コンパイラ

第8回 コード生成
― スタックマシン ―

http://www.info.kindai.ac.jp/compiler

E館3階E-331 内線5459

takasi-i@info.kindai.ac.jp

コンパイラの構造

- 字句解析系
- 構文解析系
- 制約検査系
- 中間コード生成系
- 最適化系
- 目的コード生成系

処理の流れ
情報システムプロジェクトIの場合

output (ab);

↓

字句解析系　　マイクロ構文の文法に従い解析

↓

“output” “(” 変数名 “)” “;”

↓

構文解析系　　マクロ構文の文法に従い解析

↓

<output_statement> ::= “output” “(” <exp> “)” “;”

↓

コード生成系　　VSMアセンブラの文法に従い生成

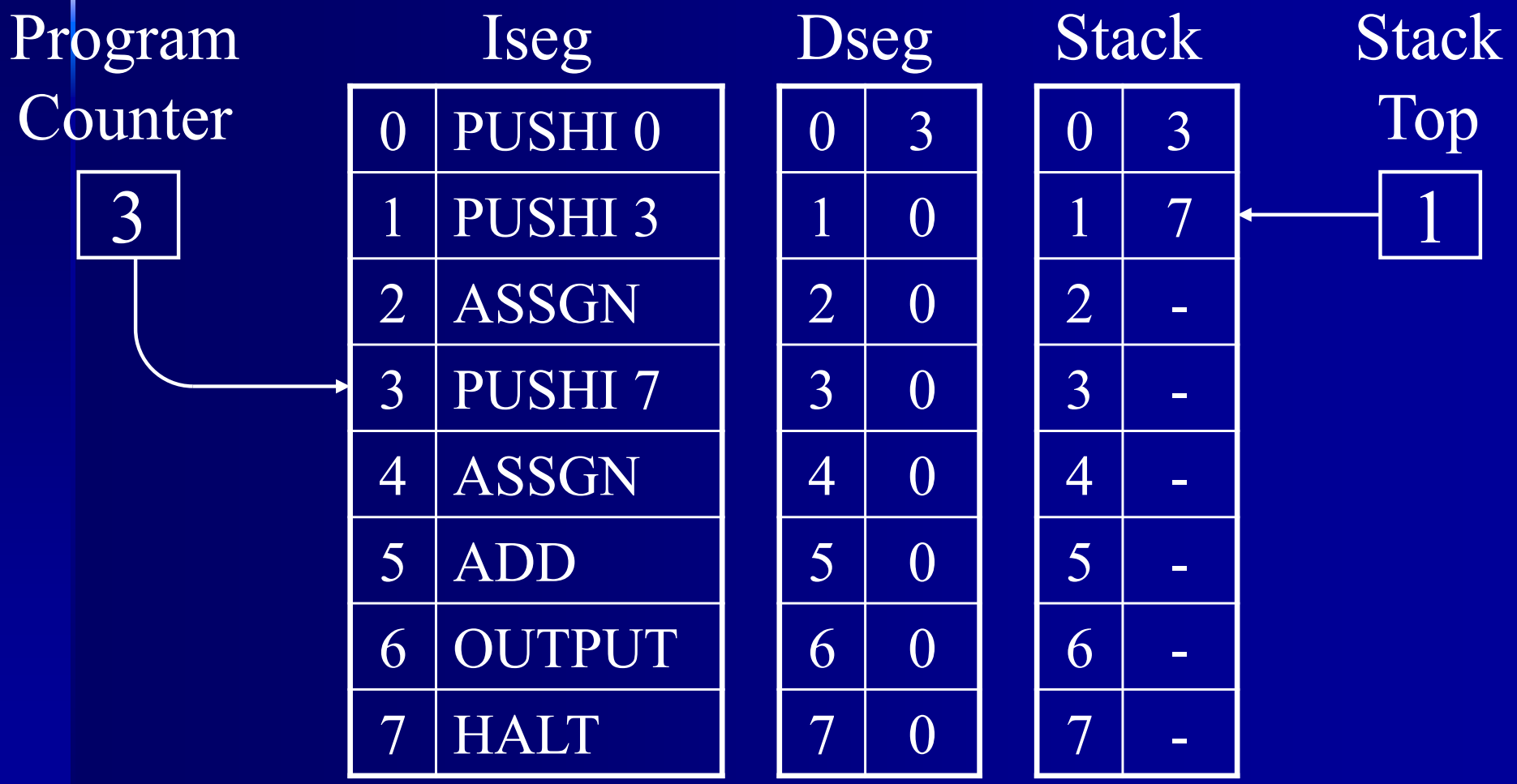
↓

1. PUSH &ab　　2. OUTPUT

スタックマシン (stack machine)

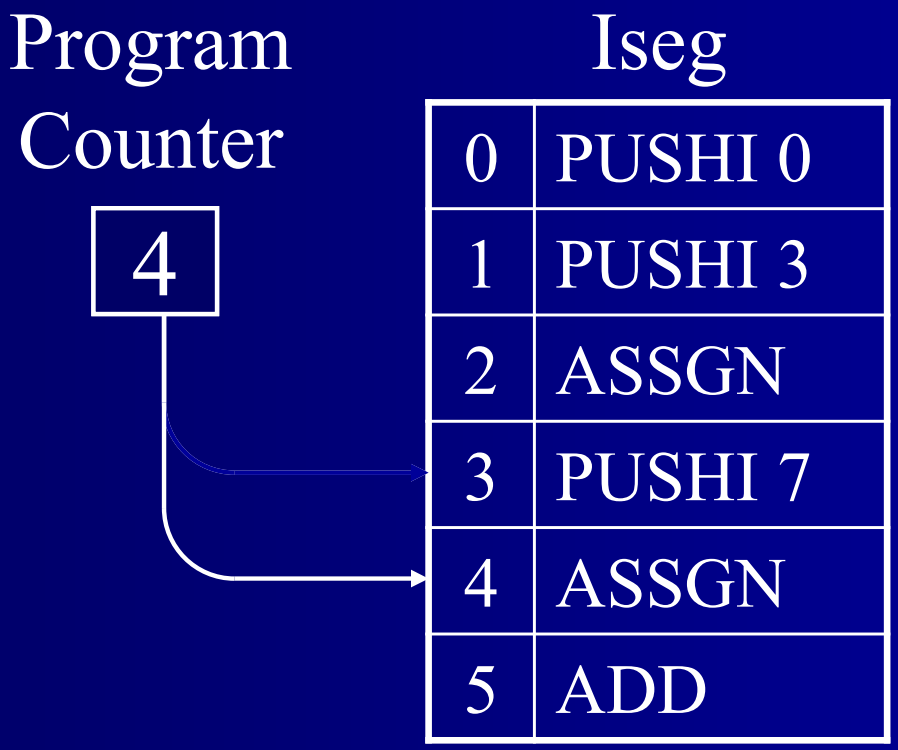
- スタックマシン
  - Iseg[] : アセンブラプログラムを格納
  - Dseg[] : 実行中の変数値を格納
  - Stack[] : スタック(作業場所)
  - Program Counter : 現在の Iseg の実行位置
  - Stack Top : 現在のスタックの操作位置

スタックマシン (stack machine)



Iseg と Program Counter

- VSM の動作
  1. Iseg の PC 番地の命令を実行
  2. PC := PC+1　or ジャンプ命令で指定した先



Dseg

- 実行中の変数値を格納

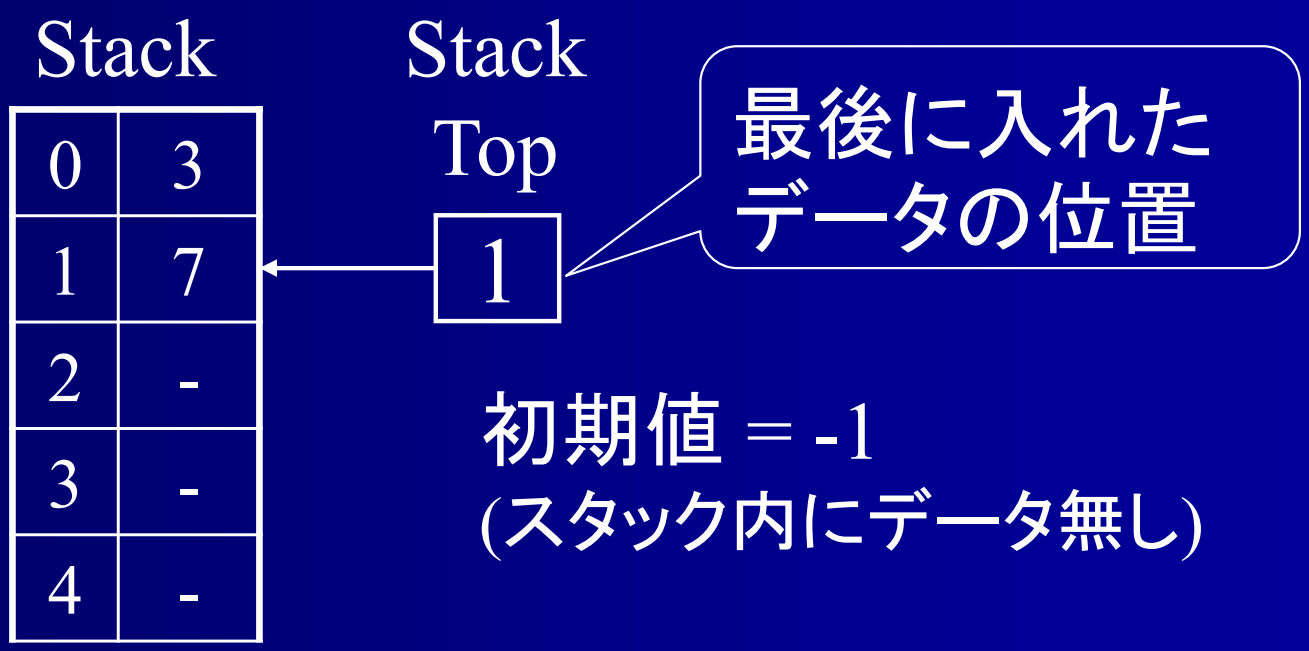
```
int i, j, x=2, y=3;
char c = ‘a’;
int a[5];
```

Dseg

0	-	i
1	-	j
2	2	x
3	3	y
4	‘a’	c
5	-	a[0]
6	-	a[1]
7	-	a[2]
8	-	a[3]
9	-	a[4]

Stack

- Stack
  - 作業場所，処理中のデータの一時置き場
  - Last In First Out



Stack, Dseg操作命令

命令	意味
PUSH *d*	Dseg の*d* 番地の値を積む
PUSHI *i*	*i* を積む
REMOVE	スタックトップを削除
POP *d*	Dseg の *d* 番地に値を書き込む
ASSGN	スタックトップの値を 2番目の値の番地に書き込む
LOAD	スタックトップの値の番地の値を積む
COPY	スタックトップの値をコピー
INC	スタックトップの値を1増やす
DEC	スタックトップの値を1減らす

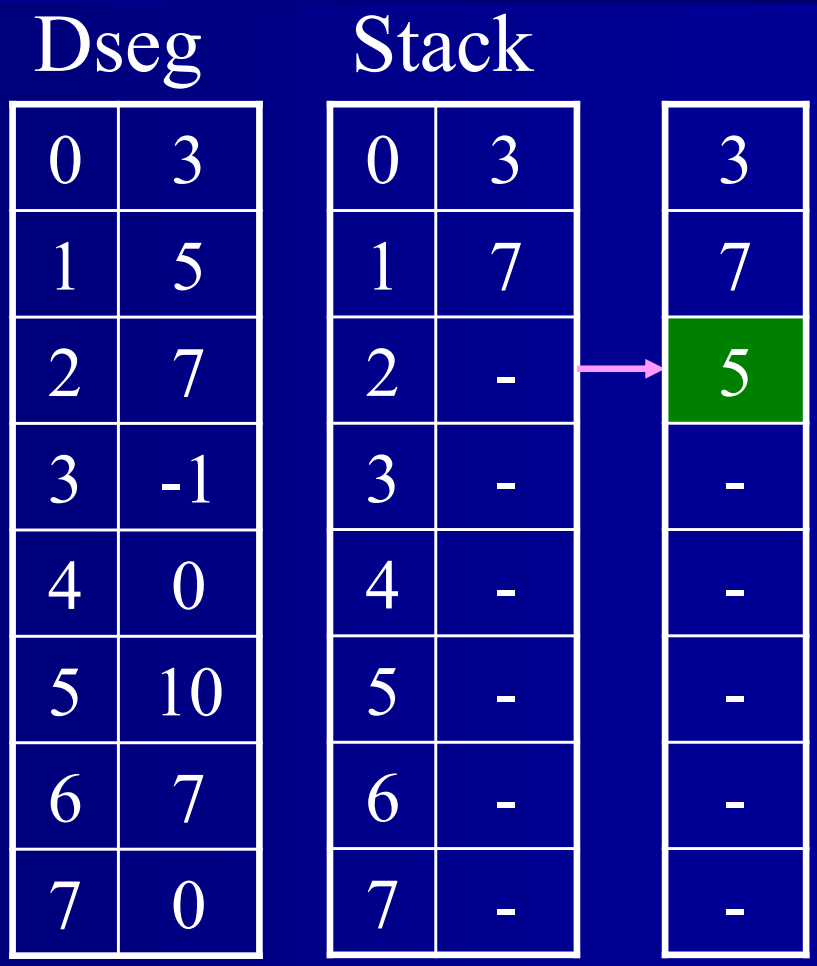
数値 → Stack
PUSHI 命令

整数値 *i* を積む

```
PUSHI i
```

例：整数 5 を積む

```
PUSHI 5
```



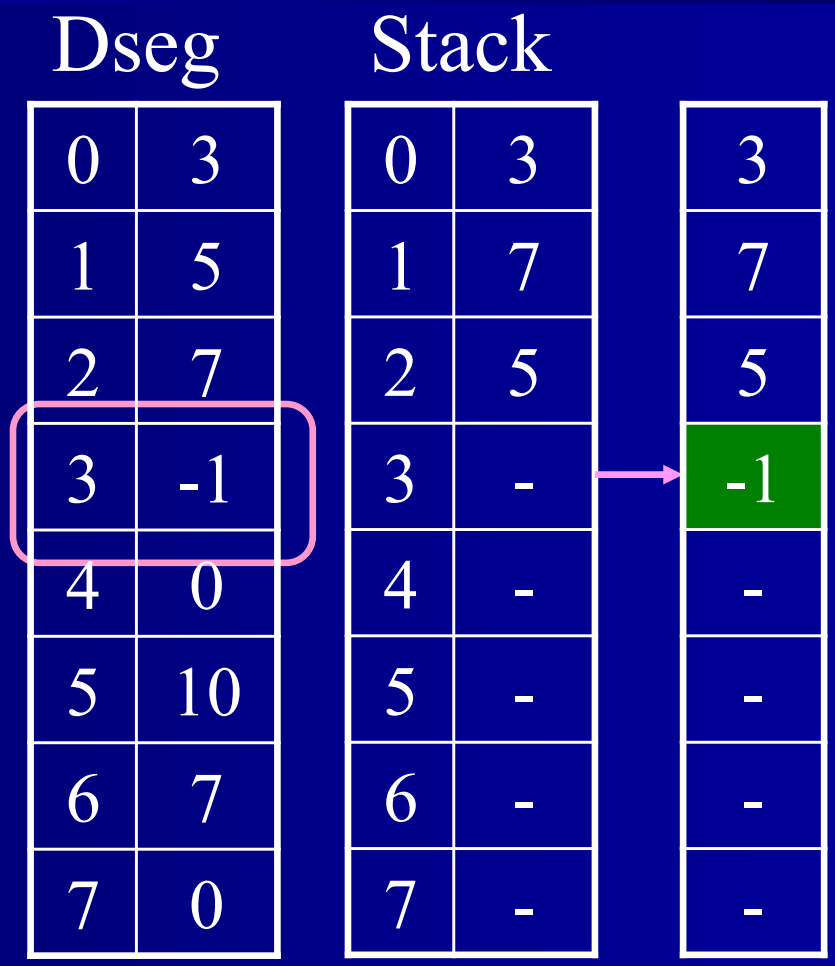
Dseg → Stack
PUSH 命令

Dsegの d 番地の
データを積む

PUSH d

例：3 番地のデータを積む

PUSH 3



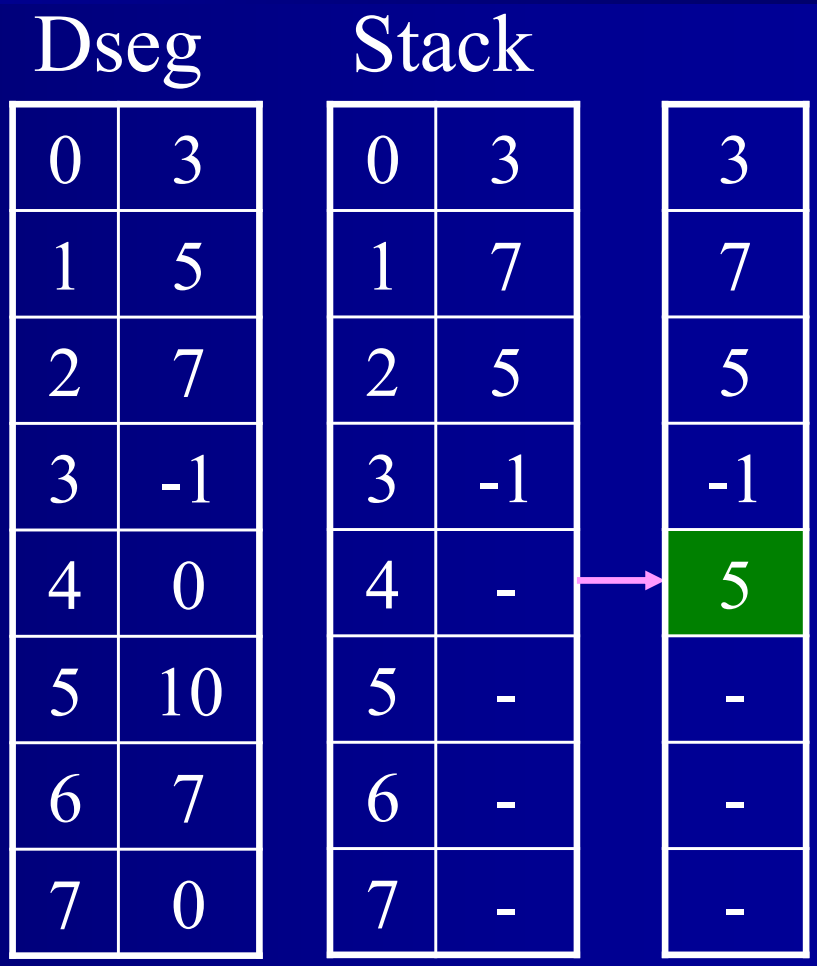
Dseg → Stack
LOAD 命令

スタックトップの番地のデータを積む

```
PUSHI d
LOAD
```

例：5 番地のデータを積む

```
PUSHI 5
LOAD
```



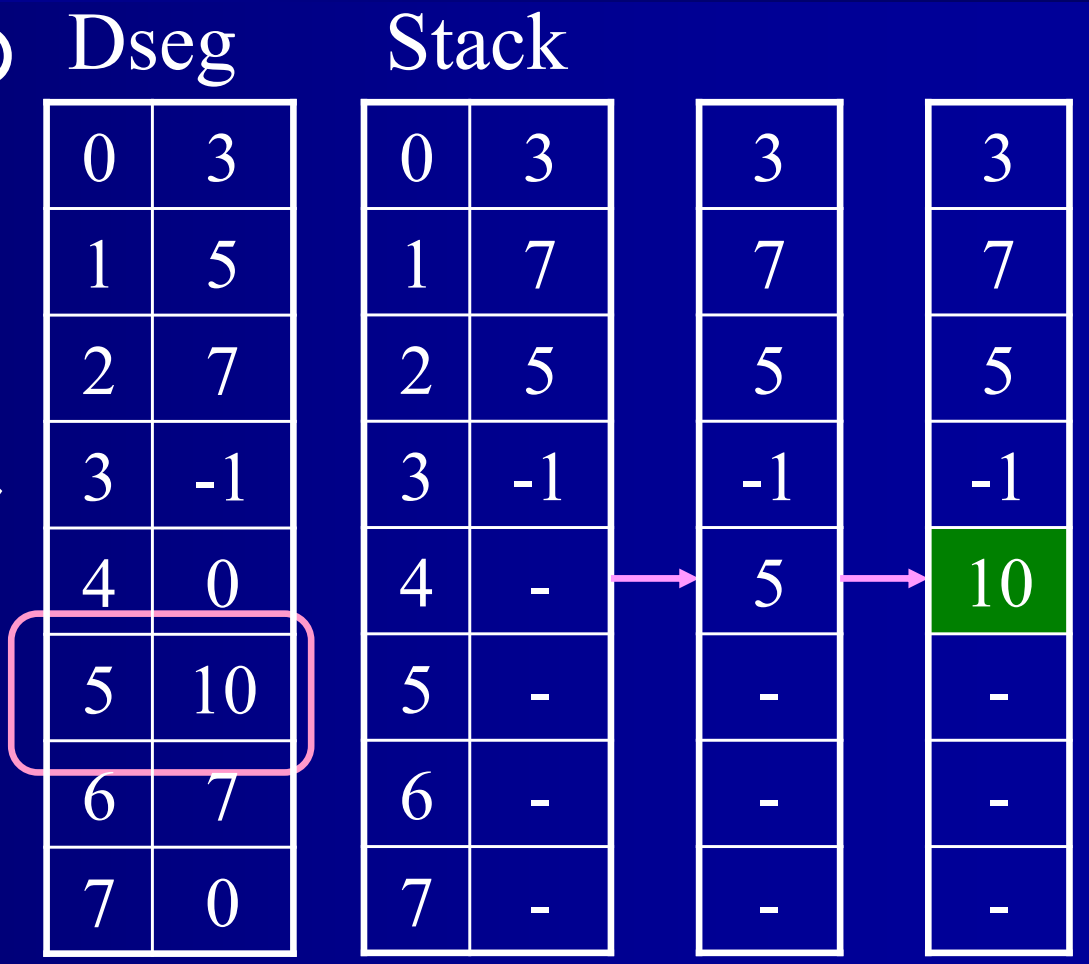
Dseg → Stack
LOAD 命令

スタックトップの番地のデータを積む

```
PUSHI d
LOAD
```

例：5 番地のデータを積む

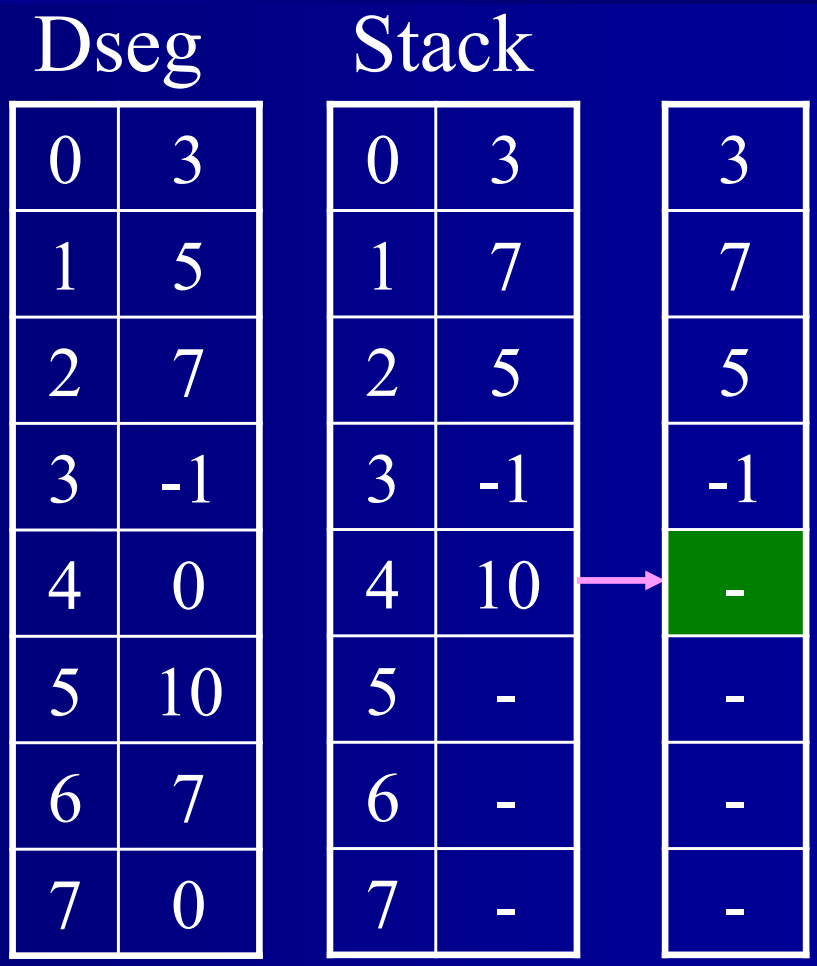
```
PUSHI 5
LOAD
```



Stack → 削除
REMOVE 命令

データを削除する

REMOVE



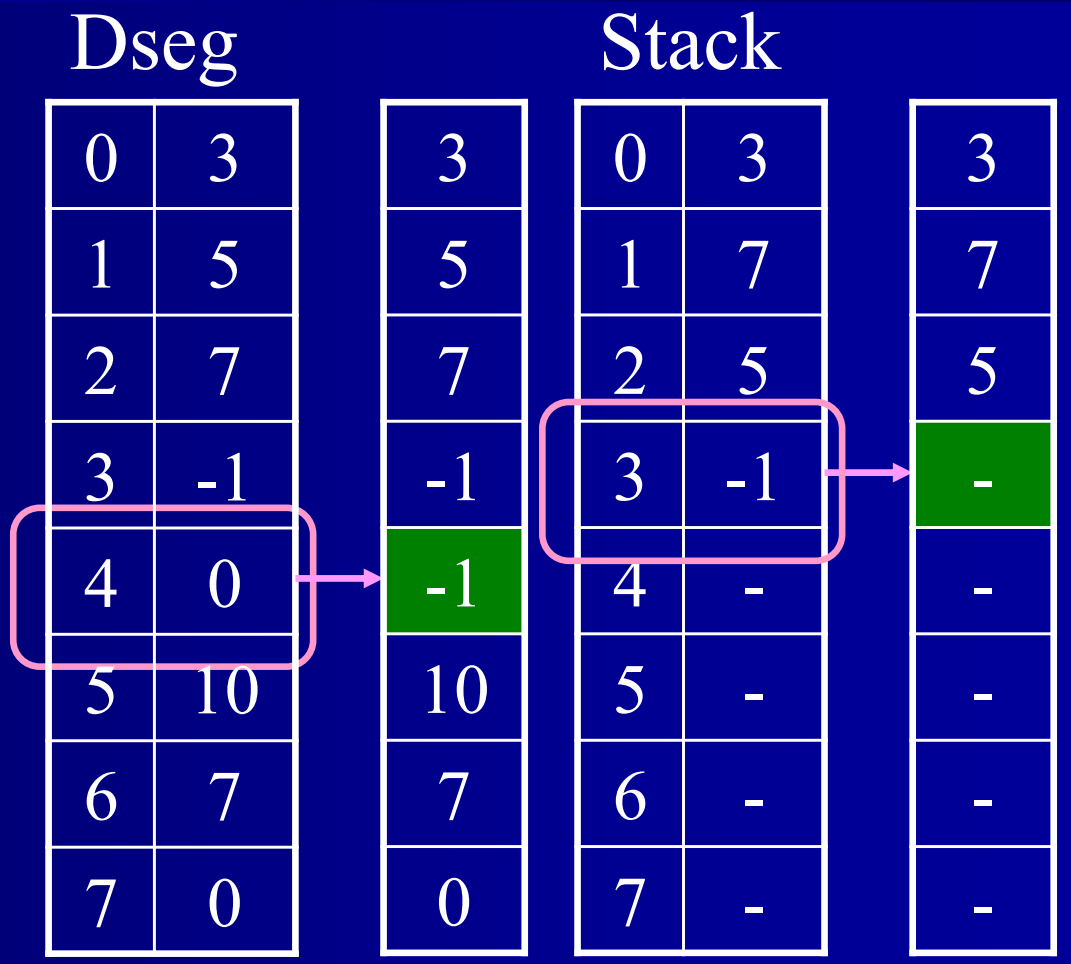
Stack → Dseg
POP 命令

Dsegの d 番地に
データを書き込む

POP d

例：4 番地にデータを書く

POP 4



Stack → Dseg
ASSGN 命令

スタックトップの値を
スタックの2番目の
番地に書き込む

```
PUSHI d
PUSHI x
ASSGN
```

例：7 番地にデータを書く

```
PUSHI 7
PUSHI 6
ASSGN
```

Dseg

0	3
1	5
2	7
3	-1
4	0
5	10
6	7
7	0

Stack

0	3
1	7
2	5
3	-
4	-
5	-
6	-
7	-

3
7
5
7
6
-
-
-

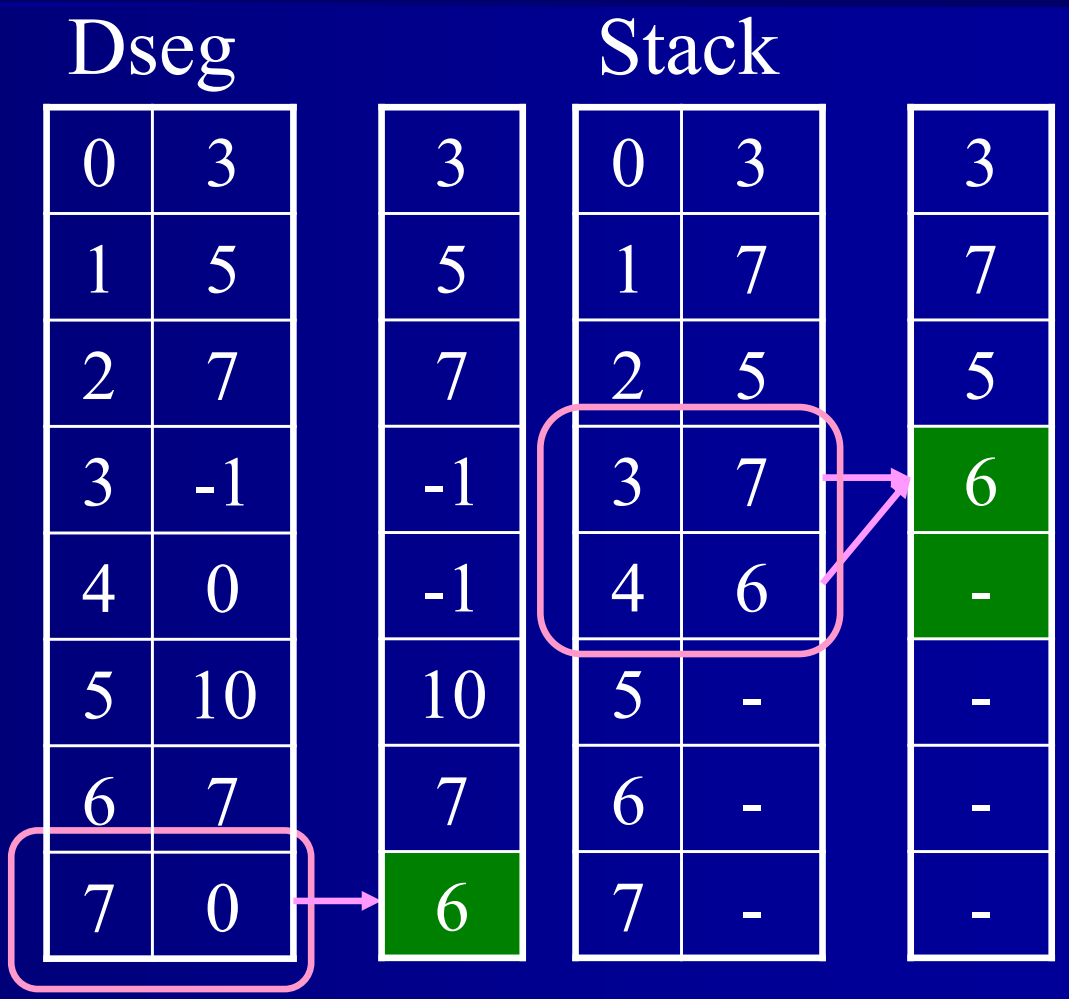
Stack → Dseg
ASSGN 命令

スタックトップの値を
スタックの2番目の
番地に書き込む

```
PUSHI d
PUSHI x
ASSGN
```

例：7 番地にデータを書く

```
PUSHI 7
PUSHI 6
ASSGN
```



Dseg の読み書き

- 実行中の変数値を格納
  - *d* 番地のデータをスタックに積む

```
PUSH d
```

```
PUSHI d
LOAD
```

  - スタックのデータを *d* 番地に書き込む

```
データをスタックに積む
POP d
```

```
PUSHI d
データをスタックに積む
ASSGN
REMOVE
```

コンパイル時に
番地が必要

Dseg からの読み込み
PUSH と PUSHI+LOAD

```
output ( x );
```

コンパイル時に
番地が分かる

⇩

PUSH 命令を使用

```
PUSH 2
OUTPUT
```

x の番地

Dseg

0	2	i
1	5	j
2	1	x
3	3	y
4	‘a’	c
5	3	a[0]
6	6	a[1]
7	9	a[2]
8	12	a[3]
9	15	a[4]

Stack

0	1
1	-
2	-
3	-
4	-
5	-
6	-
7	-

Dseg からの読み込み
PUSH と PUSHI+LOAD

```
output ( a[i] );
```

a[i] の番地は？

コンパイル時には
番地が分からない

```
PUSHI  5
PUSH   0
ADD
LOAD
OUTPUT
```

a[0] の番地

i の番地

Dseg

0	2	i
1	5	j
2	1	x
3	3	y
4	‘a’	c
5	3	a[0]
6	6	a[1]
7	9	a[2]
8	12	a[3]
9	15	a[4]

Stack

0	5
1	2
2	-
3	-
4	-
5	-
6	-
7	-

Dseg からの読み込み
PUSH と PUSHI+LOAD

```
output ( a[i] );
```

a[i] の番地は？

コンパイル時には
番地が分からない

```
PUSHI  5
PUSH   0
ADD
LOAD
OUTPUT
```

a[0] の番地

i の番地

Dseg

0	2	i
1	5	j
2	1	x
3	3	y
4	‘a’	c
5	3	a[0]
6	6	a[1]
7	9	a[2]
8	12	a[3]
9	15	a[4]

Stack

0	7
1	-
2	-
3	-
4	-
5	-
6	-
7	-

Dseg からの読み込み
PUSH と PUSHI+LOAD

```
output ( a[i] );
```

a[i] の番地は？

コンパイル時には
番地が分からない

```
PUSHI  5
PUSH   0
ADD
LOAD
OUTPUT
```

a[0] の番地

i の番地

Dseg

0	2	i
1	5	j
2	1	x
3	3	y
4	‘a’	c
5	3	a[0]
6	6	a[1]
7	9	a[2]
8	12	a[3]
9	15	a[4]

Stack

0	9
1	-
2	-
3	-
4	-
5	-
6	-
7	-

Dseg への書き込み
POP と PUSHI+ASSGN

```
int y = 5;
```

コンパイル時に
番地および
書き込む値が分かる

⇩

POP 命令を使用

➡
```
PUSHI 5
POP    3
```

Dseg

0	4	i
1	5	j
2	1	x
3	-	y
4	-	-
5	-	-
6	-	-
7	-	-
8	-	-
9	-	-

Stack

0	5
1	-
2	-
3	-
4	-
5	-
6	-
7	-

Dseg への書き込み
POP と PUSHI+ASSGN

```
int y = 5;
```

コンパイル時に
番地および
書き込む値が分かる

⇩

POP 命令を使用

```
PUSHI 5
POP    3
```

y の番地

Dseg

0	4	i
1	5	j
2	1	x
3	5	y
4	-	-
5	-	-
6	-	-
7	-	-
8	-	-
9	-	-

Stack

0	-
1	-
2	-
3	-
4	-
5	-
6	-
7	-

Dseg への書き込み
POP と PUSHI+ASSGN

```
a[i] = 10;
```

a[i] の番地は
コンパイル時には
分からない

```
PUSHI 4
PUSH  0
ADD
PUSHI 10
ASSGN
REMOVE
```

a[0] の番地

i の番地

Dseg

0	4	i
1	5	j
2	1	x
3	5	y
4	-	a[0]
5	-	a[1]
6	-	a[2]
7	-	a[3]
8	-	a[4]
9	-	-

Stack

0	4
1	4
2	-
3	-
4	-
5	-
6	-
7	-

Dseg への書き込み
POP と PUSHI+ASSGN

```
a[i] = 10;
```

a[i] の番地は
コンパイル時には
分からない

```
PUSHI 4
PUSH  0
ADD
PUSHI 10
ASSGN
REMOVE
```

a[0] の番地

i の番地

Dseg

0	4	i
1	5	j
2	1	x
3	5	y
4	-	a[0]
5	-	a[1]
6	-	a[2]
7	-	a[3]
8	-	a[4]
9	-	-

Stack

0	8
1	-
2	-
3	-
4	-
5	-
6	-
7	-

Dseg への書き込み
POP と PUSHI+ASSGN

```
a[i] = 10;
```

a[i] の番地は
コンパイル時には
分からない

```
PUSHI 4
PUSH  0
ADD
PUSHI 10
ASSGN
REMOVE
```

a[0] の番地

i の番地

Dseg

0	4	i
1	5	j
2	1	x
3	5	y
4	-	a[0]
5	-	a[1]
6	-	a[2]
7	-	a[3]
8	-	a[4]
9	-	-

Stack

0	8
1	10
2	-
3	-
4	-
5	-
6	-
7	-

Dseg への書き込み
POP と PUSHI+ASSGN

```
a[i] = 10;
```

a[i] の番地は
コンパイル時には
分からない

```
PUSHI 4
PUSH  0
ADD
PUSHI 10
ASSGN
REMOVE
```

a[0] の番地

i の番地

代入値が
残る

Dseg

0	4	i
1	5	j
2	1	x
3	5	y
4	-	a[0]
5	-	a[1]
6	-	a[2]
7	-	a[3]
8	10	a[4]
9	-	-

Stack

0	10
1	-
2	-
3	-
4	-
5	-
6	-
7	-

Dseg への書き込み
POP と PUSHI+ASSGN

```
a[i] = 10;
```

a[i] の番地は
コンパイル時には
分からない

```
PUSHI 4
PUSH  0
ADD
PUSHI 10
ASSGN
REMOVE
```

PUSHI 4 → a[0] の番地

PUSH 0 → i の番地

代入値が残る

Dseg

0	4	i
1	5	j
2	1	x
3	5	y
4	-	a[0]
5	-	a[1]
6	-	a[2]
7	-	a[3]
8	10	a[4]
9	-	-

Stack

0	-
1	-
2	-
3	-
4	-
5	-
6	-
7	-

Dseg の読み書き

- *d* 番地のデータをスタックに積む

スカラー変数の参照	配列の参照
PUSH *d*	PUSHI *d* LOAD

- スタックのデータを *d* 番地に書き込む

変数の初期値代入	それ以外
データをスタックに積む POP *d*	PUSHI *d* データをスタックに積む ASSGN REMOVE

入出力命令

命令	意味
INPUT	キーボードから整数値を読む
INPUTC	キーボードから文字を読む
OUTPUT	画面に整数値を書く
OUTPUTC	画面に文字を書く
OUTPUTLN	画面に改行を書く

入出力命令

整数値の読み込み


```
i = inputint;
```

```
PUSHI  i のアドレス
INPUT
ASSGN
REMOVE
```

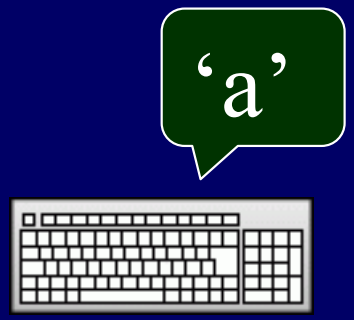
Stack

0	3
1	1
2	-
3	-
4	-
5	-
6	-
7	-

3
1
15
-
-
-
-
-

入出力命令

文字の読み込み




```
c = inputchar;
```

```
PUSHI  c のアドレス
INPUTC
ASSGN
REMOVE
```

Stack

0	3
1	1
2	-
3	-
4	-
5	-
6	-
7	-

3
1
97
-
-
-
-
-

入出力命令

整数値の書き出し

```
outputint (12);
```


```
PUSHI  12
OUTPUT
OUTPUTLN
```

Stack

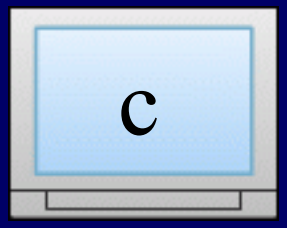
0	3
1	1
2	12
3	-
4	-
5	-
6	-
7	-

3
1
-
-
-
-
-
-

入出力命令

文字の書き出し

```
outputchar (‘c’);
```




```
PUSHI  99
OUTPUTC
OUTPUTLN
```

Stack

0	3
1	1
2	99
3	-
4	-
5	-
6	-
7	-

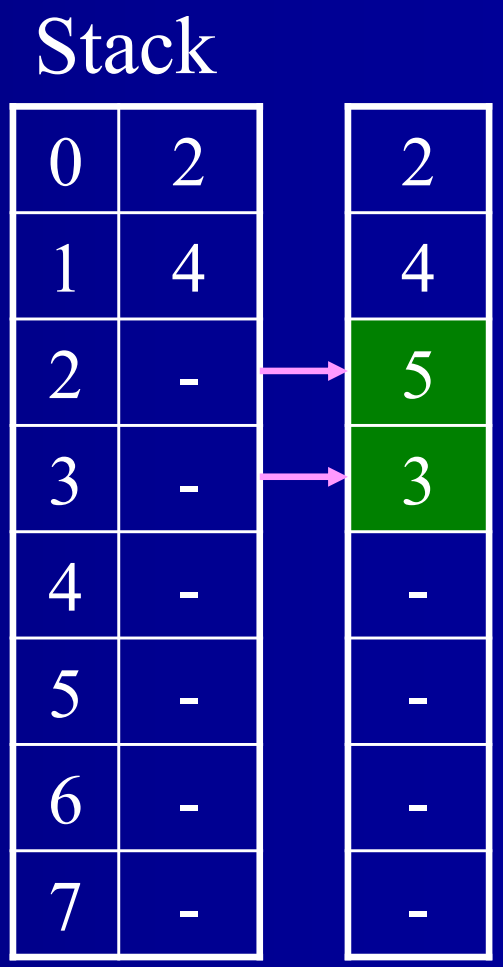
3
1
-
-
-
-
-
-

演算命令

- 演算はスタック上で行う
  1. スタックにデータを積む
  2. 演算

```
5 + 3
```

```
PUSHI 5
PUSHI 3
ADD
```

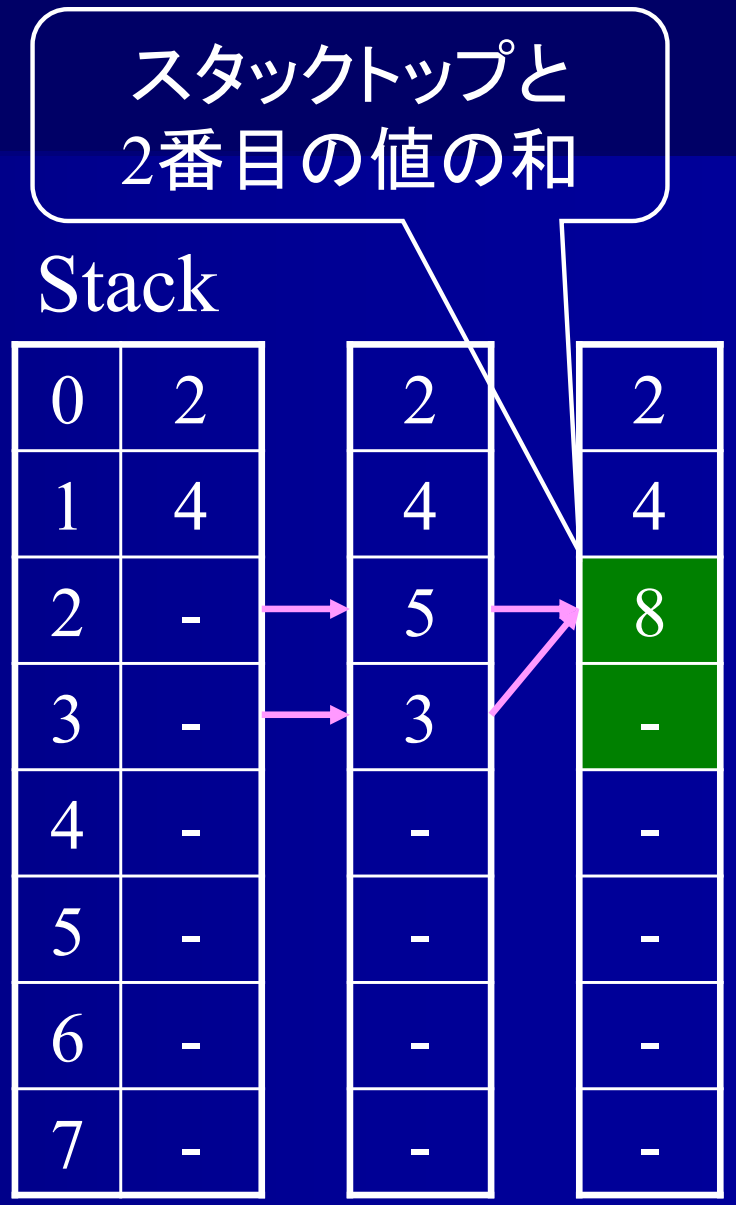


演算命令

- 演算はスタック上で行う
  1. スタックにデータを積む
  2. 演算

```
5 + 3
```

```
PUSHI 5
PUSHI 3
ADD
```



逆ポーランド記法，後置記法

- 逆ポーランド記法
  - 演算子を最後に置く

中置記法

a＋b＊c

逆ポーランド記法(後置記法)

a b c ＊ ＋

ポーランド記法(前置記法)

＋a＊b c

逆ポーランド記法の利点

- 逆ポーランド記法の利点
  - 括弧が不要
  - 演算子の優先順位を考慮しなくていい

演算子を読み込む
⇒演算子の前2つの値の演算を行う

スタックマシンに向いている

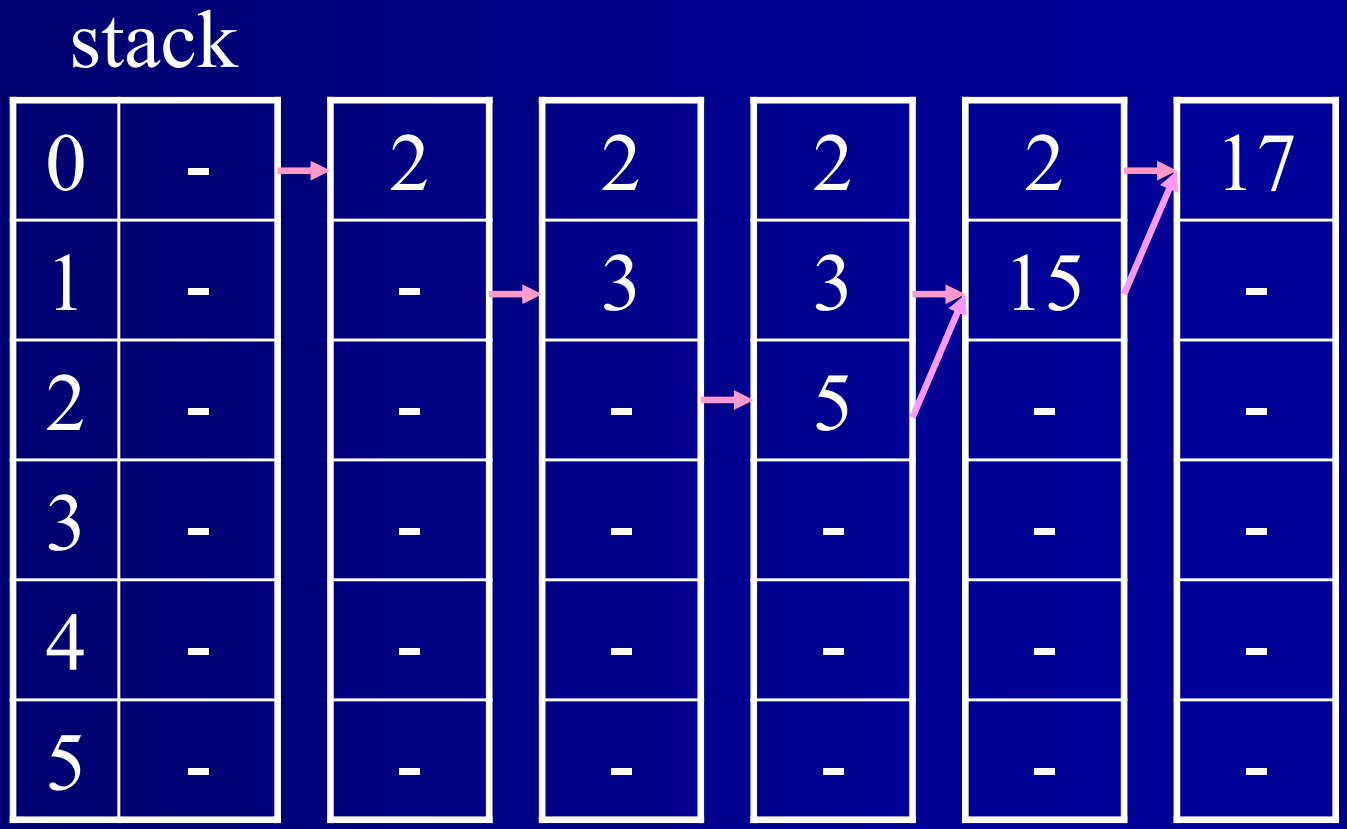
演算のアセンブラコード

例：2 + 3 * 5

⇩ 逆ポーランド記法

2 3 5 * +

```
PUSHI  2
PUSHI  3
PUSHI  5
MUL
ADD
```



演算のアセンブラコード

- 演算のアセンブラコード
  - 演算子に対応したコードを最後に置く

例 : <Exp> ::= <Term>$_1$ “+” <Term>$_2$
<Term> ::= <Factor>$_1$ “*” <Factor>$_2$

<Exp>

<Term>$_1$ のコード(右辺値)
<Term>$_2$ のコード(右辺値)
ADD

<Term>

<Factor>$_1$ のコード(右辺値)
<Factor>$_2$ のコード(右辺値)
MUL

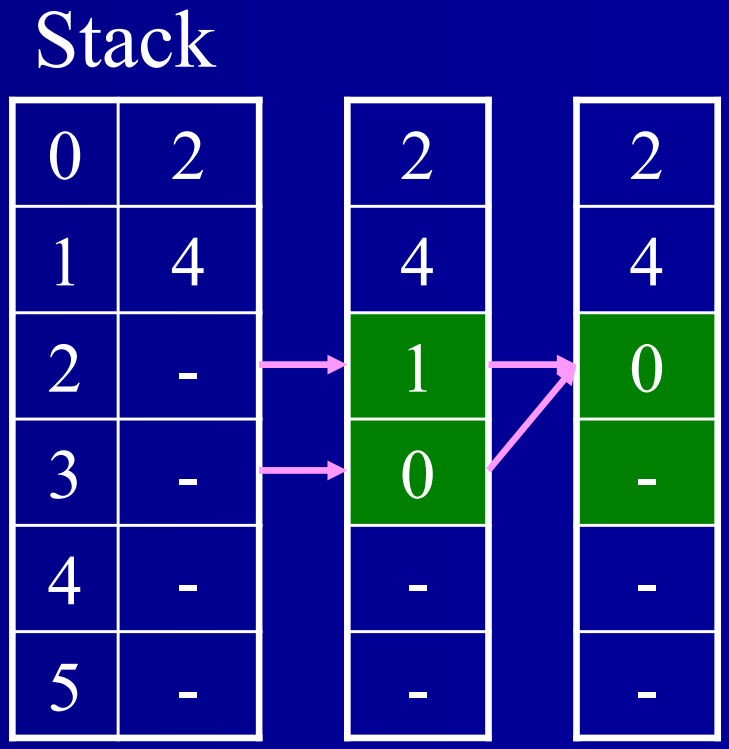
論理演算命令

0 = false, それ以外 = true として論理演算

演算結果は 0(false) か 1(true)

```
1 && 0
```

```
PUSHI 1
PUSHI 0
AND
```

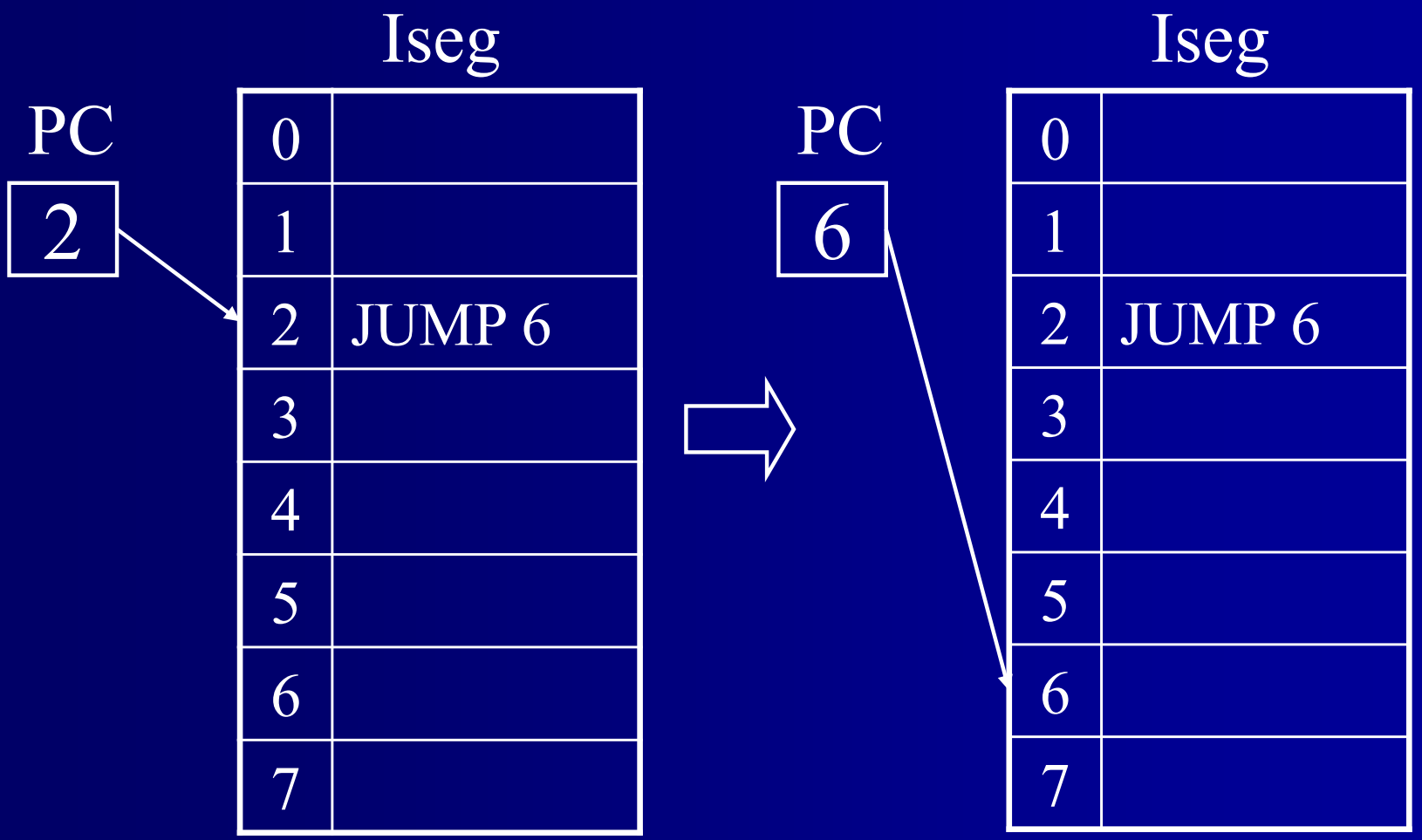


演算命令

命令	意味	
ADD	和	x + y
SUB	差	x - y
MUL	積	x * y
DIV	商	x / y
MOD	剰余	x % y
CSIGN	符号反転	- x
AND	論理積	x && y
OR	論理和	x || y
NOT	否定	! x

ジャンプ命令

- Program Counter を変更する



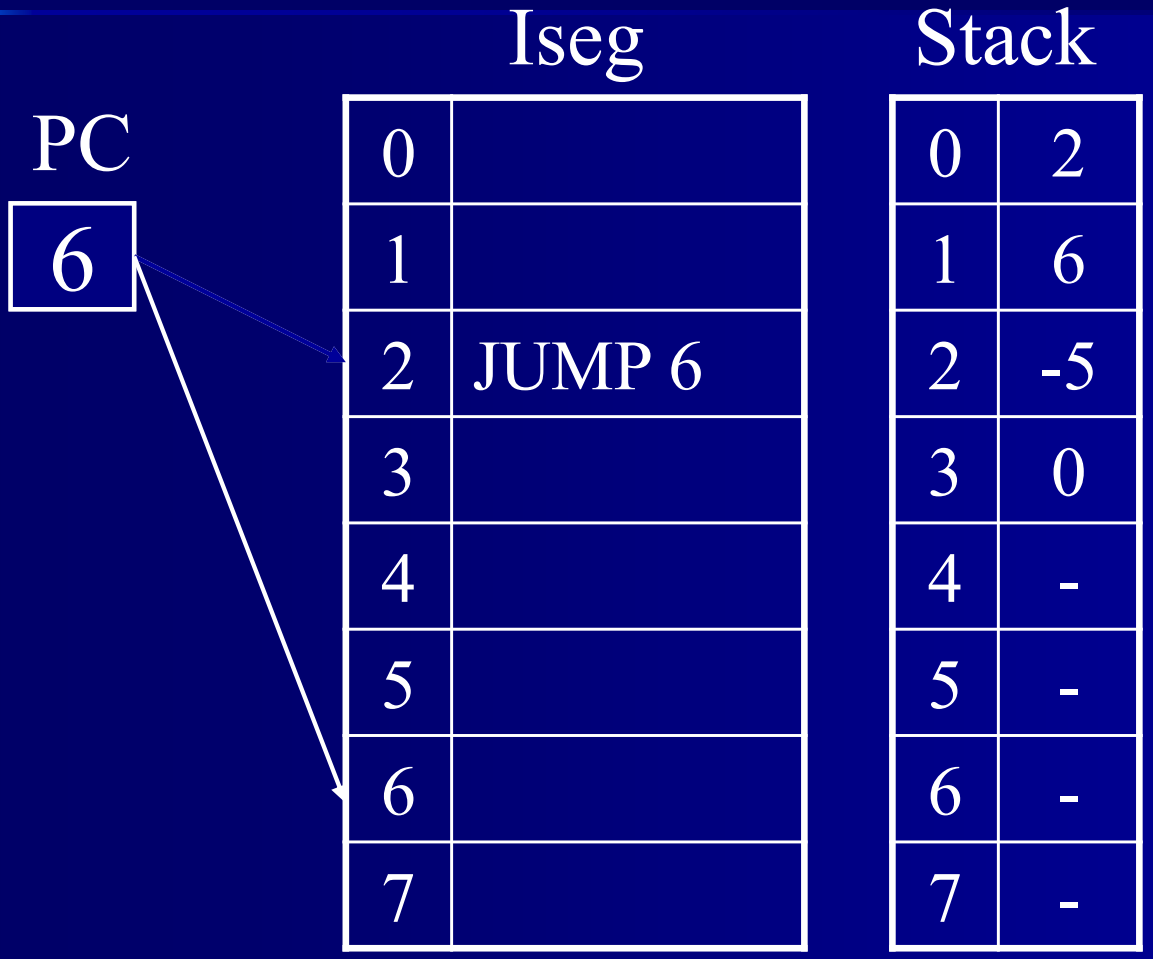
ジャンプ命令

命令	意味
JUMP	無条件ジャンプ
BEQ	条件付ジャンプ : if Stack == 0
BNE	条件付ジャンプ : if Stack != 0
BGE	条件付ジャンプ : if Stack >= 0
BGT	条件付ジャンプ : if Stack > 0
BLE	条件付ジャンプ : if Stack <= 0
BLT	条件付ジャンプ : if Stack < 0

Program Counter の動作

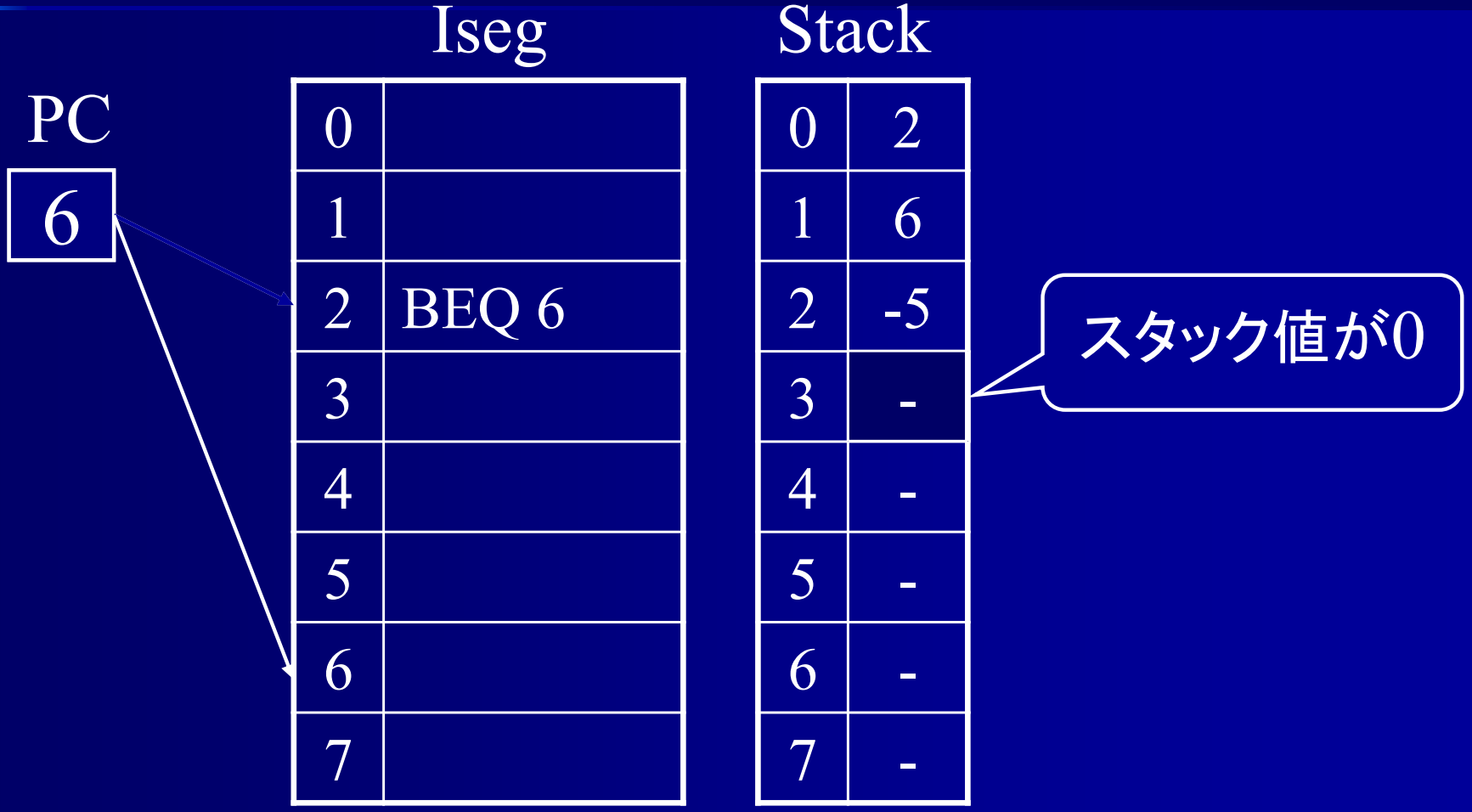
命令	スタックトップの値		
	Stack < 0	Stack == 0	Stack > 0
JUMP *i*	PC = *i*		
BEQ *i*	PC += 1	PC = *i*	PC += 1
BNE *i*	PC = *i*	PC += 1	PC = *i*
BLE *i*	PC = *i*	PC = *i*	PC += 1
BLT *i*	PC = *i*	PC += 1	PC += 1
BGE *i*	PC += 1	PC = *i*	PC = *i*
BGT *i*	PC += 1	PC += 1	PC = *i*
それ以外	PC += 1		

無条件ジャンプ



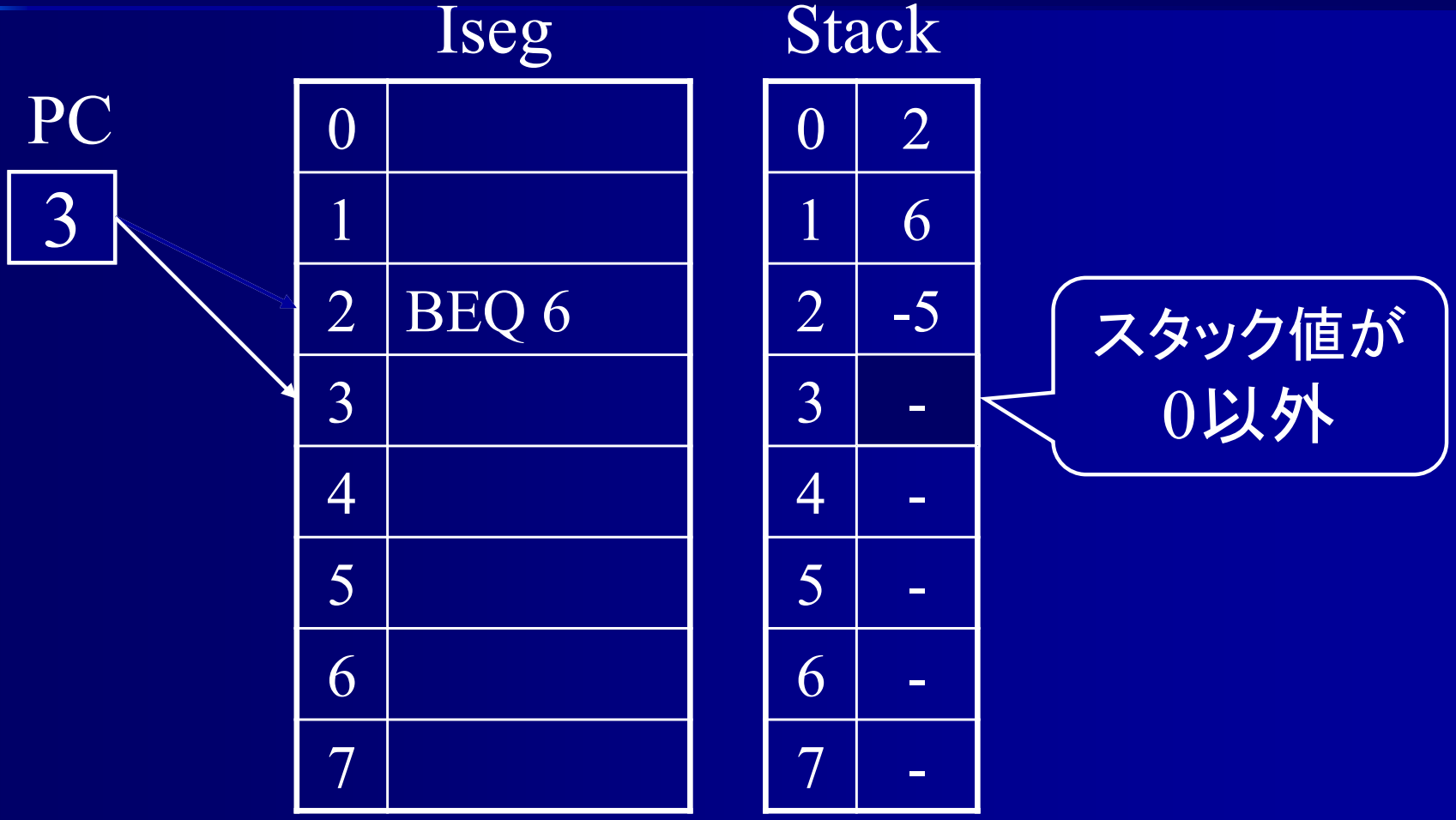
スタック値に関係なくジャンプ

条件付ジャンプ



スタック値が0ならばジャンプ

条件付ジャンプ

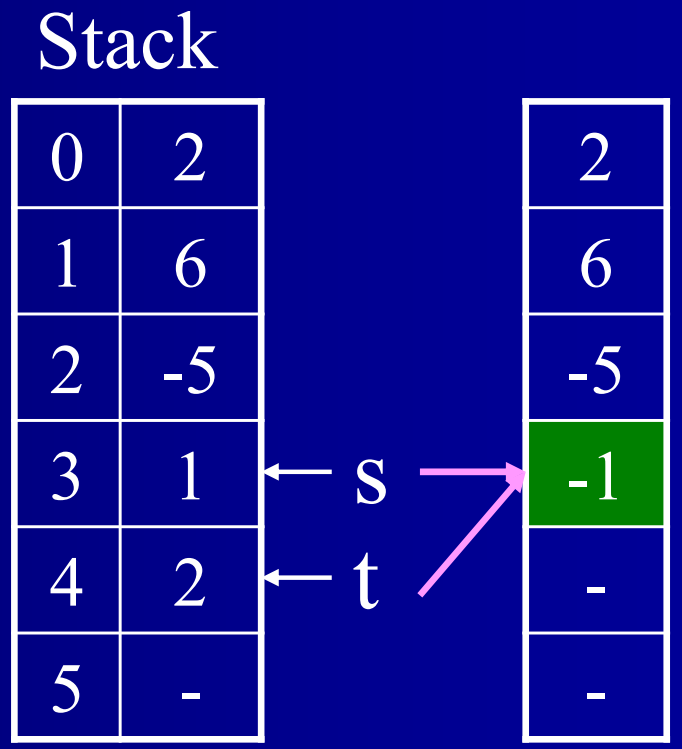


スタック値が0以外ならば次の行へ

比較
COMP命令

- スタックトップの値 t と2番目の値 s を比較
  - s == t のとき 0
  - s < t のとき -1
  - s > t のとき 1

```
PUSHI 1
PUSHI 2
COMP
```



比較命令

t : スタックトップ　s: スタックの2番目

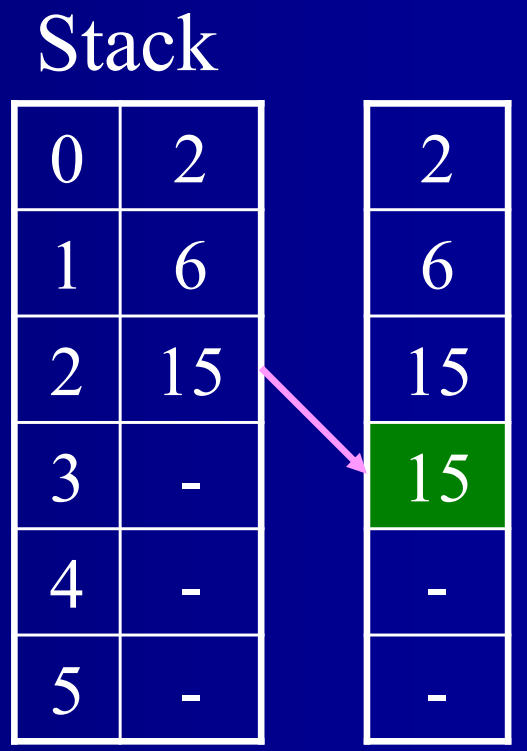
比較命令	s < t	s == t	s > t
COMP	-1	0	1
EQ	0	1	0
NE	1	0	1
LE	1	1	0
LT	1	0	0
GE	0	1	1
GT	0	0	1

情報システムプロジェクトI の
VSMアセンブラでは COMP のみ使用可

COPY 命令

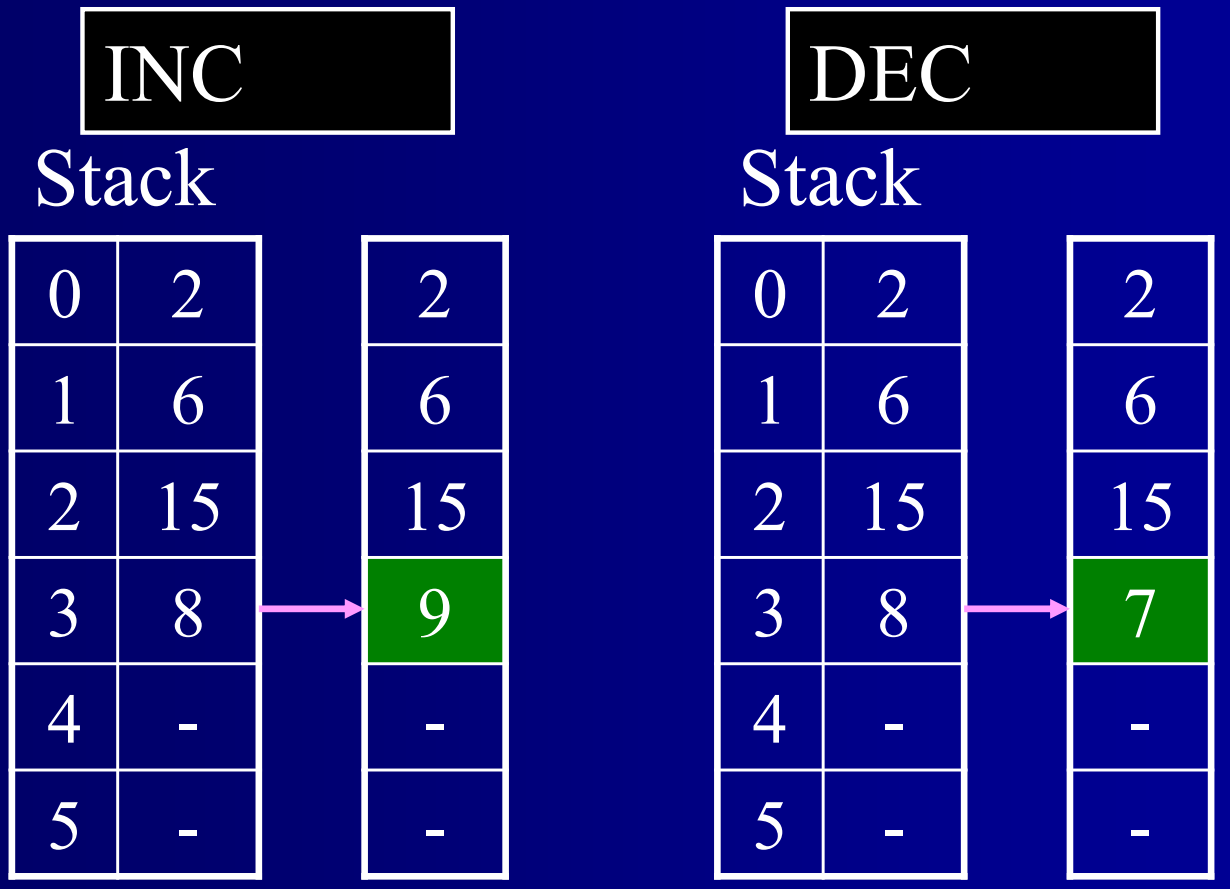
- スタックトップの値をコピー

```
PUSHI 15
COPY
```



INC 命令, DEC 命令

- スタックトップの値を1増減



変数の番地

変数宣言

```
int x = 1, y = 2, sum;
sum = x + 3;
```

Dseg

0	1	x
1	2	y
2	-	sum
3	-	-
4	-	-

変数表

名前	型	サイズ	番地
x	int	1	0
y	int	1	1
sum	int	1	2

変数の番地

変数の参照

```
int x = 1, y = 2, sum;
sum = x + 3;
```

Dseg

0	1	x	1
1	2	y	2
2	-	sum →	4
3	-	-	-
4	-	-	-

変数表

名前	型	サイズ	番地
x	int	1	0
y	int	1	1
sum	int	1	2

```
PUSHI  2
PUSH   0
PUSHI  3
ADD
ASSGN
REMOVE
```

sum の番地
x の番地
数値 3
加算
代入

```
sum = x + 3;
```

Iseg

```
0  PUSHI  2
1  PUSH   0
2  PUSHI  3
3  ADD
4  ASSGN
5  REMOVE
```

Dseg

d\i	0	1	2	3	4	5	
0	1	1	1	1	1	1	x
1	2	2	2	2	2	2	y
2					4	4	sum
3							-
4							-

Stack

s\i	0	1	2	3	4	5
0	2	2	2	2	4	
1		1	1	4		
2			3			
3						

配列

変数宣言

```
int i, j, a[5];
a[3] = 2;
```

変数表

名前	型	サイズ	番地
i	int	1	0
j	int	1	1
a	int[]	5	2

Dseg

0	-	i
1	-	j
2	-	a[0]
3	-	a[1]
4	-	a[2]
5	-	a[3]
6	-	a[4]
7	-	-

a[0] の番地

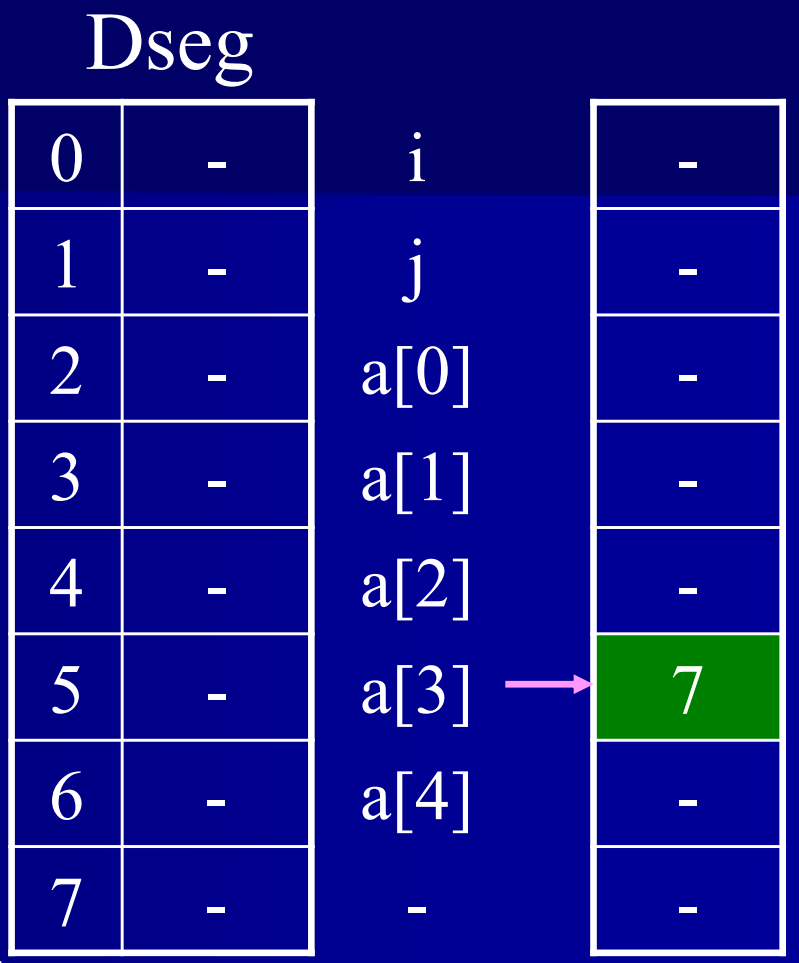
配列

変数宣言

```
int i, j, a[5];
a[3] = 7;
```

```
PUSHI  2
PUSHI  3
ADD
PUSHI  7
ASSGN
REMOVE
```

a[0] の番地
数値3
a[3]の番地計算
数値7
代入



a[i] の番地 = a[0]の番地 + i

```
a[3] = 7;
```

Iseg

```
0  PUSHI  2
1  PUSHI  3
2  ADD
3  PUSHI  7
4  ASSGN
5  REMOVE
```

Dseg

d\i	0	1	2	3	4	5	
0							i
1							j
2							a[0]
3							a[1]
4							a[2]
5					7	7	a[3]

Stack

s\i	0	1	2	3	4	5
0	2	2	5	5	7	
1		3		7		
2						
4						

データ参照

- スカラー変数 x の参照

左辺値

```
PUSHI  x のアドレス
```

右辺値

```
PUSH  x のアドレス
```

- 配列 a [3] の参照

左辺値

```
PUSHI  a[0] のアドレス
PUSHI  3
ADD
```

右辺値

```
PUSHI  a[0] のアドレス
PUSHI  3
ADD
LOAD
```

代入のアセンブラコード

<Expression> ::= <Exp> [“=” <Expression>]

<Expression> → <Exp> の場合

```
<Exp> のコード (右辺値)
```

<Expression> → <Exp> “=” <Expression> の場合

```
<Exp> のコード (左辺値)
<Expression> のコード (右辺値)
ASSGN
```

代入のアセンブラコード

例 : i = j

i の左辺値

j の右辺値

```
PUSHI  0
PUSH   1
ASSGN
```

名前	型	サイズ	番地
i	int	1	0
j	int	1	1
k	int	1	2
a	int[]	20	3
b	int[]	40	23

a[10] = b[20]

```
PUSHI  3
PUSHI 10
ADD
PUSHI 23
PUSHI 20
ADD
LOAD
ASSGN
```

i = j = k

```
PUSHI  0
PUSHI  1
PUSH   2
ASSGN
ASSGN
```

条件式のアセンブラコード

<LFactor> ::= <Exp>$_1$ “==” <Exp>$_2$

<Exp>$_1$ == <Exp>$_2$ ならば 1

```
        <Exp>1 のコード (右辺値)
        <Exp>2 のコード (右辺値)
        COMP
        BEQ     (L1)
        PUSHI 0
        JUMP    (L2)
(L1)    PUSHI 1
(L2)
```

3番地先へジャンプ

2番地先へジャンプ

条件式のアセンブラコード

例 : i == j

```
100 PUSH  i の番地
101 PUSH  j の番地
102 COMP
103 BEQ     106
104 PUSHI 0
105 JUMP  107
106 PUSHI 1
107
```

3番地先へジャンプ

2番地先へジャンプ

条件式のアセンブラコード

```
      COMP
      BEQ     (L1)
      PUSHI   0
      JUMP    (L2)
(L1)  PUSHI   1
(L2)
```

演算子	分岐コード
==	BEQ
!=	BNE
<=	BLE
<	BLT
>=	BGE
>	BGT

条件式のアセンブラコード

例 : i < j

```
100 PUSH  i の番地
101 PUSH  j の番地
102 COMP
103 BLT     106
104 PUSHI 0
105 JUMP  107
106 PUSHI 1
107
```

例 : i >= j

```
100 PUSH  i の番地
101 PUSH  j の番地
102 COMP
103 BGE     106
104 PUSHI 0
105 JUMP  107
106 PUSHI 1
107
```

条件式のアセンブラコード (EQ 命令がある場合)

<LFactor> ::= <Exp>$_1$ “==” <Exp>$_2$

<Exp>$_1$のコード
<Exp>$_2$のコード
EQ

演算子	比較命令
==	EQ
!=	NE
<=	LE
<	LT
>=	GE
>	GT

if 文(else節無し)のアセンブラコード

<If_St> ::= “if” “(” <Exp> “)” <St>

<Exp> のコード (右辺値)
BEQ (L)
 <St> のコード
(L)

<St> の次の命令の番地に分岐

(L) の番地は <St> のコードを作るまで不明

⇩

後から番地を書き直す必要あり

if 文のアセンブラコード

例 : if (f) i = 3;

```
100 PUSH   f の番地
101 BEQ    ?
```

この時点では
分岐先は不明

if 文のアセンブラコード

例 : if (f) i = 3;

```
100 PUSH   f の番地
101 BEQ    ?
102 PUSHI  i の番地
103 PUSHI  3
104 ASSGN
105 REMOVE
106
```

ここまでコードを作れば
分岐先が判明

if 文のアセンブラコード

例 : if (f) i = 3;

```
100 PUSH   f の番地
101 BEQ    106
102 PUSHI  i の番地
103 PUSHI  3
104 ASSGN
105 REMOVE
106
```

ここまでコードを作れば
分岐先が判明

while 文のアセンブラコード

<While_St> ::= “while” “(” <Exp> “)” <St>

```
(L1) <Exp> のコード (右辺値)
       BEQ  (L2)
       <St> のコード
       JUMP  (L1)
(L2)
```

JUMPの次の番地に分岐

条件式にジャンプ

(L2) の番地は <St> のコードを作るまで不明

⇩

後から番地を書き直す必要あり

while 文のアセンブラコード

例：while (f) i = 3;

```
100 PUSH   f の番地
101 BEQ    ?
```

この時点では
分岐先は不明

while 文のアセンブラコード

例：while (f) i = 3;

```
100 PUSH   f の番地
101 BEQ    ?
102 PUSHI  i の番地
103 PUSHI  3
104 ASSGN
105 REMOVE
106 JUMP   100
107
```

ここまでコードを作れば
分岐先が判明

while 文のアセンブラコード

例：while (f) i = 3;

```
100 PUSH   f の番地
101 BEQ    107
102 PUSHI  i の番地
103 PUSHI  3
104 ASSGN
105 REMOVE
106 JUMP   100
107
```

ここまでコードを作れば
分岐先が判明

if 文(else節無し) と while 文

<If_St> ::= “if” “(” <Exp> “)” <St>

<While_St> ::= “while” “(” <Exp> “)” <St>

if 文のコード

```
    <Exp> のコード
    BEQ (L)
     <St> のコード
(L)
```

while 文のコード

```
(L1) <Exp> のコード
       BEQ (L2)
        <St> のコード
       JUMP (L1)
(L2)
```

両者の差は JUMP 命令の有無のみ

for 文のアセンブラコード

<For_St> ::= “for”
“(” <Exp>$_1$ “;” <Exp>$_2$ “;” <Exp>$_3$ “)” <St>

```
        <Exp>1 のコード (右辺値)
        REMOVE
(L1)    <Exp>2 のコード (右辺値)
        BEQ   (L4)
        JUMP (L3)
(L2)    <Exp>3 のコード (右辺値)
        REMOVE
        JUMP (L1)
(L3)    <St> のコード
        JUMP (L2)
(L4)
```

for 文のアセンブラコード

<For_St> ::= “for”
“(” <Var_decl> “;” $<Exp>_2$ “;” $<Exp>_3$ “)” <St>

```
        <Var_decl> のコード
(L1)    <Exp>2 のコード (右辺値)
        BEQ   (L4)
        JUMP (L3)
(L2)    <Exp>3 のコード (右辺値)
        REMOVE
        JUMP (L1)
(L3)    <St> のコード
        JUMP (L2)
(L4)
```

前置 ++, -- のコード

<Unsigned> ::= (“++” | “--”) NAME
| (“++” | “--”) NAME “[” <Exp> “]”

<Unsigned> → “++” NAME の場合

```
PUSHI NAMEの番地
PUSH  NAMEの番地
INC
ASSGN
```

```
PUSH  NAMEの番地
INC
COPY
POP   NAMEの番地
```

```
++ y;
```

Iseg

```
0  PUSHI 1
1  PUSH  1
2  INC
3  ASSGN
```

Dseg

d\i	0	1	2	3	
0					x
1	5	5	5	6	y
2					a[0]
3					a[1]
4					a[2]

Stack

s\i	0	1	2	3
0	1	1	1	6
1		5	6	
2				
3				
4				

```
++ y;
```

Iseg

```
0  PUSH  1
1  INC
2  COPY
3  POP   1
```

Dseg

d\i	0	1	2	3	
0					x
1	5	5	5	6	y
2					a[0]
3					a[1]
4					a[2]

Stack

s\i	0	1	2	3
0	5	6	6	6
1			6	
2				
3				
4				

前置 ++, -- のコード

<Unsigned> ::= (“++” | “--”) NAME
| (“++” | “--”) NAME “[” <Exp> “]”

<Unsigned> → “++” NAME “[” <Exp> “]” の場合

```
PUSHI NAMEの番地
<Exp> のコード (右辺値)
ADD
COPY
LOAD
INC
ASSGN
```

```
++ a[1];
```

Iseg

```
0  PUSHI  2
1  PUSHI  1
2  ADD
3  COPY
4  LOAD
5  INC
6  ASSGN
```

Dseg

d\i	0	1	2	3	4	5	6	
0								x
1								y
2								a[0]
3	15	15	15	15	15	15	16	a[1]
4								a[2]

Stack

s\i	0	1	2	3	4	5	6
0	2	2	3	3	3	3	16
1		1		3	15	16	
2							
3							
4							

後置 ++, -- のコード

<Unsigned> ::= NAME (“++” | “--”)
| NAME “[” <Exp> “]” (“++” | “--”)

<Unsigned> → NAME “++” の場合

```
PUSH  NAMEの番地
COPY
INC
POP    NAMEの番地
```

配列の後置++は工夫が必要

```
y ++;
```

Iseg

```
0  PUSH  1
1  COPY
2  INC
3  POP   1
```

Dseg

d\i	0	1	2	3	
0					x
1	5	5	5	6	y
2					a[0]
3					a[1]
4					a[2]

Stack

s\i	0	1	2	3
0	5	5	5	5
1		5	6	
2				
3				
4				

前置++と後置++

<Unsigned> → “++” NAME の場合

```
PUSH  NAMEの番地
INC
COPY
POP    NAMEの番地
```

1 増やした後にコピー

⇩

増やした後の値が残る

<Unsigned> → NAME “++”の場合

```
PUSH  NAMEの番地
COPY
INC
POP    NAMEの番地
```

1 増やす前にコピー

⇩

増やす前の値が残る

後置 ++, -- のコード

前置++ の結果から1引けば演算前の値になる

i++ ⇔ (++i) -1

配列の前置++

```
PUSHI NAMEの番地
<Exp> のコード (右辺値)
ADD
COPY
LOAD
INC
ASSGN
```

配列の後置++

```
PUSHI NAMEの番地
<Exp> のコード (右辺値)
ADD
COPY
LOAD
INC
ASSGN
DEC
```

配列の後置++ は前置++の最後にDEC を付ける

加算代入のアセンブラコード

<Expression> ::= <Exp> “+=” <Expression>

```
<Exp> のコード (左辺値)
COPY
LOAD
<Expression> のコード (右辺値)
ADD
ASSGN
```

```
x += 5;
```

Iseg

```
0  PUSHI  0
1  COPY
2  LOAD
3  PUSHI  5
4  ADD
5  ASSGN
```

Dseg

d\i	0	1	2	3	4	5	
0	10	10	10	10	10	15	x
1							y
2							a[0]
3							a[1]
4							a[2]

Stack

s\i	0	1	2	3	4	5
0	0	0	0	0	0	15
1		0	10	10	15	
2				5		
3						
4						

式文のアセンブラコード

<Exp_St> ::= <Exp> “;”

<Exp> のコード (右辺値)
REMOVE

スタックトップに残った
式の評価値を削除

“;” が来れば式終了 ⇒ 式の評価値はもう不要

break 文のアセンブラコード

<Break_St> ::= “break” “;”

JUMP (対応するループ, switch 文の外へ)

<Continue_St> ::= “continue” “;”

JUMP (対応するループの条件式へ)

(※) for 文は継続式(式3)へ

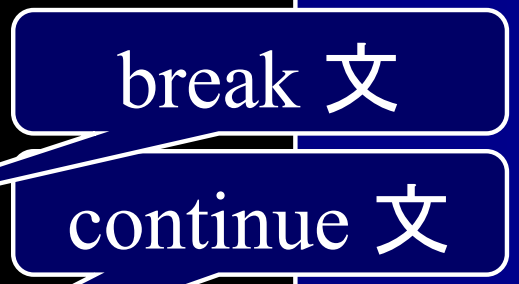
対応するループが無ければエラー

break 文のアセンブラコード
(while 文からの脱出の場合)

while (<Exp>)
{ <St_1> break ;
<St_2> continue
<St_3> }

の場合

(L1) <Exp> のコード (右辺値)
BEQ (L2)
<St_1> のコード
JUMP (L2)
<St_2> のコード
JUMP (L1)
<St_3> のコード
JUMP (L1)
(L2)



break 文 : ループ外へ
continue 文 : 継続式へ
while 文終了時に
break 文の飛び先決定

プログラム末尾のアセンブラコード

<Program> ::= <Main> “$”

ファイル末

<Main> のコード
HALT

末尾に HALT を積む

2次元配列

```
int a[M][N];
```

N

a	0	1	2	3
0	-	-	-	-
1	-	-	20	-
2	-	-	-	-
3	-	-	-	-
4	-	-	-	-

M

```
a[1][2] = 20;
```

Dseg

0	-	a[0][0]	10	-	a[2][2]
1	-	a[0][1]	11	-	a[2][3]
2	-	a[0][2]	12	-	a[3][0]
3	-	a[0][3]	13	-	a[3][1]
4	-	a[1][0]	14	-	a[3][2]
5	-	a[1][1]	15	-	a[3][3]
6	20	a[1][2]	16	-	a[4][0]
7	-	a[1][3]	17	-	a[4][1]
8	-	a[2][0]	18	-	a[4][2]
9	-	a[2][1]	19	-	a[4][3]

配列のアドレス

多次元配列の
アドレス計算は
各次元の大きさが必要

1次元配列

```
int a[N];
```

a[i] のアドレス : (a[0] のアドレス) + i

2次元配列

```
int a[M][N];
```

a[i][j] のアドレス : (a[0][0] のアドレス) + N*i + j

3次元配列

```
int a[L][M][N];
```

a[i][j][k] のアドレス : (a[0][0][0] のアドレス)
+ M*N*i + N*j + k

配列のアドレス

a[<Exp>$_1$]

```
PUSHI a[0] の番地
<Exp>1 のコード (右辺値)
ADD
```

a[<Exp>$_1$][<Exp>$_2$]

```
PUSHI a[0][0] の番地
<Exp>1 のコード (右辺値)
PUSHI  N
MUL
ADD
<Exp>2 のコード (右辺値)
ADD
```

a[<Exp>$_1$][<Exp>$_2$][<Exp>$_3$]

```
PUSHI a[0][0][0] の番地
<Exp>1 のコード (右辺値)
PUSHI M*N
MUL
ADD
<Exp>2 のコード (右辺値)
PUSHI N
MUL
ADD
<Exp>3 のコード (右辺値)
ADD
```

if 文(else節有り)の アセンブラコード

<If_St> ::= “if” “(” <Exp> “)” <St>$_1$ [“else” <St>$_2$]

```
      <Exp> のコード (右辺値)
      BEQ (L1)
       <St>1 のコード
(L1)
```

else 節無し

```
      <Exp> のコード (右辺値)
      BEQ (L1)
       <St>1 のコード
      JUMP (L2)
(L1)  <St>2 のコード
(L2)
```

else 節有り

do-while 文のアセンブラコード

<Do_St> ::= “do” <St> “while” “(” <Exp> “)” “;”

```
(L)   <St> のコード
      <Exp> のコード (右辺値)
      BNE (L)
```

for 文のアセンブラコード

<For_St> ::= “for”
“(” <Exp>$_1$ “;” <Exp>$_2$ “;” <Exp>$_3$ “)” <St>

```
        <Exp>1 のコード (右辺値)
        REMOVE
(L1)    <Exp>2 のコード (右辺値)
        BEQ   (L4)
        JUMP (L3)
(L2)    <Exp>3 のコード (右辺値)
        REMOVE
        JUMP (L1)
(L3)    <St> のコード
        JUMP (L2)
(L4)
```

for 文のアセンブラコード

<For_St> ::= “for” “(” [<Exp>$_1$ { “,” <Exp>$_{1'}$ }] “;”
[<Exp>$_2$] “;”
[<Exp>$_3$ { “,” <Exp>$_{3'}$ }] “)” <St>

for (<Exp>$_{11}$, <Exp>$_{12}$;
<Exp>$_2$;
<Exp>$_{31}$, <Exp>$_{32}$) <St> の場合

<Exp>$_{11}$ のコード (右辺値)
REMOVE
<Exp>$_{12}$ のコード (右辺値)
REMOVE
(L1) <Exp>$_2$ のコード (右辺値)
BEQ (L4)
JUMP (L3)

(L2) <Exp>$_{31}$ のコード (右辺値)
REMOVE
<Exp>$_{32}$ のコード (右辺値)
REMOVE
JUMP (L1)
(L3) <St> のコード
JUMP (L2)
(L4)

for 文のアセンブラコード

<For_St> ::= “for” “(” [<Exp>$_1$ { “,” <Exp>$_{1'}$ }] “;”
[<Exp>$_2$] “;”
[<Exp>$_3$ { “,” <Exp>$_{3'}$ }] “)” <St>

for (; ;) <St> の場合

<Exp>$_2$ を省略すると無限ループ

(L1) JUMP (L3)
(L2) JUMP (L1)
(L3) <St> のコード
JUMP (L2)
(L4)

無限ループ
＝ループ外へ出る
分岐命令無し

switch 文のアセンブラコード

<Switch_St> ::= “switch” “(” <Exp> “)” “{” { <St> } “}”

```
        <Exp> のコード (右辺値)
        JUMP (最初の case 値ラベルへ)
        <St> のコード
(L)  REMOVE
```

<Case_Lb> ::= “case” <Const> “:”

```
        JUMP (L’)
(L) COPY
        <Const> のコード (右辺値)
        COMP
        BNE (次の case 値ラベルへ)
(L’)
```

<Default_Lb> ::= “default” “:” ⇒ ラベルのみでコード無し

switch 文のアセンブラコード

switch (<Exp>)
{ case <$Const_1$> : <St_1> break ;
case <$Const_2$> : <St_2> break ;
default : <St_3> break ; } の場合

<Exp> のコード (右辺値)
JUMP (L1)
JUMP (L1')
(L1) COPY
<$Const_1$> のコード (右辺値)
COMP
BNE (L2)
(L1') <St_1> のコード
JUMP (L4)

JUMP (L2')
(L2) COPY
<$Const_2$> のコード (右辺値)
COMP
BNE (L3)
(L2') <St_2> のコード
JUMP (L4)
(L3) <St_3> のコード
JUMP (L4)
(L4) REMOVE

switch 文のアセンブラコード

switch (<Exp>)
{ case <$Const_1$> :
case <$Const_2$> : <St_1> break ;
default “:” <St_2> break ; } の場合

<Exp> のコード (右辺値)
JUMP (L1)
JUMP (L1’)
(L1) COPY
<$Const_1$> のコード (右辺値)
COMP
BNE (L2)
(L1’) JUMP (L2’)

(L2) COPY
<$Const_2$> のコード (右辺値)
COMP
BNE (L3)
(L2’) <St_1> のコード
JUMP (L4)
(L3) <St_2> のコード
JUMP (L4)
(L4) REMOVE

outputstr文のアセンブラコード

<Outputstr_st>

::= “outputstr” “(” STRING | NAME “)” “;”

outputstr (“hello”) の場合

```
PUSHI ‘h’
OUTPUTC
PUSHI ‘e’
OUTPUTC
   :
PUSHI ‘o’
OUTPUTC
OUTPUTLN
```

文字列の長さ分繰り返す

outputstr文のアセンブラコード

<Outputstr_st>

::= “outputstr” “(” STRING | NAME “)” “;”

Outputstr (str) の場合 (char str[5])

```
PUSH str[0] の番地
OUTPUTC
PUSH str[1] の番地
OUTPUTC
    :
PUSH str[4] の番地
OUTPUTC
OUTPUTLN
```

配列 str の長さ分繰り返す

setstr文のアセンブラコード

<Setstr_st>

::= “setstr” “(” NAME, STRING | NAME “)” “;”

setstr (mes, “what?”) の場合

```
PUSHI ‘w’
POP mes[0] の番地
PUSHI ‘h’
POP mes[1] の番地
    :
PUSHI ‘?’
POP mes[4] の番地
```

文字列の長さ分繰り返す

setstr文のアセンブラコード

<Setstr_st>

::= “setstr” “(” NAME, STRING | NAME “)” “;”

setstr (mes, str) の場合 (char mes[10], str[5])

PUSH str[0] の番地
POP mes[0] の番地
PUSH str[1] の番地
POP mes[1] の番地
:
PUSH str[4] の番地
POP mes[4] の番地

配列 str の長さ分繰り返す