情報工学実験1
～プログラムの動作原理～

山田 浩史

本実験の目的

- プログラムが実際のコンピュータでどのように動作しているかを理解・実感する
  - これまで学んだ 2 つの理解を結びつける
    - プログラミング(プログラミング序論・演習など)
    - コンピュータアーキテクチャ(計算機アーキテクチャ基礎，計算機の動作原理(1つ前の実験))

※ Linux 上で課題をこなしてください

- EDEN の環境を想定した課題になっています

内容

- 自分たちが作ったプログラムがどのように実行されているかを知る
  - メモリ上でプログラムがどう展開されるか?
  - プログラムがどう実行されているか?

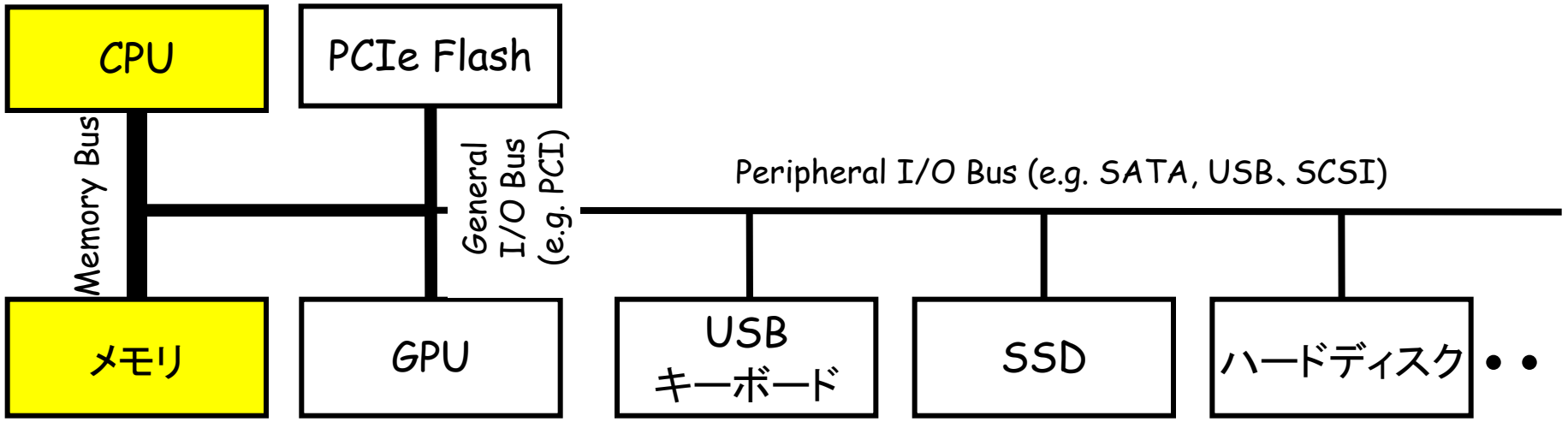
- x86 命令セットを題材にプログラムの具体的な動作を知る
  - x86: 最も広く利用されている命令セット
    - Intel や AMD 製の CPU はほとんどこれ
  - 周りのコンピュータは実際にどう動くのか?

Why プログラムの動作原理?

- 大事なんだけど講義で触れられない話
  - コンピュータを勉強する上では必須の知識
  - 動作原理を知っていればデバッグや用途に応じてツールを使い分けられる
- コンピュータの動作全体の理解を促す
  - ブラックボックスの見通しがよくなる
- システムプログラミングの「序の口」
  - システムプログラミング: OS や CPU の機能を使いまくるプログラミング
  - Computer Science を学んだ者のみが許されるプログラミングの世界

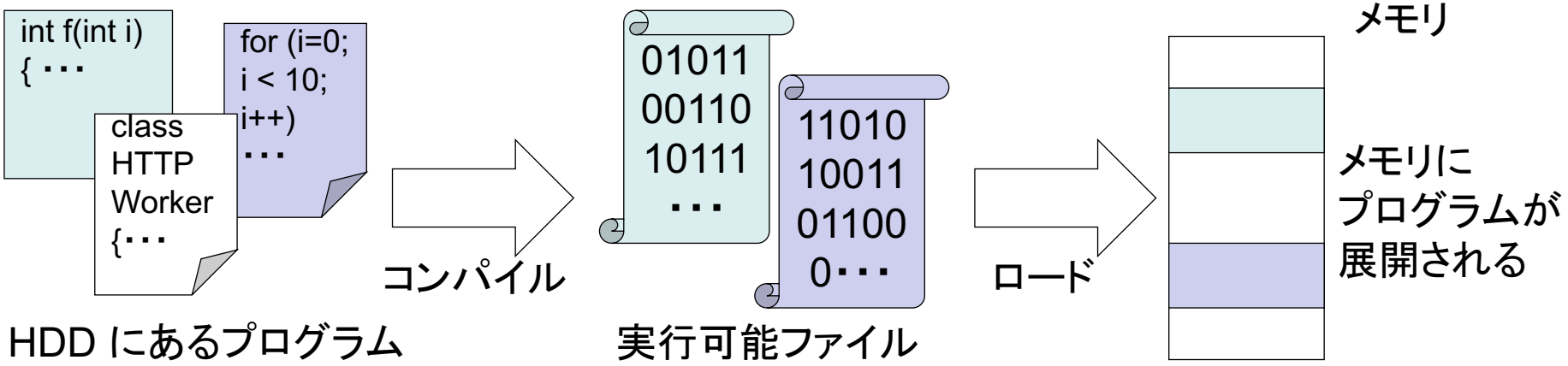
コンピュータ(汎用計算機)の構成

- CPU
  - 算術演算、論理演算、メモリ・I/O デバイスとのやりとりを司る
- メモリ(Memory)
  - データとプログラムをおぼえておく場所
- I/O デバイス
  - （おおざっぱには） CPU とメモリ以外の装置
- バス (Bus)
  - それぞれを繋ぐ伝送路
    (Memory Bus，General I/O Bus，Peripheral I/O Bus)



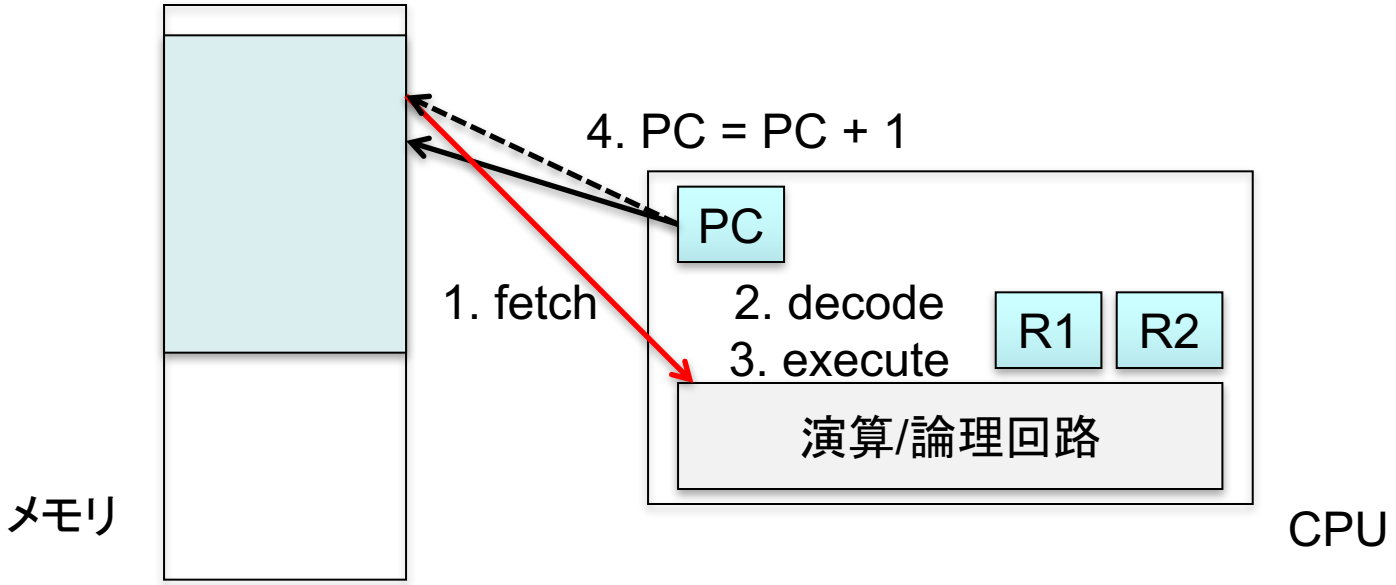
プログラムを動かすまで

1. プログラムを作る
   - コンピュータにさせることをプログラミング言語で記述
     - C 言語を使ったり，Java 言語を使ったり
2. コンパイルする
   - プログラミング言語から機械語(CPU が理解できる形式)に変換する
     - コンパイラと呼ばれるソフトウェアが行う
3. プログラムをメモリ上に展開して実行する
   - CPU は展開されたメモリを参照しながら動作する



CPU の動作(超簡略)

- プログラムカウンタ(Program Counter，PC)で指されているメモリアドレスの命令を実行する
  - PC はメモリのどこかを指している
- 次のことを無限に繰り返している
  1. PC で差されているメモリ番地から命令を取得(fetch)
  2. 取得した命令を解釈(decode)
  3. 命令を実行(execute)
  4. PC に 1 を加える / JUMP 命令で PC が書き換わる

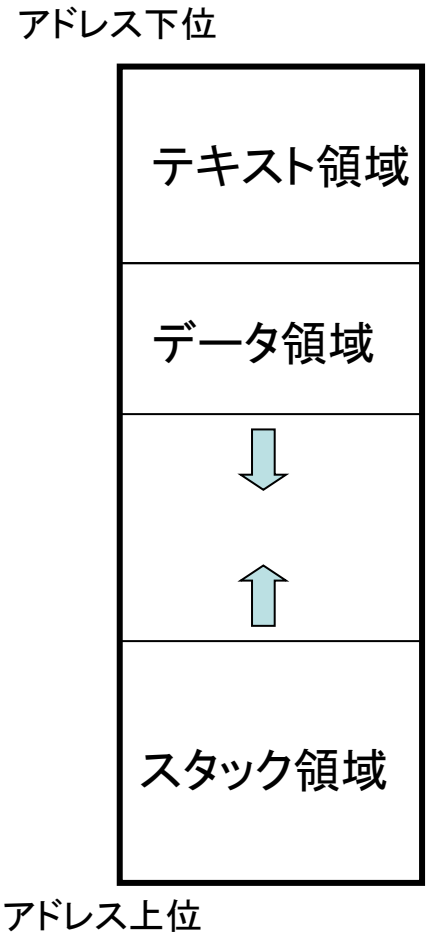


Questions…

- プログラムはどんな形でメモリに展開されて、どう動いているのか
  - 再帰呼び出しとか malloc() とかすると何が起こるの？
- 実際の CPU はどうやってプログラムを実行しているのか
  - ホントにアーキテクチャで勉強したことが周りのコンピュータで行われているの?
    - 説明のため、簡単なアセンブラを用いることが多い

メモリ内でのプログラム

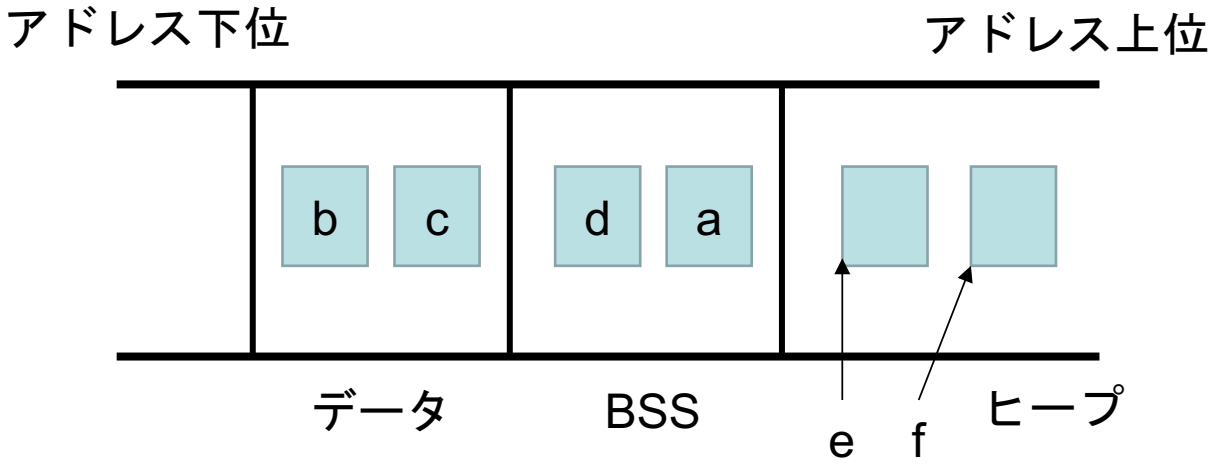
- プログラムをロードすると、以下の領域ができる
  - テキスト領域(a.k.a. コード領域)
    - プログラム（機械語命令の列）を格納
    - 実行可能ファイルにすべて記述されている
    - プログラムカウンタによって指される
  - データ領域
    - データ(大域変数や malloc() で割り当てたメモリ)を格納
    - プログラム実行時に決まる値は実行可能ファイルに記述されている
  - スタック領域
    - 関数の戻り番地や局所変数などを格納
    - スタックポインタによって指される



テキスト領域とデータ領域

- テキスト領域
  - 機械語を格納したら読み込み専用となる
- データ領域
  - 主として 3 つの領域に別れる
    - データ: 初期化された大域/静的変数を格納
    - BSS: 初期化されていない大域/静的変数を格納
    - ヒープ: malloc() などで確保された領域を格納

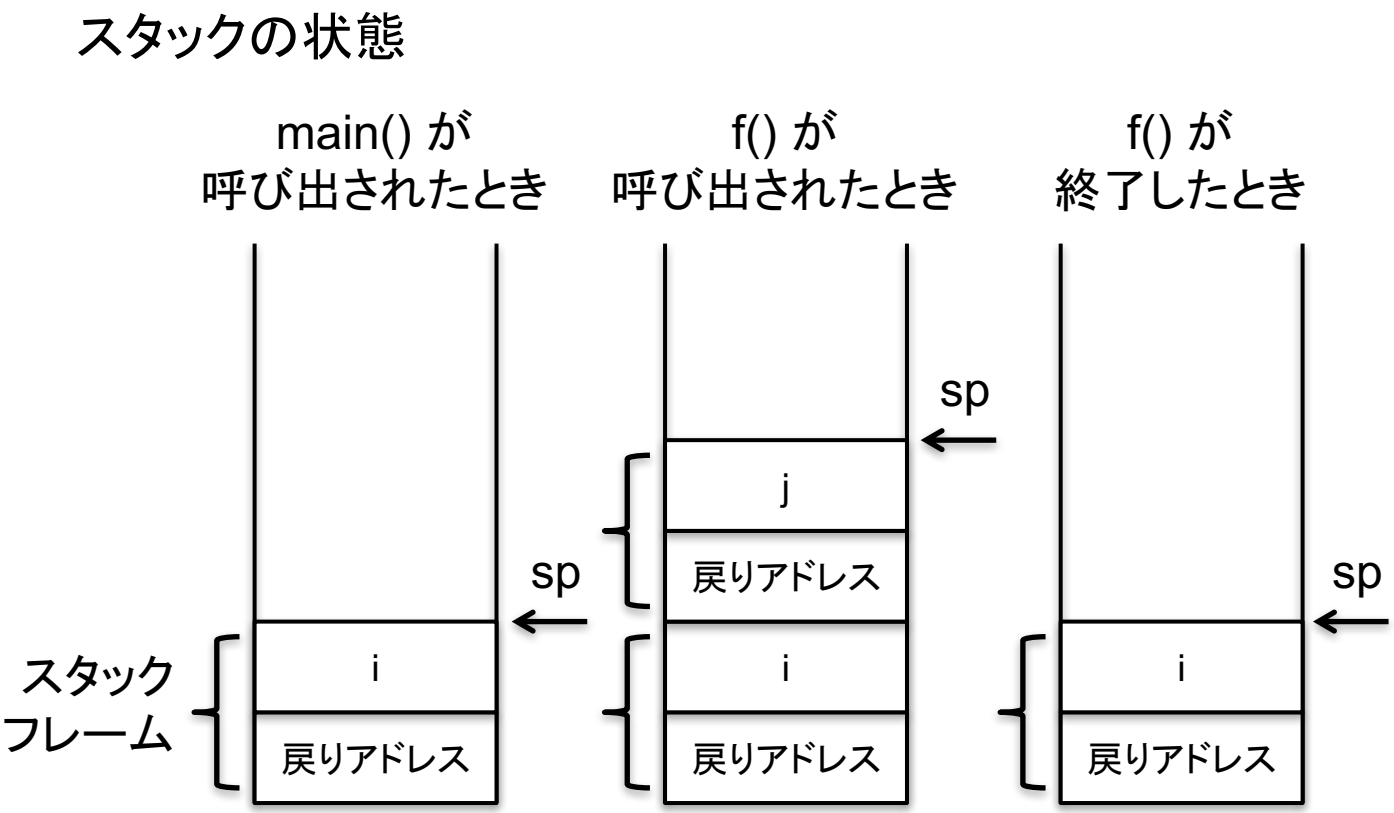
```
int a;
int b = 100;
static int c = 100;
static int d;
….
int main()
{ …
   e = malloc(sizeof(int));
   f  = malloc(sizeof(int));
}
```



スタック領域

- 関数の戻りアドレスや局所変数が格納される

```
int f(int i)
{
    int j = i;
    printf(“%d¥n”, j);
    return 0;
}

int main()
{
    int i;

    i = 1000;
    f(i);
    return 0;
}
```

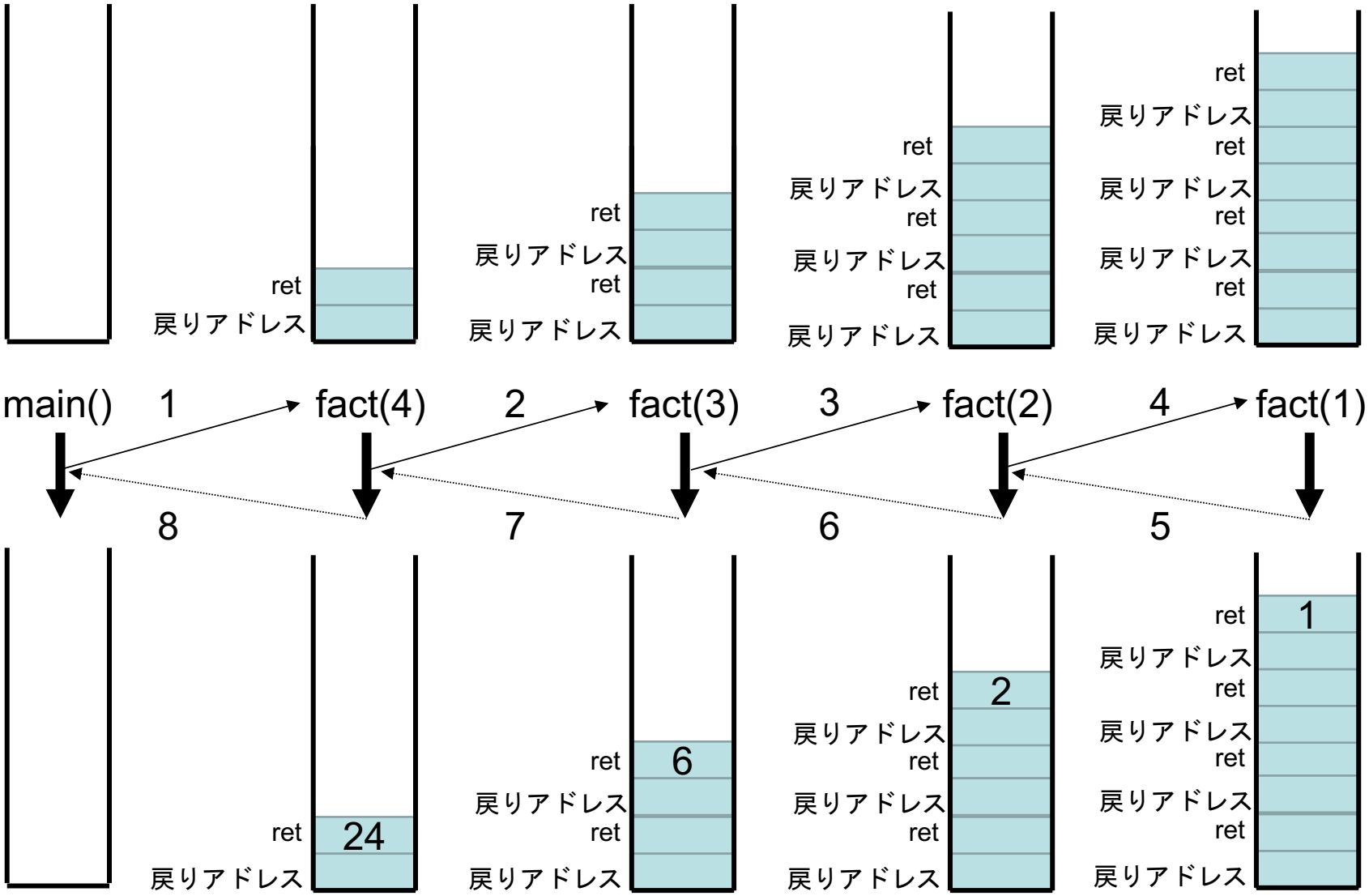


再帰関数

- 関数を呼び出しまくる
  ⇒ スタックフレームが積まれまくる

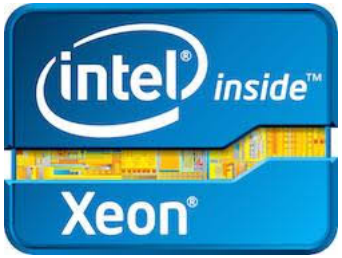
```
int fact(int num)
{
    int ret;
    if (num == 1) ret = 1;
    else ret = num * fact(num-1)
    return ret;
}

int main()
{
    ...
    value = fact(4);
    …
}
```

再帰関数(fact(4))



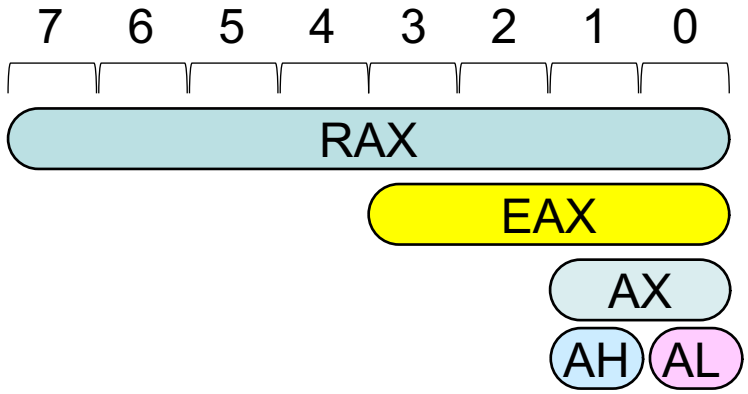
x86 命令セット

- Intel や AMD の CPU がサポートしている命令セット
  - 世で一番利用されている命令セット
- CISC(Complex Instruction Set Computer)に分類される
  - ⇔ RISC(Reduced Instruction Set Computer)



レジスタ

- 汎用レジスタ
  - RAX, RBP, RSP, RBX, RCX, RDX, …
  - 32 bit, 16 bit, 8 bit のレジスタとして アクセス可能
- フラグレジスタ
  - RFLAGS
- プログラムカウンタ
  - RIP
- 他にも・・・
  - コントロールレジスタ，FPU用レジスタなど



よく使うレジスタ

- RAX: アキュームレータ
  - 演算結果や返り値を格納する
  - int 型を使うことが多いため EAX で登場することが多い
- RSP: スタックポインタ
  - スタックの先頭を指す
- RBP: ベースポインタ
  - スタックフレームの底を指す
  - = 前のスタックフレームの先頭を指す

命令

- 基本的な命令はサポートされている
  - 名前から察せると思います

- データ転送命令
  - ✓ mov

- スタックの利用
  - ✓ push
  - ✓ pop

- 関数
  - ✓ call
  - ✓ leave
  - ✓ ret

- 算術命令
  - ✓ add / sub
  - ✓ imul / idiv
  - ✓ inc / dec

- 条件分岐
  - ✓ cmp
  - ✓ jmp系

プログラムをアセンブラに変えてみよう

- gcc の –S オプションを付けてコンパイル
  - test.c をコンパイルしたら test.s というファイルが生成される
    - GNUアセンブラで記述されたアセンブラが見られる

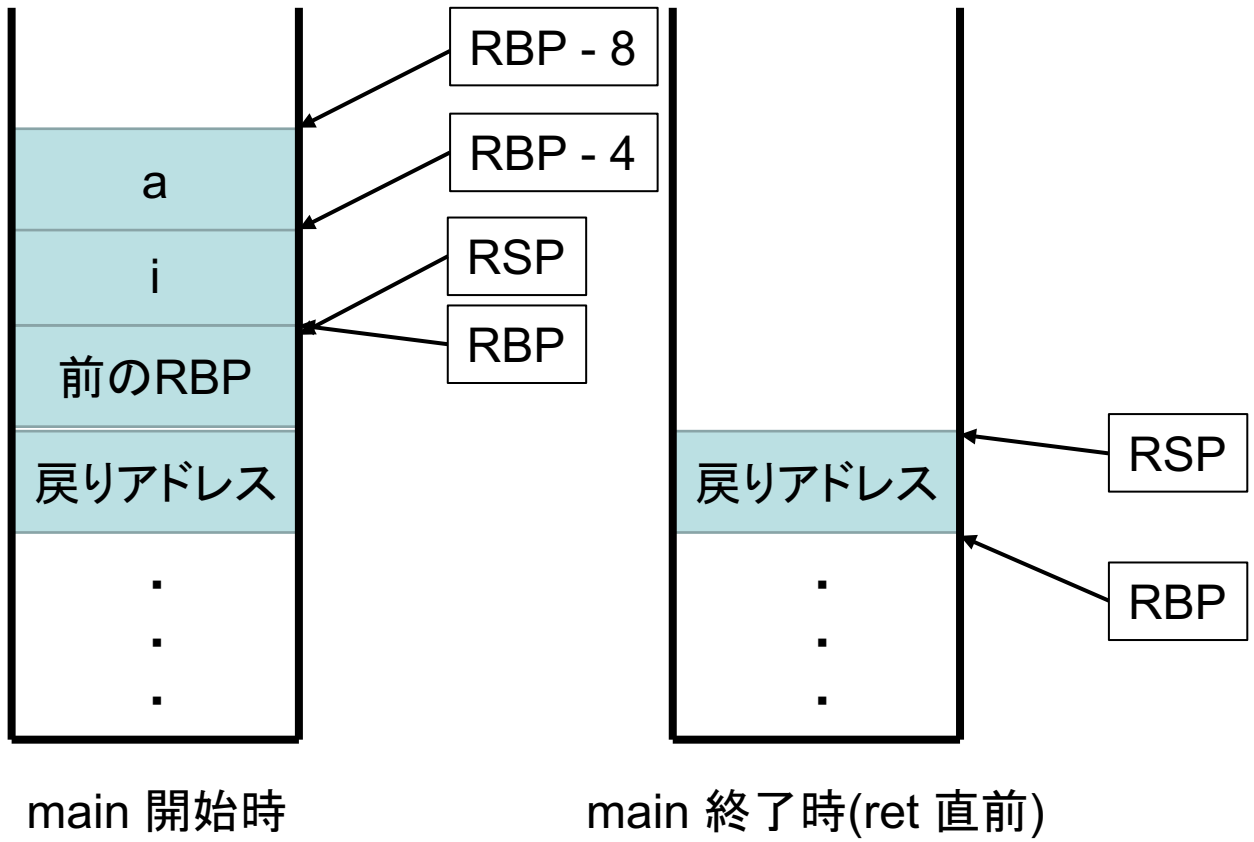
```
int main(void)
{
    int a, i;
    a = 0;
    for (i = 1; i < 11; i++)
        a = a + i;
    return 0;
}
```

```
main:.LFB0:
    pushq %rbp
    movq  %rsp, %rbp
    movl   $0, -8(%rbp)
    movl   $1, -4(%rbp)
    jmp    .L2
.L3:
    movl -4(%rbp), %eax
    addl   %eax, -8(%rbp)
    addl   $1, -4(%rbp)
.L2:
    cmpl   $10, -4(%rbp)
    jle    .L3
    movl   $0, %eax
    popq   %rbp
    ret
```

実行の様子

- RBP 活用して局所変数にアクセスする

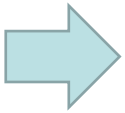
```
main:.LFB0:
    pushq %rbp
    movq  %rsp, %rbp
    movl   $0, -8(%rbp)
    movl   $1, -4(%rbp)
    jmp    .L2
.L3:
    movl -4(%rbp), %eax
    addl   %eax, -8(%rbp)
    addl   $1, -4(%rbp)
.L2:
    cmpl   $10, -4(%rbp)
    jle    .L3
    movl   $0, %eax
    popq   %rbp
    ret
```



もういっちょ

- gcc –S でソースコードをコンパイル

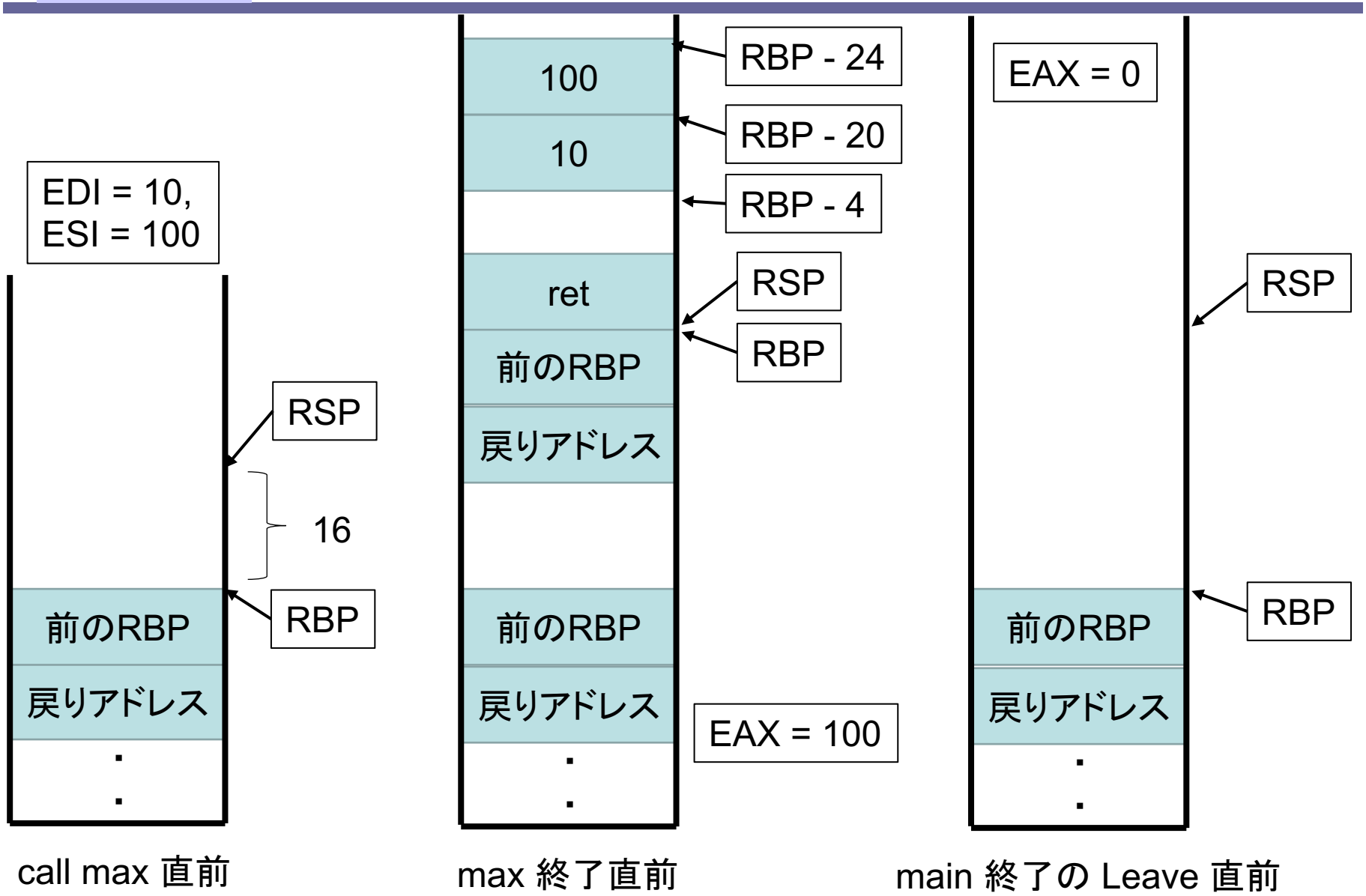
```
int max(int a, int b)
{
    int ret;
    if (a < b) ret = b;
    else       ret = a;
    return ret;
}

int main()
{
    int a;
    a = max(10,100);
    return 0;
}
```



```
max:
.LFB0:
    pushq %rbp
    movq %rsp, %rbp
    movl %edi, -20(%rbp)
    movl %esi, -24(%rbp)
    movl -20(%rbp), %eax
    cmpl -24(%rbp), %eax
    jge .L2
    movl -24(%rbp), %eax
    movl %eax, -4(%rbp)
    jmp .L3
.L2:
    movl -20(%rbp), %eax
    movl %eax, -4(%rbp)
.L3:
    movl -4(%rbp), %eax
    popq %rbp ret
```

```
main:
.LFB1:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp
    movl $100, %esi
    movl $10, %edi
    call max
    movl %eax, -4(%rbp)
    movl $0, %eax
    leave
    ret
```

関数ジャンプの様子


EDI = 10,
ESI = 100
RSP
16
RBP
前のRBP
戻りアドレス
call max 直前
100
10
ret
前のRBP
戻りアドレス
RBP - 24
RBP - 20
RBP - 4
RSP
RBP
前のRBP
戻りアドレス
EAX = 100
max 終了直前
EAX = 0
RSP
RBP
前のRBP
戻りアドレス
main 終了の Leave 直前

スタック破壊

- 局所変数を誤って使用すると簡単にスタックが壊れてしまう
  - スタックの破損 = プログラムの破壊
    - 局所変数変更による意図せぬ挙動
    - 戻りアドレス変更による意図せぬ挙動
- 巧妙に破壊されると、プログラムの動きが乗っ取られてしまう
  - ネットワークの攻撃の動作原理となっている

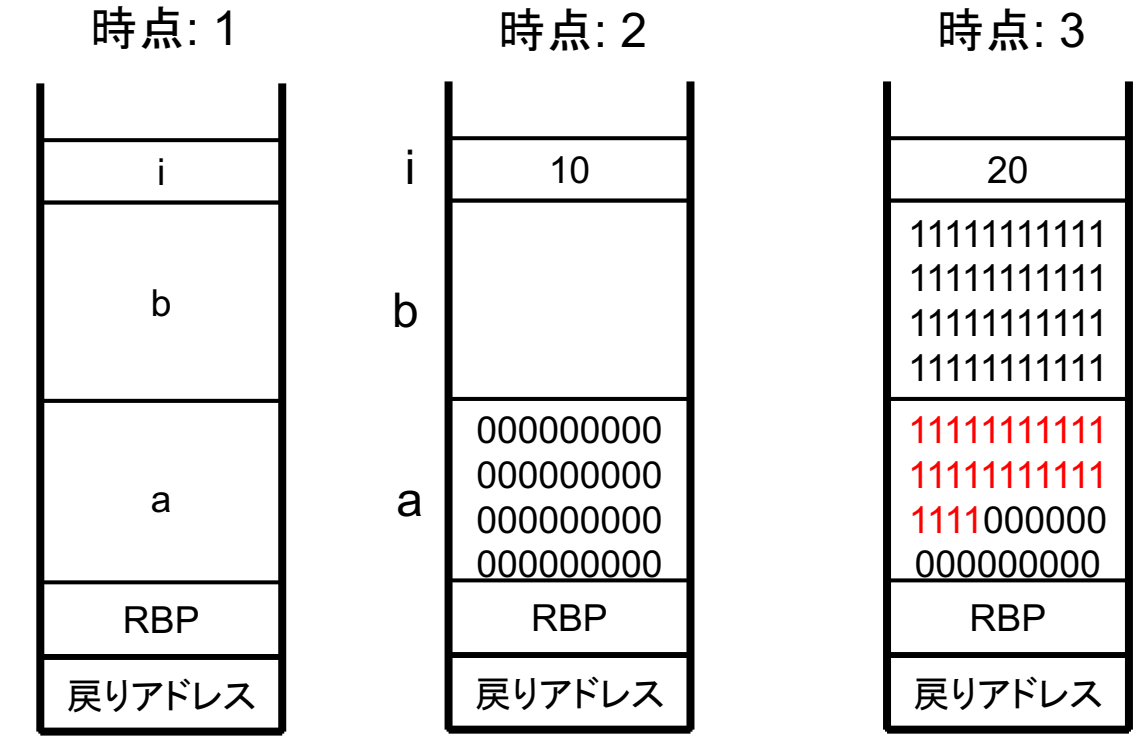
スタック破壊の例

- スタックは簡単に壊せる

```
#include <stdio.h>
int main()
{
    int a[10], b[10], i;

    for (i = 0; i < 10; i++)
        a[i] = 0;
    for (i = 0; i < 20; i++)
        b[i] = 1;
    for (i = 0; i < 10; i++)
        printf("%d¥n", a[i]);
    return 0;
}
```

1
2
3

スタックの様子



レポートについて(1/2)

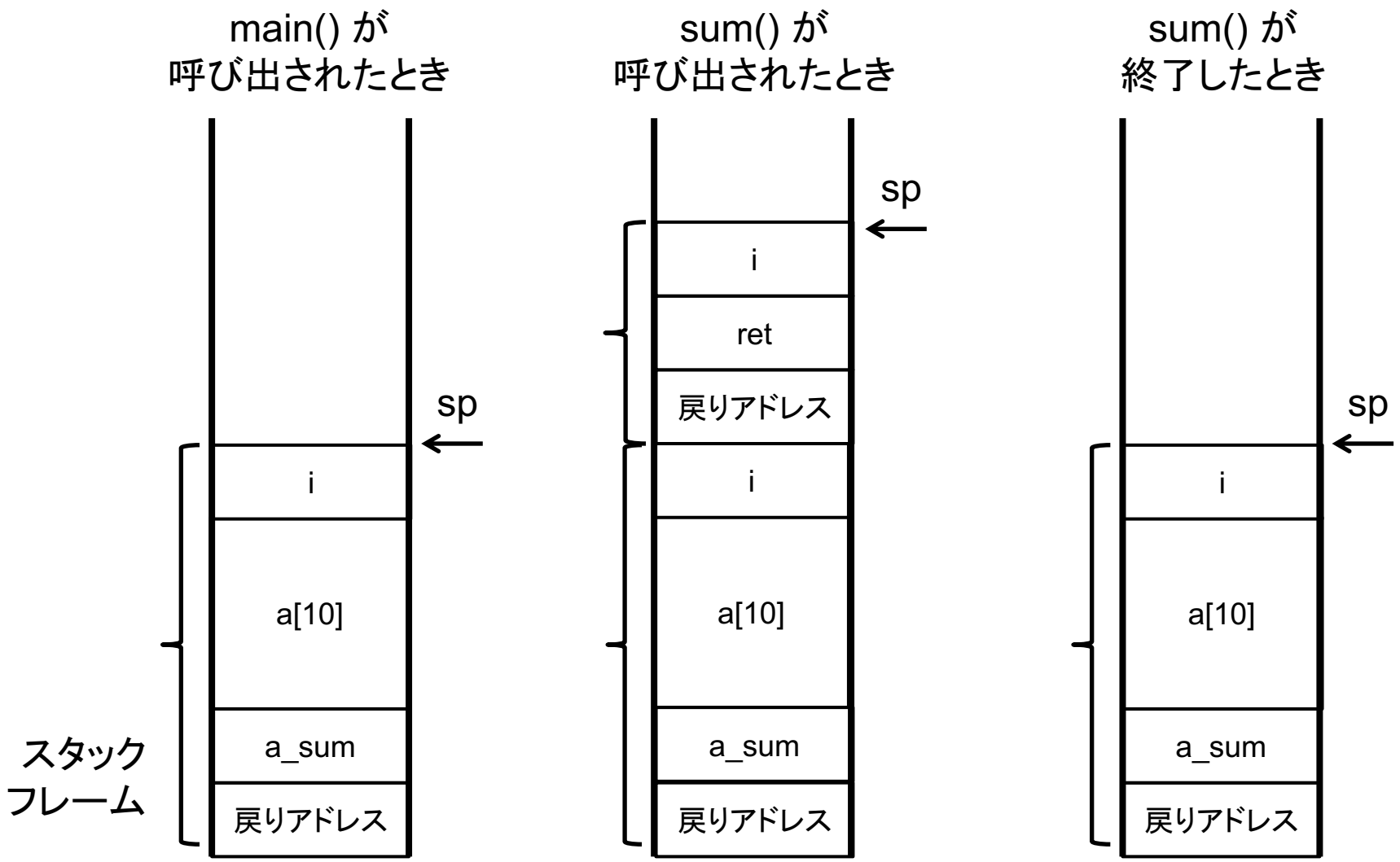
- 課題1～4 に取り組んでレポートにまとめる
- 締切: 11/21(水) 0:00 （11/20(火) 24:00）
- 提出先: レポート投函システム
  - PDF 形式で提出してください
  - ファイル名は“学籍番号 8 桁数字_名字(ローマ字)”
    - 例: 10268039 の寺田くん => 10268039_terada

レポートについて(2/2)

- 課題の解答だけを書いてください
  - 問題文は書かなくてよいです
  - 実行結果だけでもダメ
    - 意図通りなのか，そうでないのか，etc.
- 原理は不要です
- 無駄な考察は不要です
  - 考察が必要な部分は「XXXを考察せよ」と明記してあります

補足

スタックの状態


main() が
呼び出されたとき
sum() が
呼び出されたとき
sum() が
終了したとき
sp
i
a[10]
a_sum
戻りアドレス
ret
スタック
フレーム