

Base 3

2

112

84

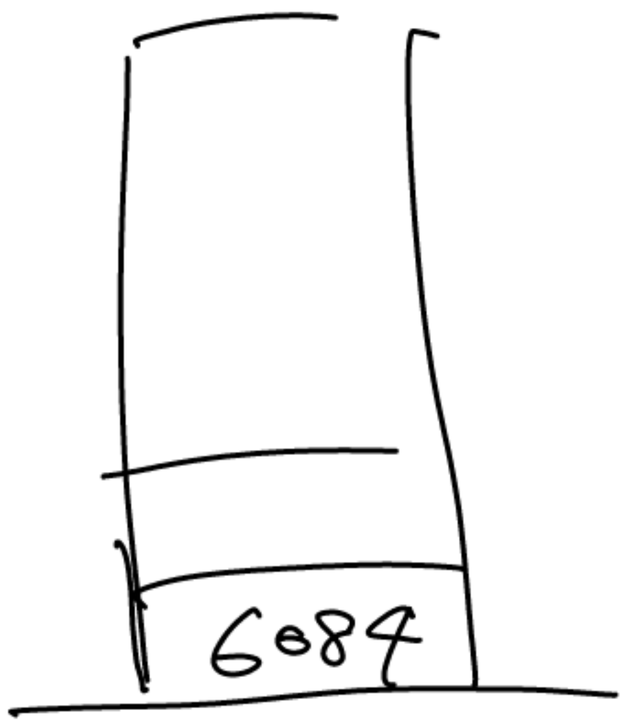
28

57 → 31 → 53

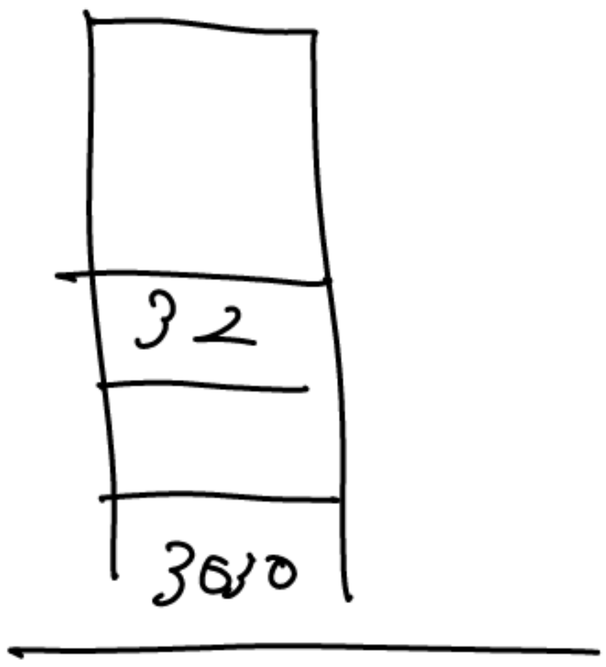
add → 57 → 84 → 57 → 84 → 112

57 → neg → 57 → 28 → and → 85 → 82

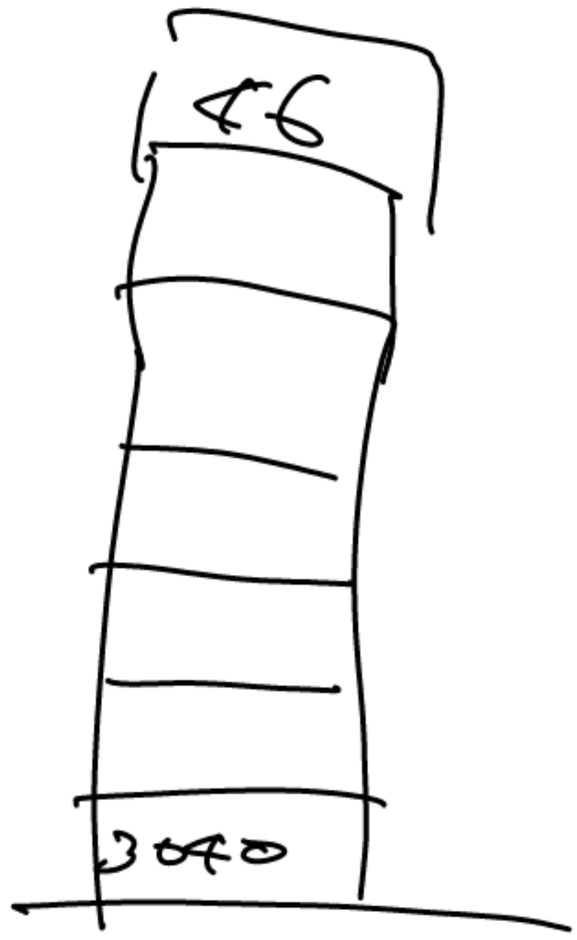
→ 28



Constant



This



That

Module 2 VM. Program Control

Functions are abstractions

↖ what about their implementation?

> Basic language can be extended at will

Branching commands

- goto
- if-goto
- label

function commands

- call
- function
- return

Branching

- unconditional
- conditional

goto label (unconditional)

→ label へとプログラムカウンタを Jump する

if-goto label (conditional)

cond = pop

if cond = true, label へとプログラムカウンタを Jump する

condition は事前に stack に push されている

Functions in VM language

Sqrt (x - 17 + x * 5)

↓

```
push x
push 17
sub
push x
push 5
call Math.multiply
add
call Math.sqrt
```

part of OS

関数を実行すると主は事前に

引数を Stack に Push する。

↓

関数の返り値は Stack よの引数と入れ替わる。

Executing:

```
push constant 8
push constant 5
call mult (2)
```

caller ⟷ callee

← この2は関数に
いくつstackの値を
渡すかしめして…

(2) なので ~~5, 8~~

Function Call and Return Implementation

Functionを連続して呼ぶことを Calling chain と
言う　　foo > bar > sqrt > ...

※ Functionは独自のStateを持つ

How to maintain the states ?

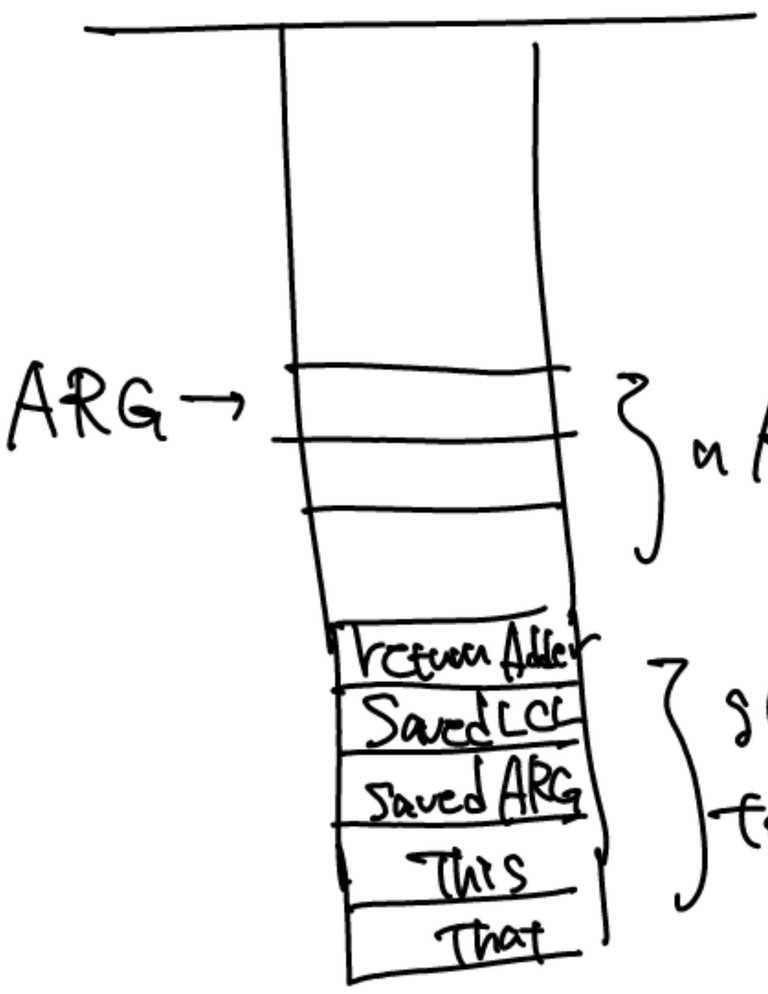
The calling pattern is LIFO

↑

yes, that's Stack !!

---

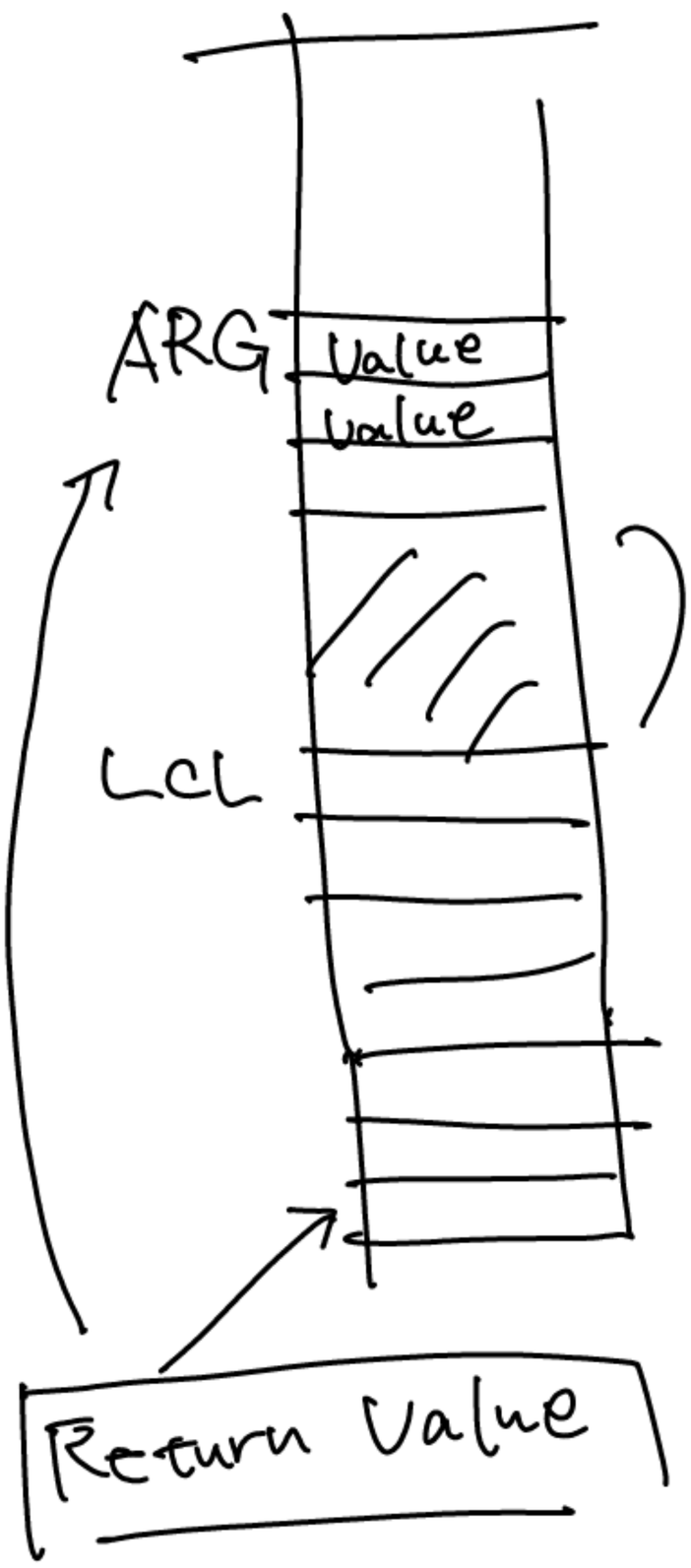
call foo nArgs



VM Implementation

1. set arg address
2. saves the caller's frame
3. Jump to execute foo

Called function



VM Implementation

1. set up local variables
2. Put return value
3. replace 2 value with ARG 0
4. Restore segment pointers of the caller
5. Clears the stack
6. sets SP for the caller
7. Jump to return address

Handling call

push return Address
push LCL
push ARG
push THIS
push THAT

ARG = SP - 5 - nArgs

LCL = SP

goto functionName

(returnAddress) // Declare a label for return address

save frame

Handling function

(functionName) ← Declare label

repeat nVars times:
push 0
) initialize the local variables to 0

Handling Return

endFrame = LCL
retAddr = *(endFrame - 5)
*ARG = pop() ← return value が stackの一番上にあるので

SP = ARG + 1 ← SP以上は dead values になる

THAT = *(endFrame - 1)
THIS = *(endFrame - 2)
ARG = *(endFrame - 3)
LCL = *(endFrame - 4)
) restore caller's values

goto retAddr

Booting (Hack Platform)

1. VM Program Convention

   main 関数を もつ Main.vm が 必要

2. VM Implementation convention

   VM Implementationが 仕込まれると

   Sys.init ⟶ Main.main という 実行が

   行われる

// Bootstrap code

SP = 256

Call Sys.init

Module 3

The jack program is collection of one or more Jack classes.

Array

a part of class

* not type

primitive types

- int
- char
- boolean

class type は User で任意に作れる

List processing

List とは

- null か
- an atom, followed by a list

(atom, list)

(5,)

- null
- (5, null)
- (3, (5, null))

---

do v.print()

とは v を print 関数に渡してほしい

implicitly vオブジェクトは this

Graphics Optimization

- Hack Ram に書き込むことで
  Display に表示できる

OS memory class APIs

- peek
- poke

OS screen class API

- drawline
- drawPixel
- drawRectangle

複雑な image を描くには?

drawPixel? 例えば1つの split を書くのに
75回の書き込みが必要
↓
OS の drawPixel は Memory.peek, poke の
API を使っていてその処理(機械語)は
40ものstepが必要 75×40!! inefficiency

Custom drawing

16 bit の値を書き込むことで、オペレーション回数を

1 Row

減らす。

d. Memory.poke(addr0, 4064)

↳ これは機械語で4cinc

ⓐ 4064

D = 4

ⓐ addr0

M = D

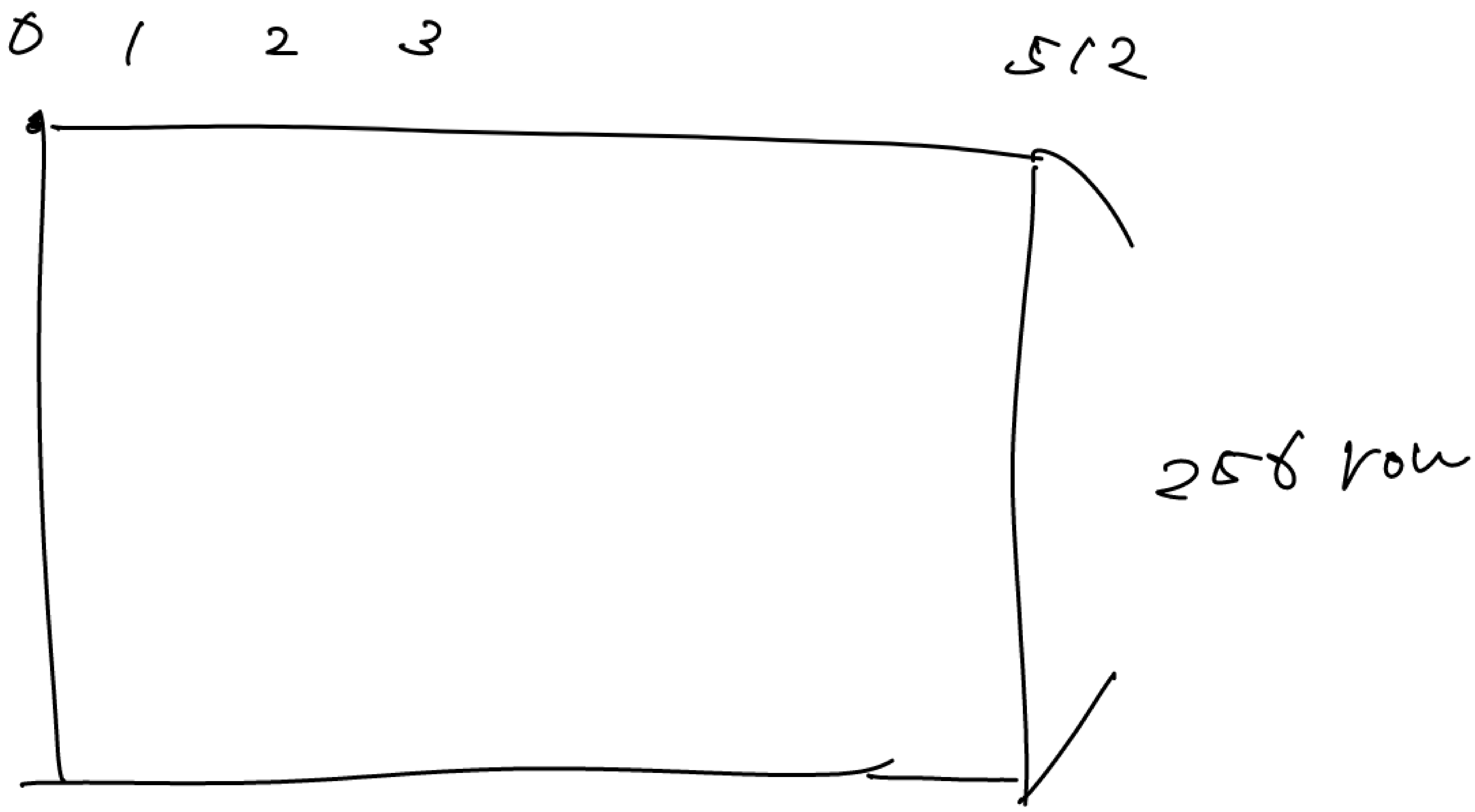
Perspective

What is weak type language?

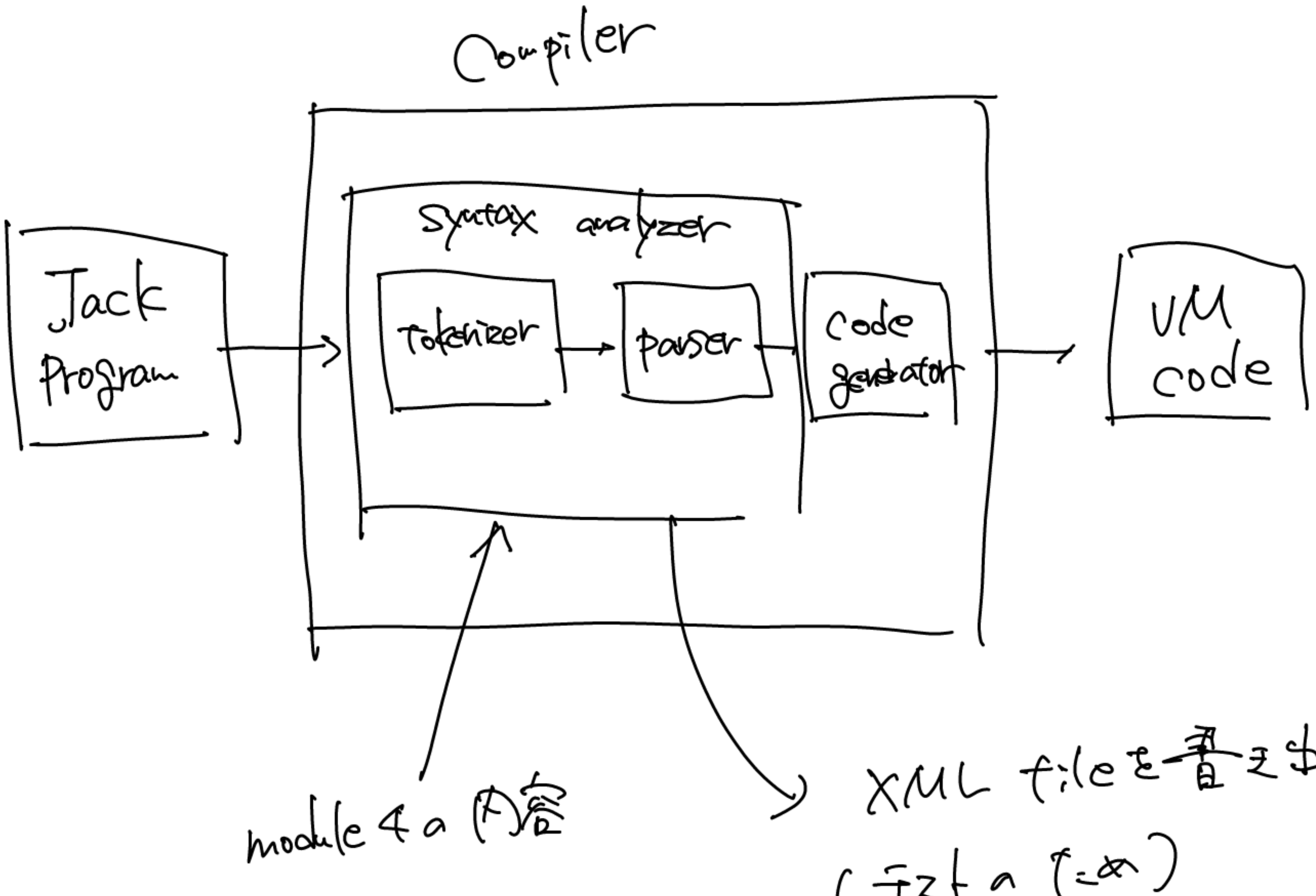
- regulating casting is headeck
- allow us to take a control of underneath machine.

↑ like C implemented Unix.

↑ this is why many OS are developed in C++, which enables developers to access to hardware easily.


0 1 2 3
512
256 row

Module 4: Compiler I: Syntax Analysis



module 4 a 内容

XML file を書き出す
(テスト a 下)
project 10

Lexical Analysis

Tokenizing = grouping characters into tokens

input → tokenized input

```
if (x < 0) {   ⟶   if
                   (
                   x
                   <
                   0
```

A programming language specification must document its allowable tokens.

Jack tokens

- keyword
- symbol
- integerConstant
- stringConstant
- identifier

pseudocode

```
tokenizer = new Tokenizer ("file.jack")
tokenizer.advance();
while (tokenizer.hasMoreTokens()) {
    classification = tokenizer.classification();
    print classification;
    print value;
    print newLine;
    tokenizer.advance();
}
```

Grammer

a set of rules, describing how tokens can be combined to create valid language constructs.

rule には 2 種類

- terminal rule ⟶ constant だけを含む、
- Non-terminal rule ⟶ それ以外のルール

(5'1)

verb: 'went' | 'ate' | 'said'

Jack grammer

Statement : if Statement | while Statement |
let Statement

statements : statement * ← 0 or more

if Statement : 'if' '(' expression ')'
'{' statements '}'
while, let は以下同様

expression : term (op term)? ← 0 or 1

term : varName | constant

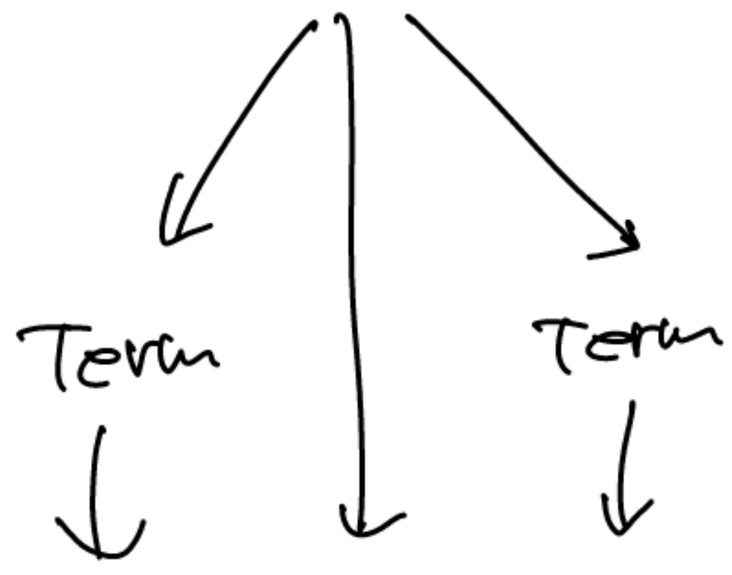
varName : a string not begining with a digit

constant : decimal

op : +, -, =, >, <

Parse Trees

count + 1 ⟶ (Parser)



count + 1

木構造を使って構造を表現する

Parser Logic

Parser は 各 statement を コンパイルする
method を 持つ。

```
class Parser {

    compile Statements () { }

    compile Term () { }

}
```

logic

1. Follow the right-hand side of rule, and parse the input accordingly
2. IF right-hand side specifies a non-terminal rule XXX,

Call compile XXX

LL grammer : can be parsed by a recursive descent parser without backtracing

↑ friendly grammer!

never have to go back

LL(k) parser : a parser that needs to look ahead at most k tokens in order to determine which rule is applicable.

↑

Jack's grammer is LL(1)

Jack Analyzer

How to unit-test Jack Analyzer, which runs tokenizer and parser?

↑

output XML code

Example) return x;

<returnStatement>

<keyword>

return

</keyword>

<expression>

<term>

<identifier> x </identifier>

</term>

</expression>

<symbol> ; </symbol>

</returnStatement>

Proposed Implementation

- JackTokenizer
- Compilation Engine
- Jack Analyzer (most top module)

Input could be
  - a file → [file].xml
  - a directory → [dir].xml

Perspective

- ハスケルでは Error Handling は扱わない

↑

実装は複雑になる.

Compiler II : Code Generation

Objective : developing a full-scale compiler

Compilation challenges

- Handling variables
- " expressions
- " flow of control
- " objects
- " arrays

VM language は Push, pop といった シンプルな 処理しか できないので Jack と VM language の ギャップを 埋める のは 簡単では ない。

Handling Variables

We have to handle

- class-level variables
  - field
  - static
- subroutine-level variables
  - argument
  - local

↑

variables have properties:

- name (identifier)
- type (int, char, boolean, class name)
- kind (field, static, local, argument)
- scope (class level, subroutine level)

↑

- needed for code generation
- should be managed in symbol tables

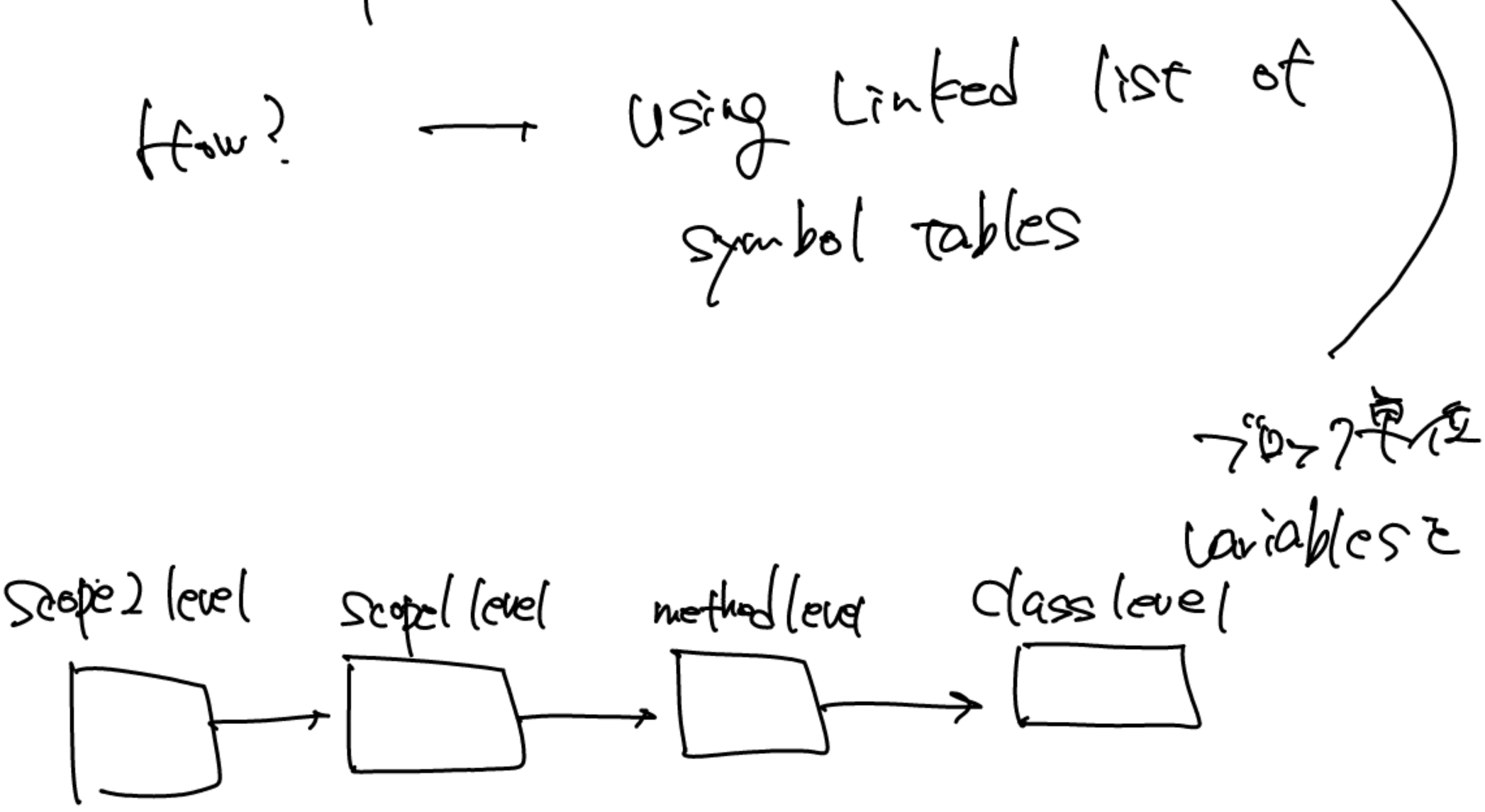
Nested scoping

- Some language, such as Java, supports unlimited nested scoping

↑

How? ⟶ Using Linked list of symbol tables

→ "ブロック単位で" variablesを宣言できる

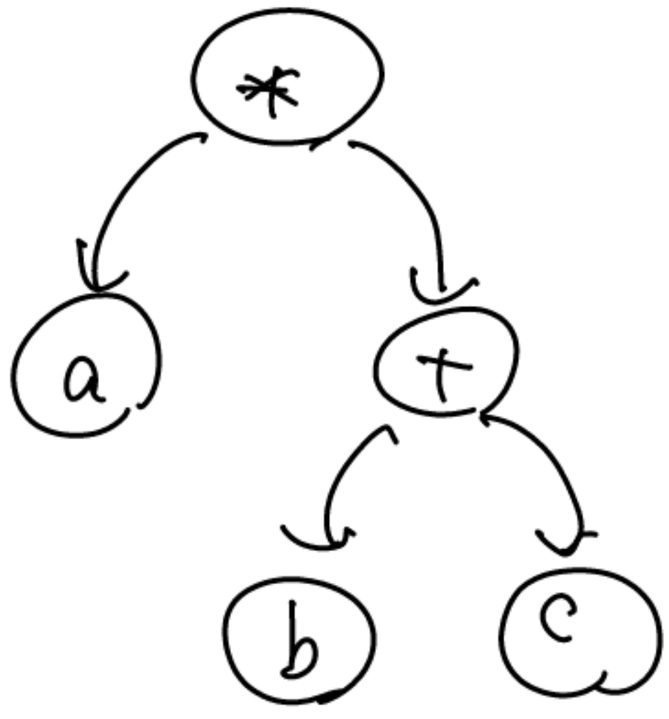


この方向にlookupする

Handling Expressions

- Infix (human oriented)

  a * (b + c)



人間は上a式で例えば (b+c) を b plus c と
読む。c つぎに b とは読まない。
(Infix と呼ばれる)

- prefix (functional)

  * a + b c

- postfix (stack oriented)

  a b c + *

Soure code

Parse Tree

code gen

stack-machine code

Handling Flow

Statements のハンドリングについて

特に If と while がキモになる。

Compiler の仕事はユニークなラベルを生成すること。

(If, while の条件 expression は Negate して

false のとき True 又, True のとき false 又に入れ替える。

実装がシンプルになる)

Handling Objects : Construction

- caller
- callee

) 2つの perspective がある

caller perspective

var Point p2, p1

var int d;

// No code is generated, just update symbol table

let p1 = Point.new (2, 3)

push 2

push 3

call Point.new

pop p1 // p1 = base address of the new object, so it should be assigned to p2

Caller perspective

// constructor Point new (int ax, int ay)

↑

The compiler creates the subroutine's symbol table.

push 2 (x and y)

call Memory.alloc 1

Objectを作成に必要なサイズを計算する（この場合は

xとyとfield数：2） OSのAPI, allocを使って

memory blockを取得する

pop pointer 0 // allocで取得してきたaddressを

This のベースアドレスにする

// let x = ax; let y = ay;

push argument 0

pop this 0

push argument 1

pop this 1

```
// let pointCount = pointCount + 1;
push static 0
push 1
add
pop static 0
// return this
push pointer 0
return
```

Objects Manipulation

Caller

OO style

```
p1.distance(p2)
p2.getx()
obj.foo(x1, x2 ...)
```

Procedural style

```
distance(p1, p2)
getx(p1)
foo(obj, x1, x2 ...)
```

Object の 第1引数とした method に わたしたい

```
obj.foo(x1, x2)
```

```
push obj
push x1
push x2
call foo
```

// method int distance (Point other)

// var int dx, dy

↓

creates symbol table, no code is generated

this	Point	argument	0
other	Point	argument	1
dx	int	local	0
dy	int	local	1

Caller がPushしたオブジェクトのアドレスを

This のベースアドレスにセットする

push argument 0

pop pointer 0 // THIS = argument 0

// let dx = x - other.getX()

push this 0

push argument 1

call Point.getX

sub

pop local 0

void メソッドの場合 return は

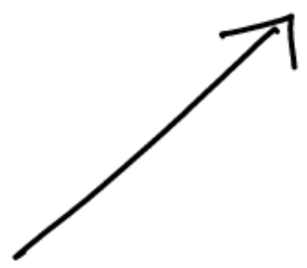
push constant 0

return

とする。Convention として 全てのメソッドは return

statement で 終わる必要がある。

constan 0 は ダミーな値



ダミーな値は callee の方である

call Point.print

pop temp 0

Handling Arrays

Construction

- var Array arr;
  generates no code
  only effects the symbol table.

- let arr = Array.new(n)
  ↑
  same with Object construction

manipulation

This	That
↓	↓
obj	array
pop pointer 0	pop pointer 1

```
// RAM[8056] = 17
push 8056
pop pointer 1
push 17
pop that 0
```

```
// arr[2] = 17
push arr    // base address
push 2
add
pop pointer 7
push 17
pop that 0
```

VM code は特定の RAM の場所を知らない…(大事)

↑ いつも 0 index を使う.

// a[i] = b[j]

Stack 1:

a[i]
b[j]

Stack pointer temp

add
push i
pop pointer 1
pop that 0
push a

Stack 2:

a[j]
b[?]

Stack 3:

b[j]

Pointer

a[j]

temp

b[j]

address

pop temp 0
pop pointer 1
push that 0

// 2

add
pop pointer 1
push that 0

// 3

pop temp 0

// arr [exp 1] = exp 2

push arr

computing and pushing the value of exp 1

add

computing and pushing the value of exp 2

pop temp 0

pop pointer 1

push temp 0

pop that 0

Perspective

1. Jack is simple, what will make language complicated?

- no type system
- no inheritance

↑ if inheritance, method call should be determined on runtime.

2. How difficult to be like Java?

inheritance ← should be difficult.

- adding control flows, such as switch or for (trivial)
- add data type (trivial)

3. Compiler optimization?

Optimize output that compiler genarates.

In Jack, optimization should be less VM codes.

Operating System

- Language extensions / standard lib
  - Math, String etc.
- System oriented service
  - memory management
  - file system
  - I/O device drivers
  - UI management
  - multi-tasking
  - Networking
  - security

) → = a JRE 実装の1つ

Efficiency Matters

全ての下で動いているので パフォーマンス 重要

例) Multiplication

```
let x : 409 * 5003;
```

↓ repitive addition

```
multiply (x. y)          ⟶ O(n) = n
    sum = 0
    for i .. y - 1 do
        sum = sum + x
    return sum
```

x	0	0	0	1	1	0	1	1	
y	0	0	0	0	1	0	0	1	i'th bit of y
	0	0	0	1	1	0	1	1	(1)
	0	0	1	1	0	1	1	0	0
	0	1	1	0	1	1	0	0	0
	1	1	0	1	1	0	0	0	(1)
$x*y$	1	1	1	1	0	0	1	1	

sum ここを足す

x を 1ビットずつずらす

w-bit numbers

```
multiply (x, y)
  sum = 0
  shifted x = x
  for i = 0 ... w-1 do
    if((ith bit of y) == 1)
      sum = sum + shifted x
    shifted x = shifted x * 2
return sum
```

→ $O(n) = w$

Running Time

C = 10

Input size	linear (cN)	log (c log$_2$N)
8	80	30
16	160	40
32	320	50
64	640	60
100	1000	70
1000	10000	100
1 000 000	10 000 000	200
1 000 000 000	10 000 000 000	300

Mathmetical Operations

Division

Naive approach → repetitive Subtraction

↑

O(N)

(N = input size)

better

algorithm

(5'J) 125 / 3

1. What is the lagest number x =

(100, 90, 80, 70, 60, (50), 40, 30, 20, 10)

$3 * x \leq 125$

2. what is the largest number x = $3 * x \leq 25$

(9, (8), 7, 6, 5, 4, 3, 2, 1)

3, 1 is less than 3

done!

```
devide (x, y)
    if (y > x) return 0
    q = divide(x, 2*y)
    if ((x - 2*q*y) < y)
        return 2*q
    else
        return 2*q + 1
```

Square Root $\sqrt{x}$

two properties (x^2)

1. its inversed function can be computed easiley

2. it is a monotonically increasing function

↓

use binary search ($\log n$)

sqrt (x) :

```
y = 0
for j = n/2 - 1 ... 0 do
    if (y + 2^j)^2 ≤ x then y = y + 2^j
return y
```

Memory Access

implementation notes

```
class Memory {
    static Array ram;
    static void init() {
        let ram = 0;
        :
    }
```

← init は初期化の private 関数

← 0 をベースアドレスにセットする

```
    let ram[addr] = val
```

→ ram[i] results

0 + i in the RAM

Heap Management

Area we put actual data is called Heap

init:

free = heapBase

alloc(size): // allocate memory block of size words

block = free

free = free + size

return block

deAlloc(object):

↑

というメソッドにするか？

Linked list で扱う (segment や node)

- Over head がある (2)
  - 次の pointer を保持する
  - size を格納する

alloc (size):

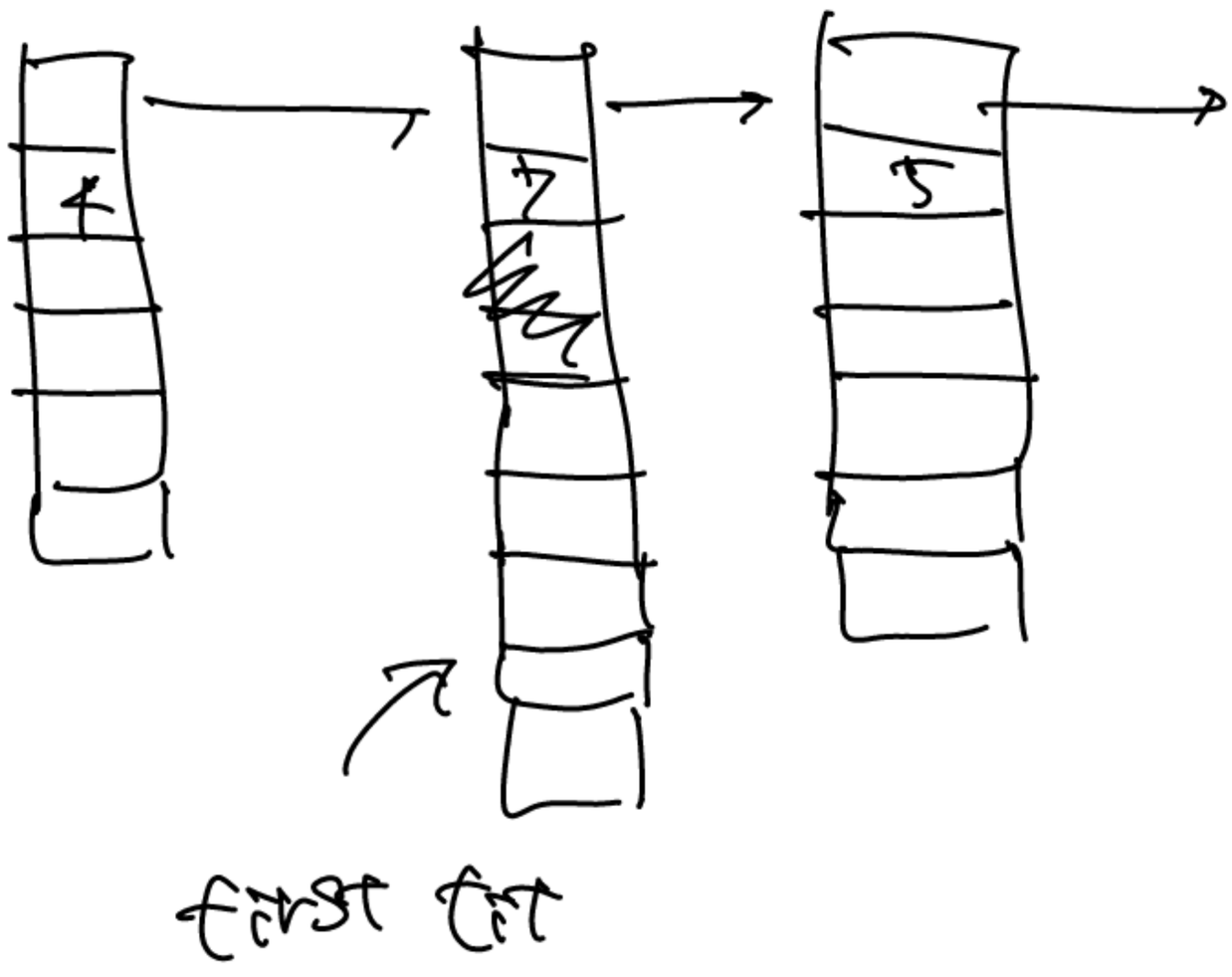
if segment size ≥ size + 2

we say segment is possible

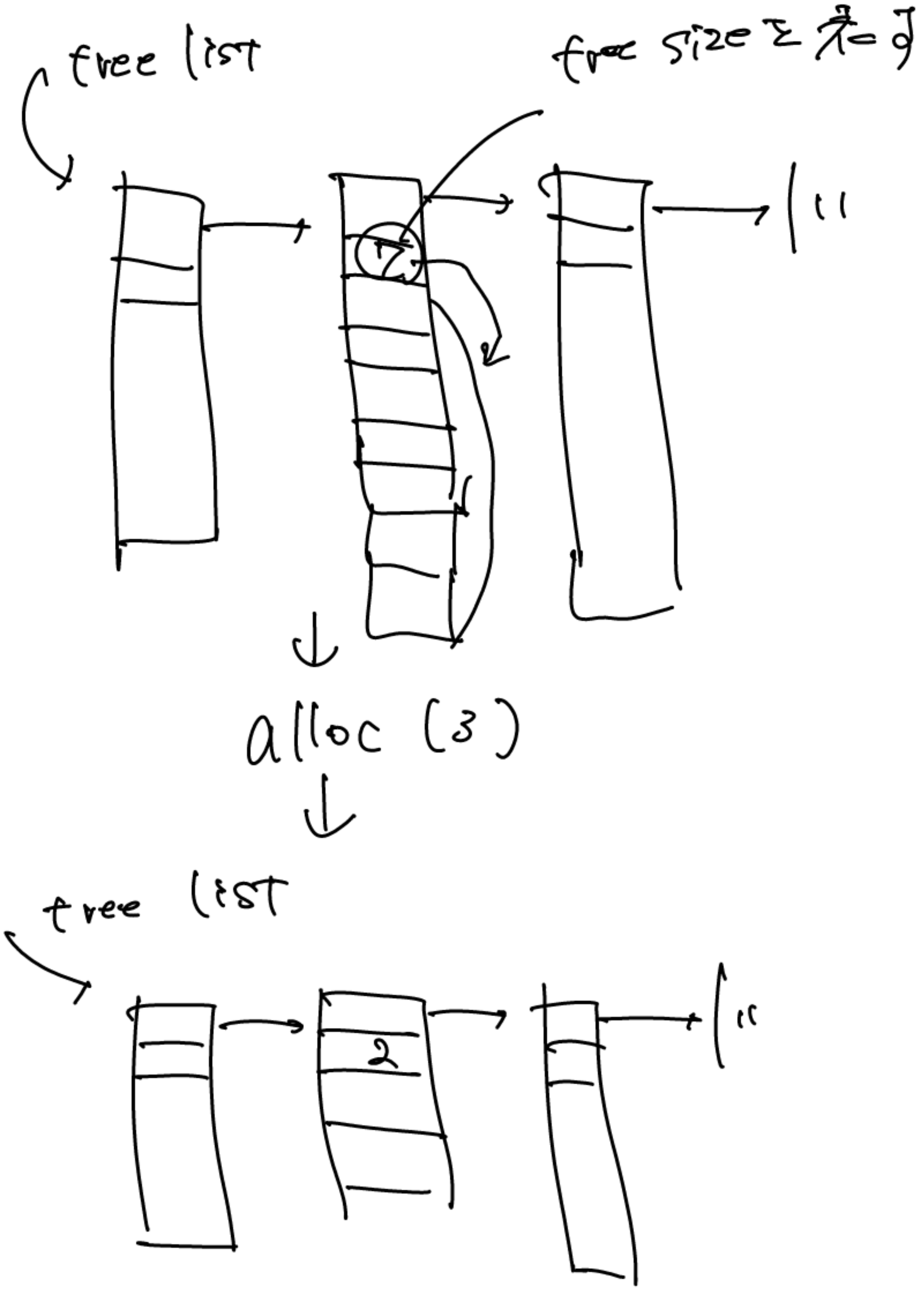
- search for free list (どの大きさを探すか)
  - 最初に見つけた穴 (first fit)
  - 一番小さい穴 (best fit)

- Carve a block of size size + 2

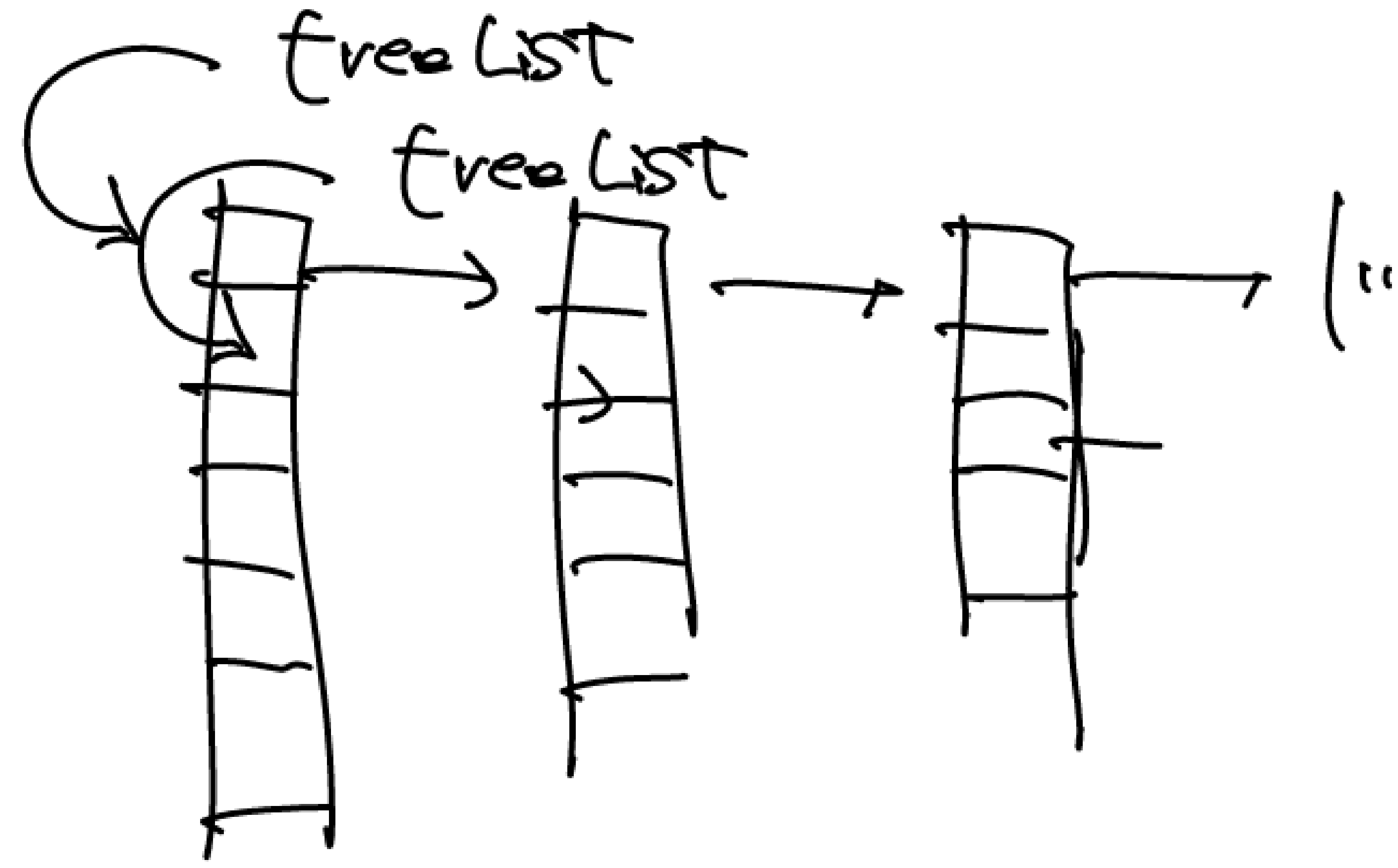
- return the bass address of the block.

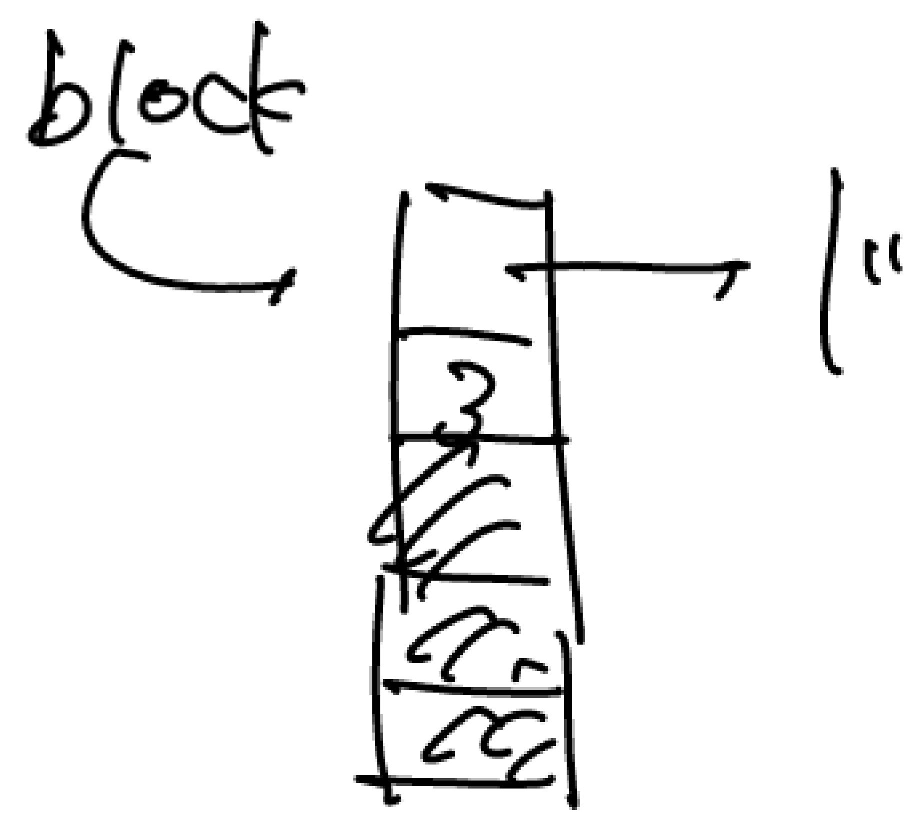


alloc :



deAlloc(obj)



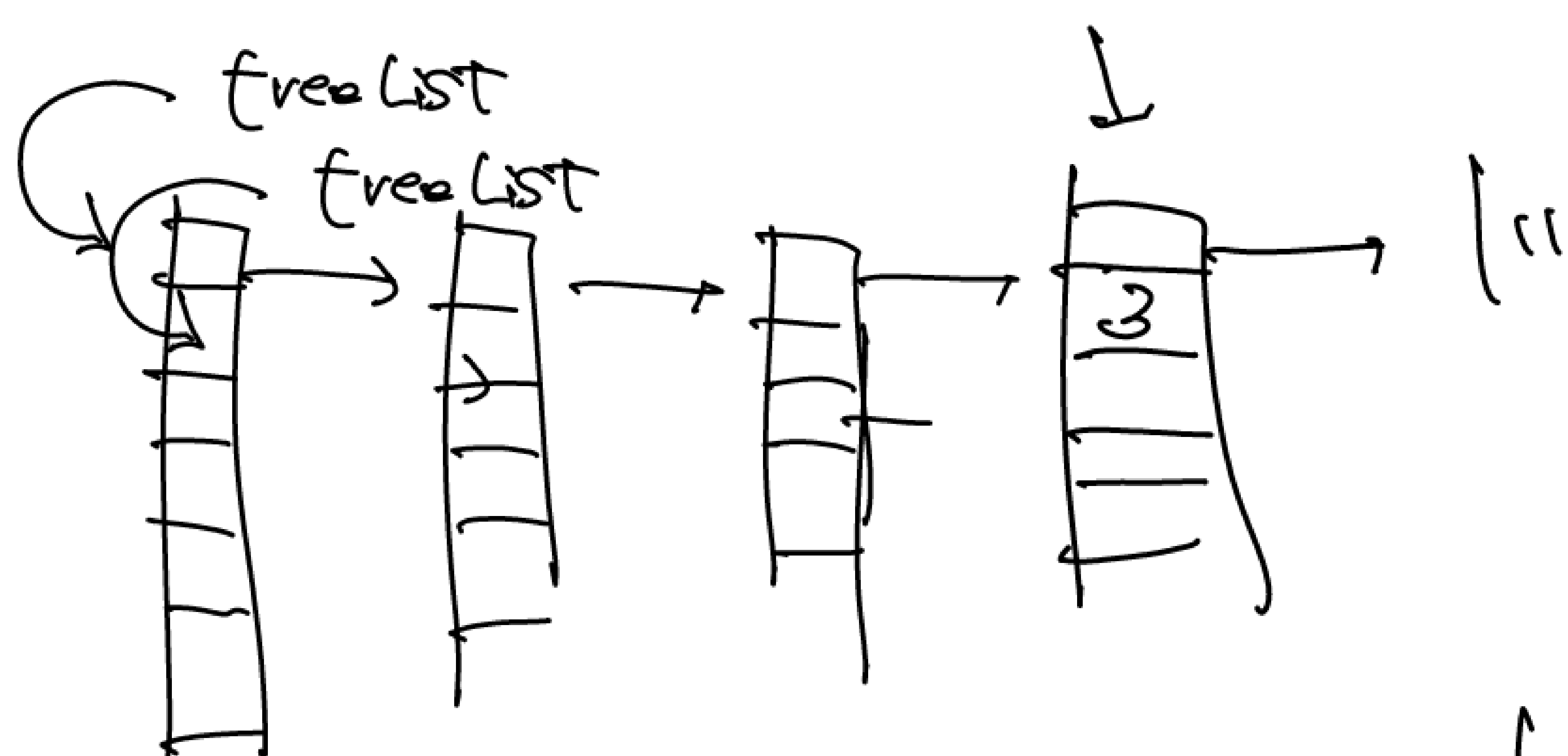


↓

deAlloc(block)

↓

末尾に追加する



The more we recycle, the more freeList becomes fragmented → deflag

Graphics

very important section

Screen Class

vector graphics

in file

drawLine(3, 0, 11, 0)

drawRectangle(3, 1, 9, 5)

⋮

instruction を持つ

bitmap graphics

in file

000 111 000 ...

00 1000 0 ...

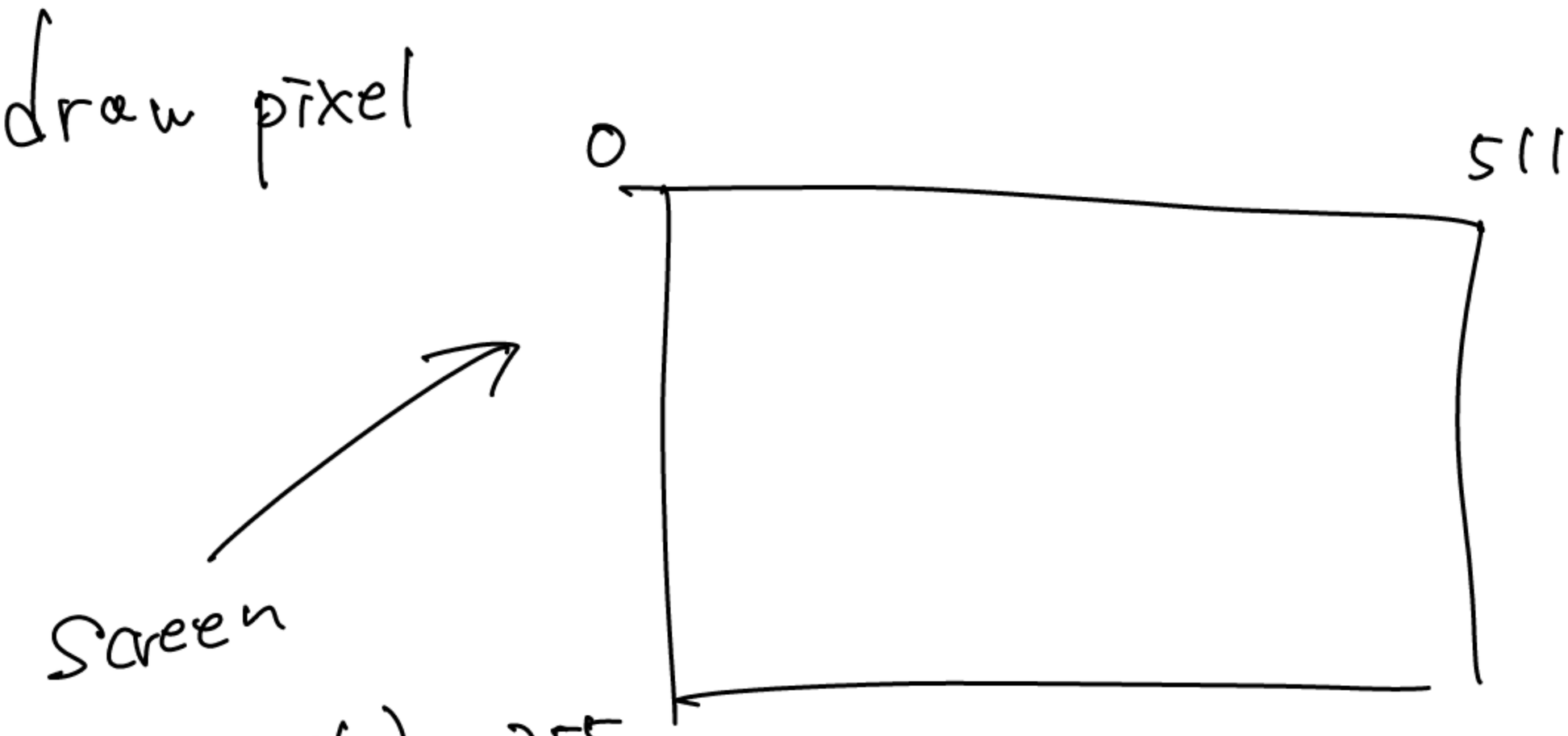
⋮

bit を持つ

↑

- scale する
- bitmap に直接描く

draw pixel



function drawPixel (int x, int y) {

address = 32 * y + x / 16

(512 ÷ 16) (256 ÷ 16)

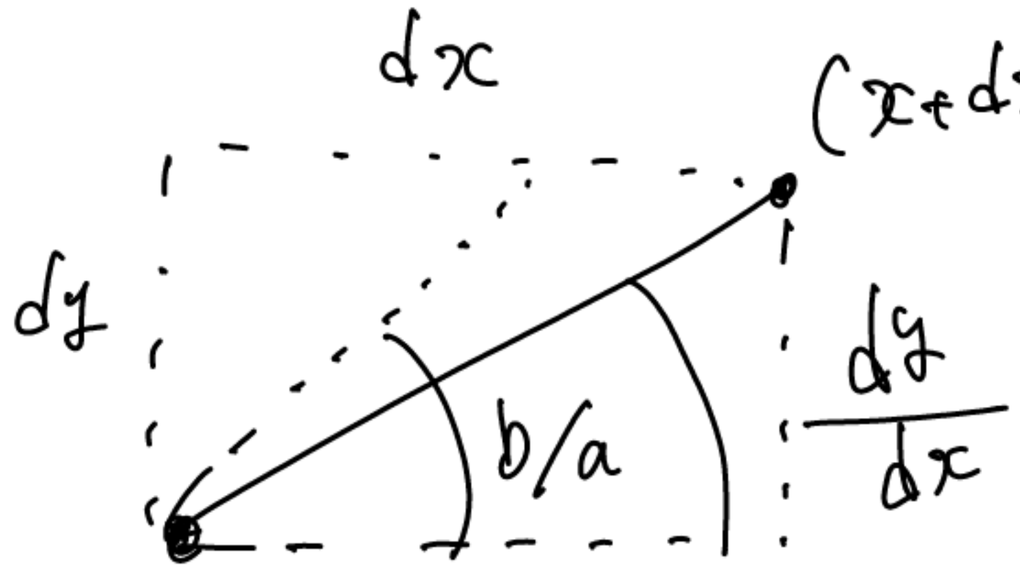
value = Memory.peek [base + addr]

set (x % 16) bit of value to color

do Memory.poke (addr, value)

Basic idea : image drawing is implemented through a sequence of drawLine operations

Challenge : draw lines fast



how many went to right

how many went to UP

```
a = 0; b = 0;
while ((a ≦ dx) and (b ≦ dy))
    drawpixel (x+a, y+b);
    //decide go right or up
    If go to right
        a = a+1
    else
        b = b+1
```

b/a , dy/dx を計算

b/a > dy/dx なら right に行く

Optimization

(b/a > dy/dx) → (b*dx > a*dy)

let diff = a*dy - b*dx

if a = a+1 → diff goes up by dy

if b = b+1 → diff goes down by dx

↓ 반전

a=0, b=0, diff = 0

while ((a≦dx) and (b≦dy))

drawPixel(x+a, y+b);

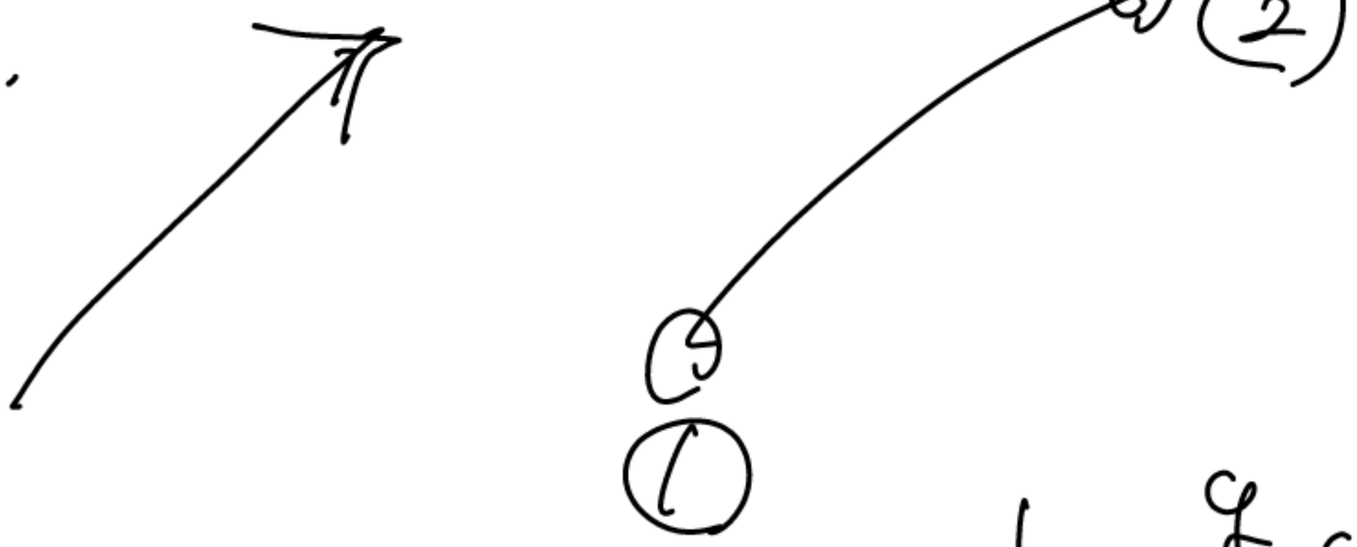
if (diff<0) { a=a+1; diff = diff+dy)

if (diff>0) { b=b+1;

diff = diff - dx }

only invoke add, sub operations.

($x_1 < x_2$ and $y_1 < y_2$)

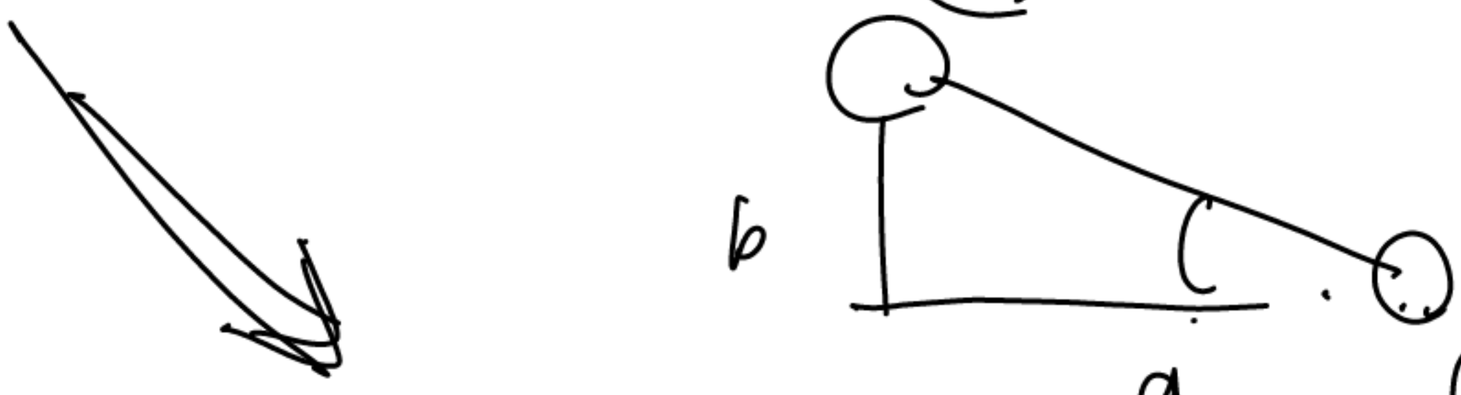


$dx = x_2 - x_1$

$dy = y_2 - y_1$

$a = 1$

($x_1 > x_2$ and $y_1 > y_2$)

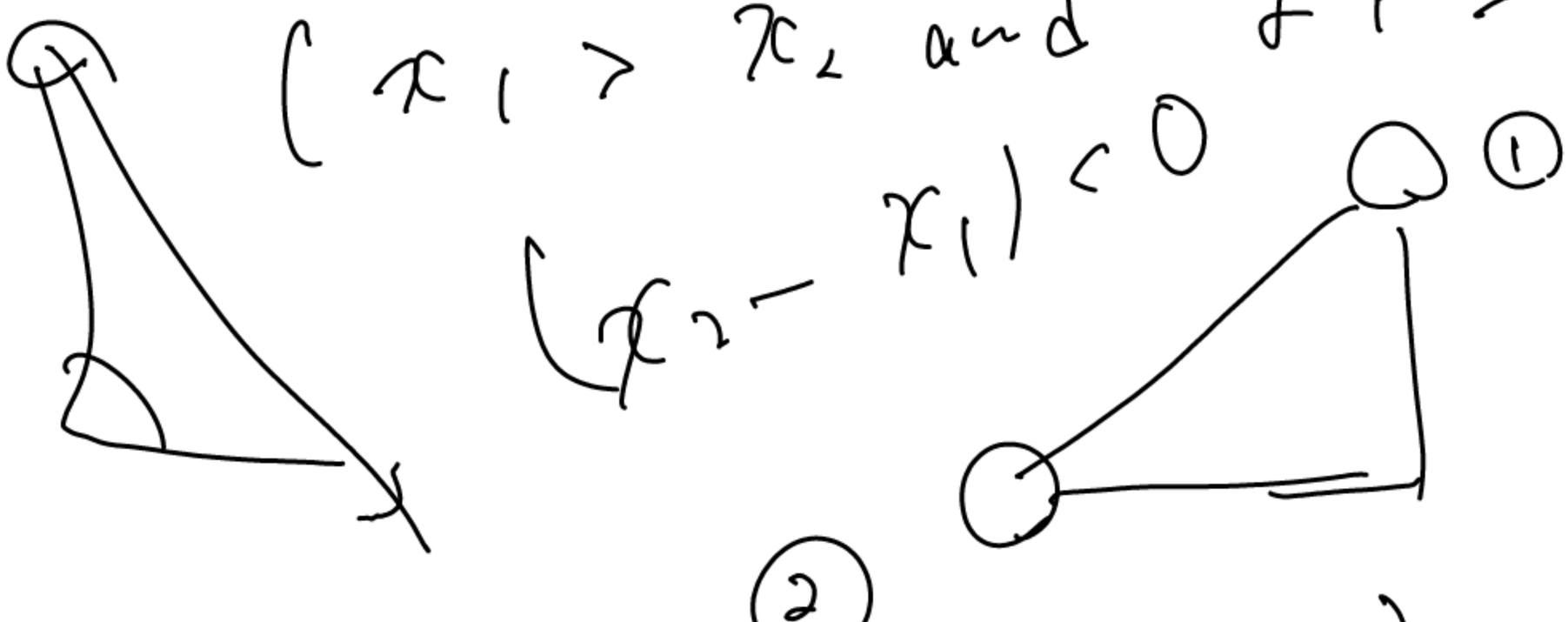


$dx = x_1 - x_2$

$dy = y_1 - y_2$

$a = -1$

($x_1 > x_2$ and $y_1 < y_2$)

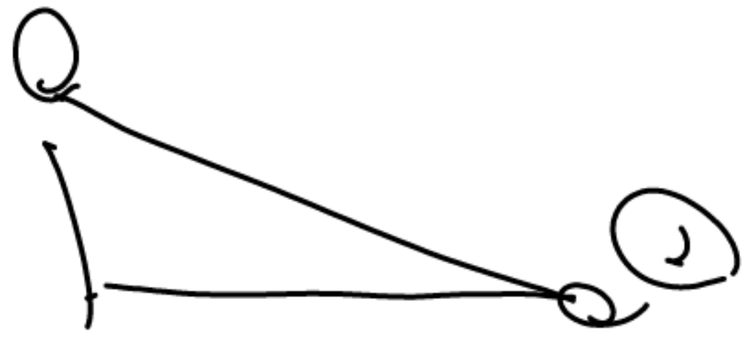


$dx = x_1 - x_2$

$dy = y_1 - y_2$

$a = -1$

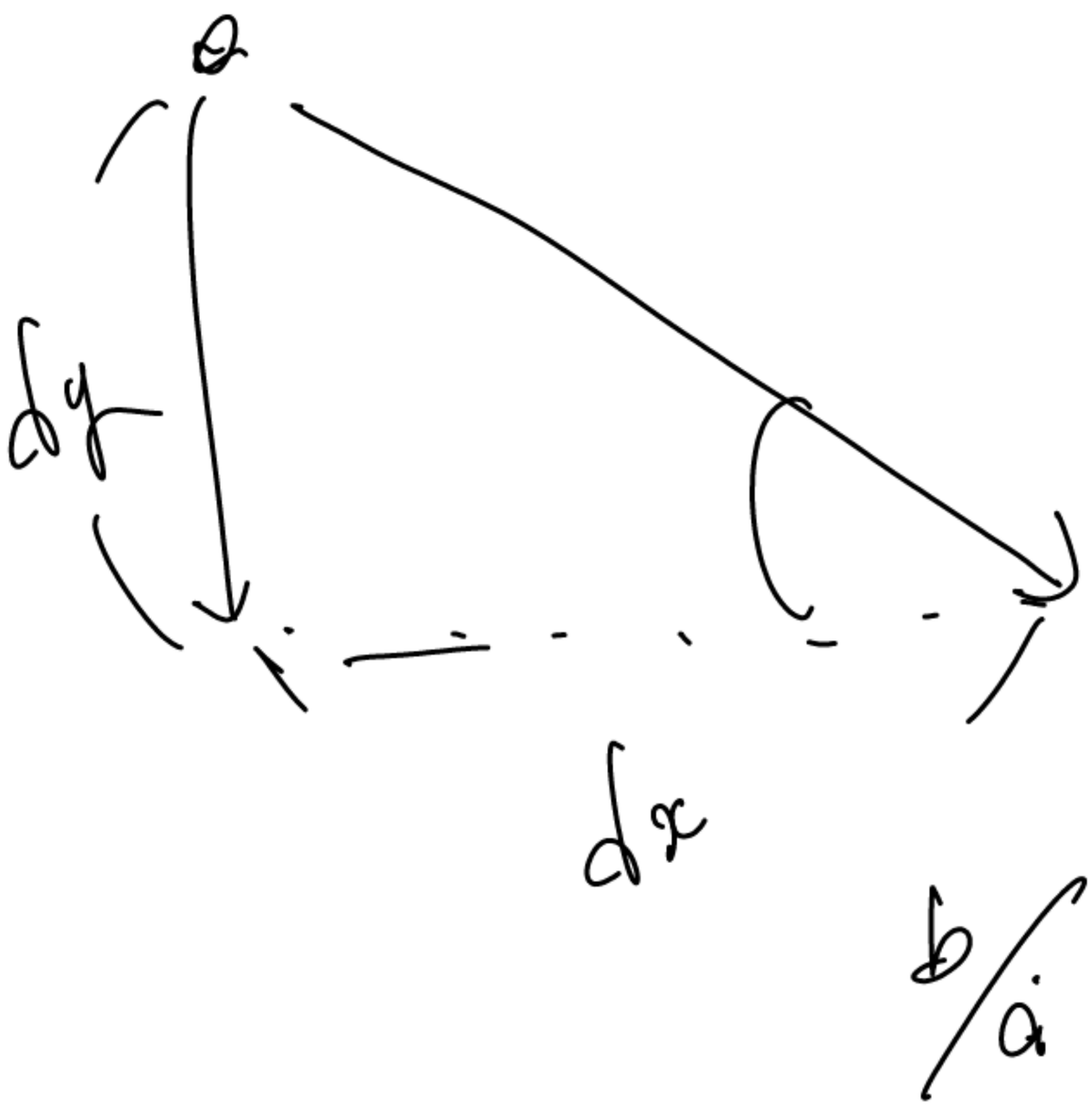
($x_1 < x_2$ and $y_1 > y_2$)

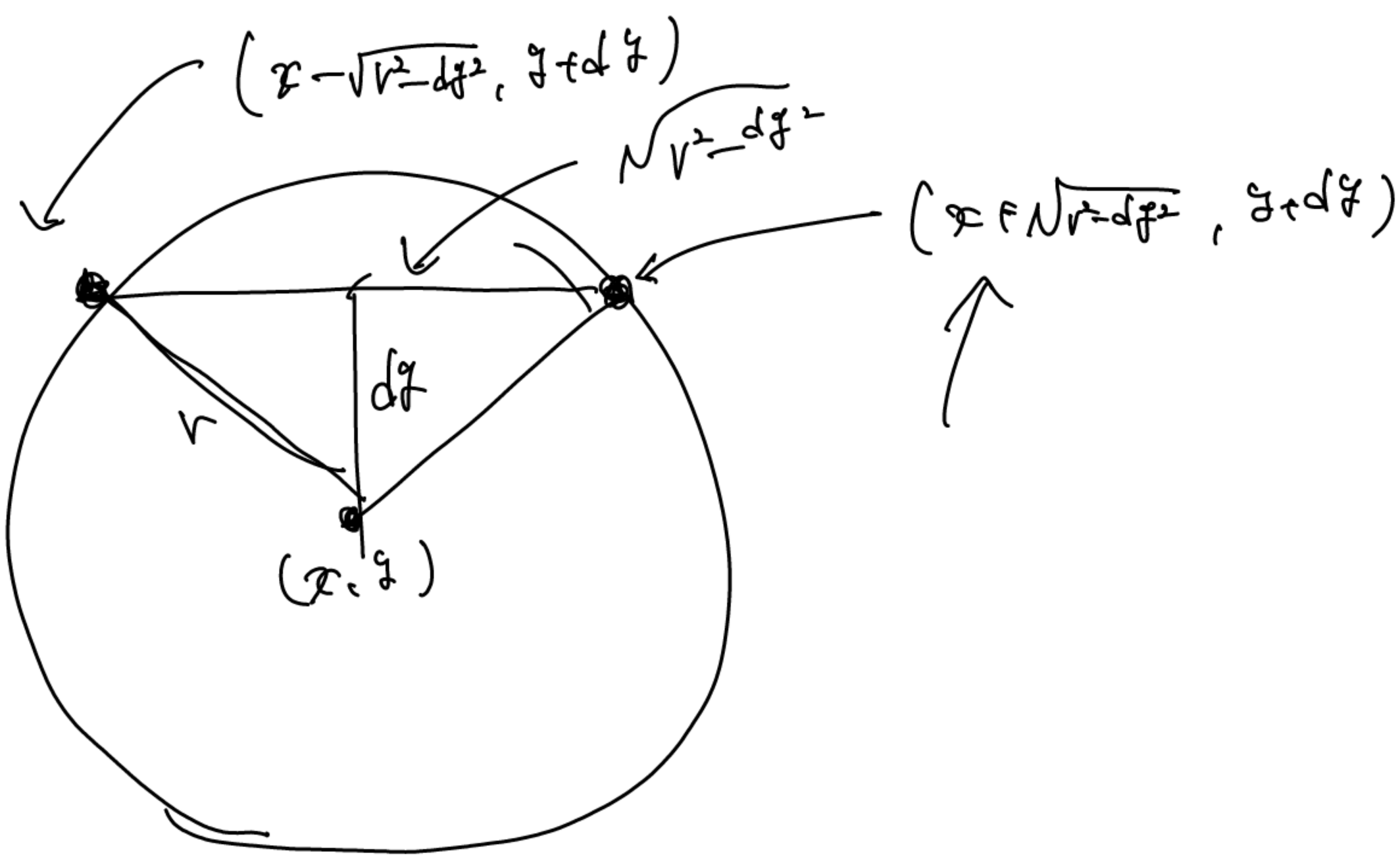


$dx = x_2 - x_1$

$dy = y_1 - y_2$

$a = 1$


dy
dx
b
a
dy/dx



draw Circle (x, y, r)

for each $dy = -r$ to r do:

draw Line $(x-\sqrt{r^2-dy^2}, y+dy,$

$x+\sqrt{r^2-dy^2}, y+dy)$

Textual Output

managed by Output class

Hack font

- 11px hight and 8 pixel wide
- 2 empty right columns.
  1 empty bottom row for character spacing

Input

handle keyboard

no key is pressed ⟶ keyboard register is 0

otherwise ⟶ set char code

keypressed : use Memory.peek

readChar :
readLine :
readInt :
) follow pseudo codes.

String Processing

int ⟷ string algorithms

```
int2string(int):
    lastD = val % 10;
    c = character for lastD
    if (val<10)
        return c
    else
        return int2string(val/10).append(c)
```

```
string2int(str):
    val = 0
    for(i = 0 .. str.length) do
        d = str(i)
        val = val*10+d
    return val
```

Array processing

function new (int size)

method void dispose

↑

Its function instead of constructor

↑

to abolde calling Memory.alloc

Sys Class

last class !!!

function init ()

Bootstrapping

Hardware Contract

Start with the instruction in Rom[0]

VM contract

sp = 256

call Sys.init

Jack contract

Main.main() is an entry point

OS Contract

Sys.init should initialize the OS,

call Main.main()

```
Class Sys {

    function void init() {
        do Math.init()
        do Memory.init()
        do Screen.init()
        :
        do Main.main()
}
```

initialize

Sys.halt ⟶ Loop

Sys.wait ⟶ machine specific using a Loop

Perspective

Jack OS lacks

- Multi threading
- File system
- Window System
- Security. Communication

User level. ユーザー permission などには入れない

str "123"

val = 0

for (i:0.. str.length - i) {

val = val * 10 + str[i]

return val

i	val
0	1
1	12
2	123

0101 0010

or (black) white

0111 0101

0111 0010 white.

↓ ?

0101 ×