

# システムプログラム 第2回

---

創域理工学部 情報計算科学科

松澤 智史

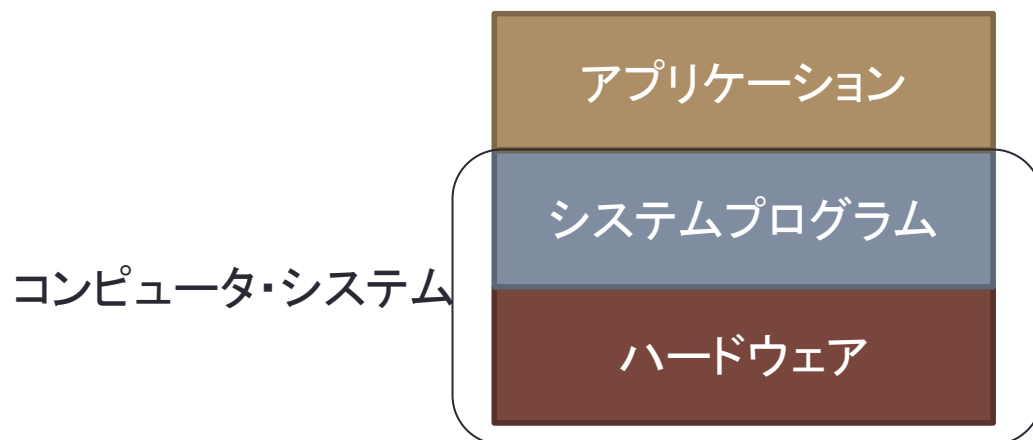
# コンピュータ・システム

## コンピュータ・システム

ハードウェアとシステムプログラムが連携して  
アプリケーションを走らせるように構成したもの

アプリケーションプログラムのコードを作成するところから  
実際に動作するまでの一連の流れを追う

今回の概要



# ソースコードと機械語

ソースコード

```
#include <stdio.h>

int main(){
    printf("Hello");
}
```

人が理解しやすい言語



翻訳  
(コンパイル)

機械語

```
01010001011101001001
00101000000101111010
01011110001111010010
1000111111111101001
01000011011000000001
00101001011111001111
00000101000100000100
10110001000100000010
01000000100010011100
```

CPUが実行できる言語



# ソースコード(ソースファイル)

- 機械語に一意に翻訳可能な文法を維持しつつ, 可能な限り自然言語に近い形式のテキストファイル(プログラム)
- 自然言語的な表現であるが, ソースコードも01のビット列であり, 8ビットの塊(1バイト)を1文字としている

[illegible]

# ASCIIコード(アスキーコード)

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

```
tusedls08$ cat hello.c
#include <stdio.h>

int main(){
    printf("Hello, world\n");
    return 0;
}
tusedls08$ hexdump hello.c
00000000 6923 636e 756c 6564 3c20 7473 6964 2e6f
00000100 3e68 0a0a 6e69 2074 616d 6e69 2928 0a7b
00000200 2020 7270 6e69 6674 2228 6548 6c6c 2c6f
00000300 7720 726f 646c 6e5c 2922 0a3b 2020 6572
00000400 7574 6e72 3020 0a3b 0a7d
000004a0
tusedls08$
```

# ソースコードと機械語 その2

## ソースコード

```
6923 636e 756c 6564
3c20 7473 6964 2e6f
3e68 0a0a 6e69 2074
616d 6e69 2928 0a7b
2020 7270 6e69 6674
2228 6548 6c6c 2c6f
7720 726f 646c 6e5c
2922 0a3b 2020 6572
7574 6e72 3020 0a3b
```

人が理解しやすい？数値列



翻訳  
(コンパイル)

## 機械語

```
01010001011101001001
00101000000101111010
01011110001111010010
1000111111111101001
0100011011000000001
00101001011111001111
00000101000100000100
10110001000100000010
01000000100010011100
```

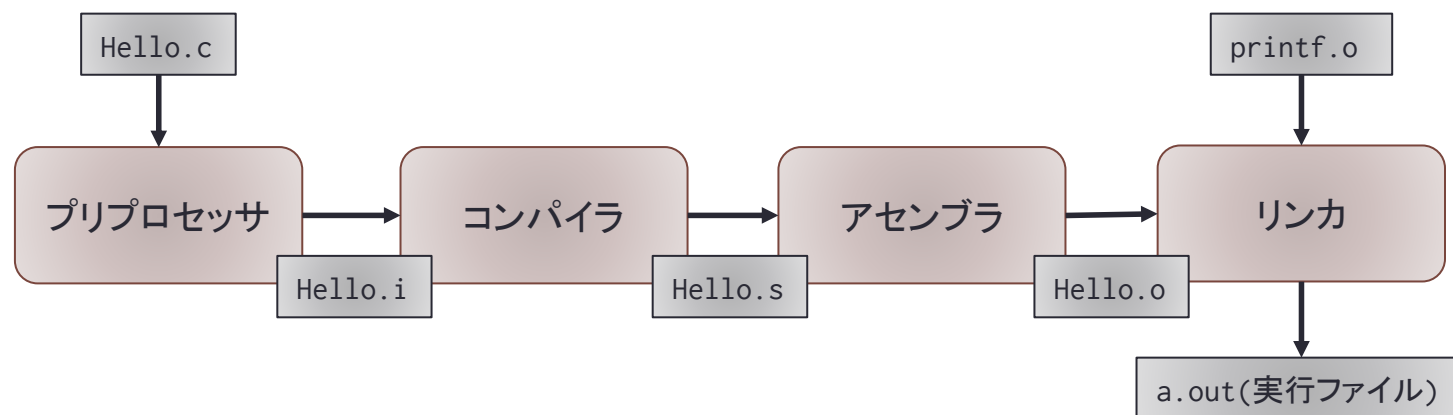
CPUが実行できる数値列



単なる数値の変換

# 翻訳(コンパイル)

- 翻訳プログラム群によって機械語に変換する
- C言語の例(gcc)では, 以下の4つの手順を行う



# プリプロセッサ

- #で始まるディレクティブに従って改変
  - #define マクロの定義
  - #include ヘッダファイルの読み込み
  - #pragma ホストマシンやオペレーティングシステムに固有の機能
  - など
- gccでプリプロセスのみ行う場合は -E オプションを付与

```
tusedls08$ cat test.c
#define ONE 1
int main(){
ONE+1;
}
tusedls08$ gcc -E test.c
# 1 "test.c"
# 1 "<組み込み>"
# 1 "<コマンドライン>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<コマンドライン>" 2
# 1 "test.c"

int main(){
1 +1;
}
tusedls08$
```

```
tusedls08$ gcc -E hello.c |cat -s |less
# 1 "hello.c"
# 1 "<組み込み>"
# 1 "<コマンドライン>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<コマンドライン>" 2
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 375 "/usr/include/features.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 392 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 393 "/usr/include/sys/cdefs.h" 2 3 4
# 376 "/usr/include/features.h" 2 3 4
# 399 "/usr/include/features.h" 3 4
# 1 "/usr/include/gnu/stubs.h" 1 3 4
...skipping...
# 1 "hello.c"
# 1 "<組み込み>"
```

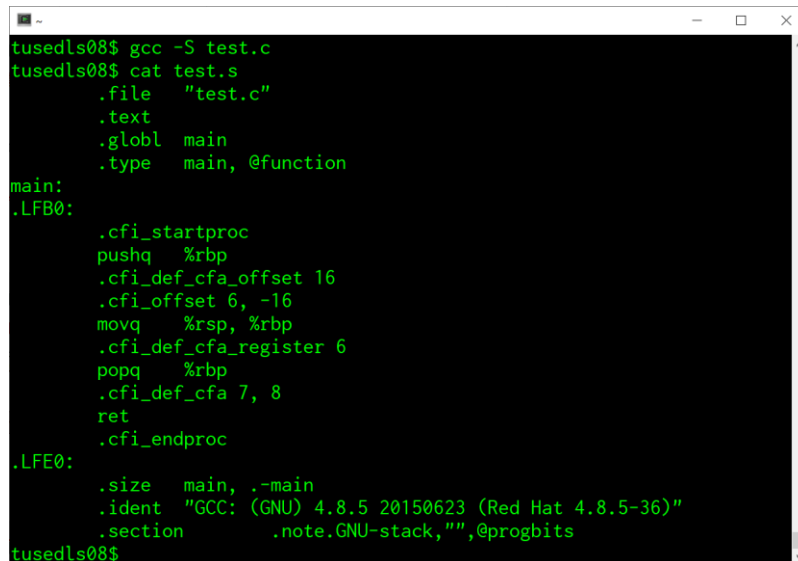


# コンパイラ

- プリプロセッサによって出力されたテキストファイル `***.i` をアセンブリ言語プログラムのテキストファイル `***.s` に翻訳する

## アセンブリ言語

- 機械語命令をテキスト形式で記述したもの
- CPUのロード・ストア・演算・ジャンプなどの命令をテキスト化



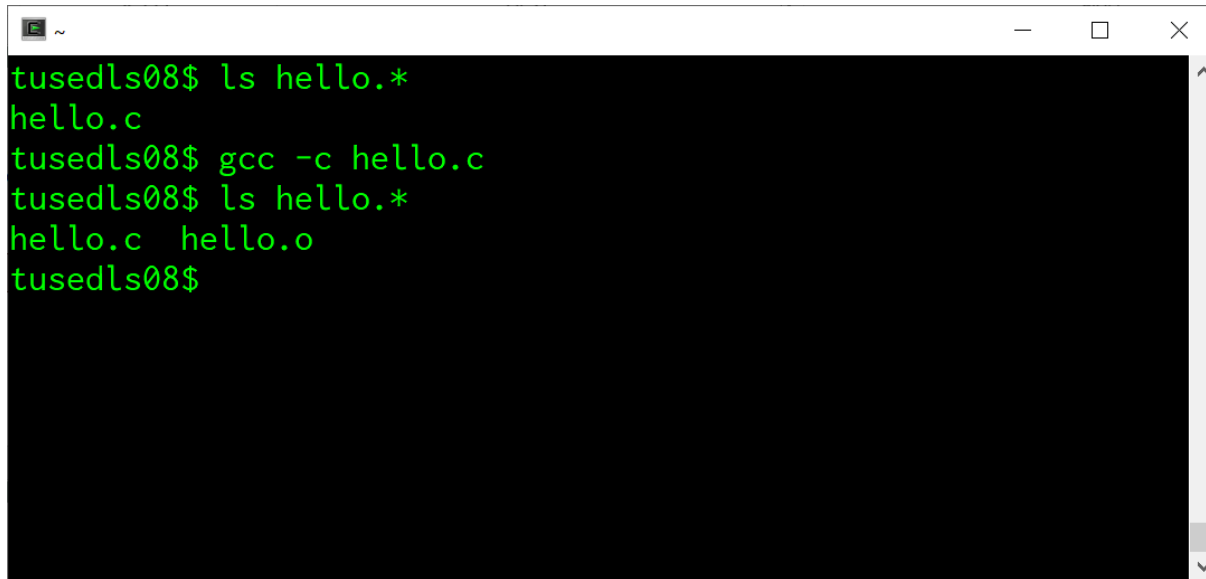
```
tusedls08$ gcc -S test.c
tusedls08$ cat test.s
.file "test.c"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-36)"
.section .note.GNU-stack,"",@progbits
tusedls08$
```

メモ

\$ gcc -S でアセンブリ言語記述の  
ファイル生成まで行える

# アセンブラ

- `***.s` を機械語命令に翻訳し, リロケータブル・オブジェクト・プログラムと呼ばれる形にパッケージ化する
- 出力されるファイルは `***.o`

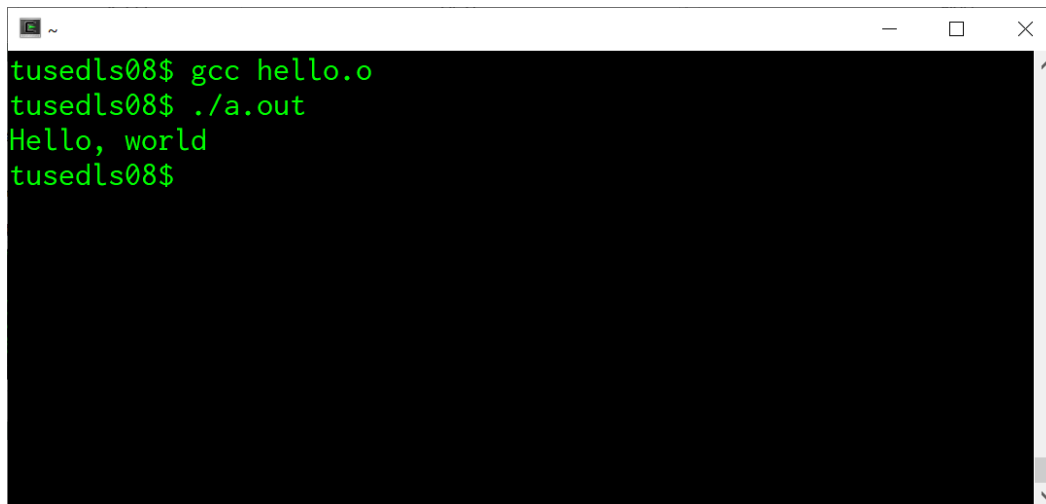
A terminal window with a black background and green text. The window title is '~'. The text inside shows a series of commands and their outputs: 'tusedls08\$ ls hello.\*' followed by 'hello.c', then 'tusedls08\$ gcc -c hello.c', and finally 'tusedls08\$ ls hello.\*' followed by 'hello.c hello.o'. The prompt 'tusedls08\$' is shown at the end.

```
tusedls08$ ls hello.*
hello.c
tusedls08$ gcc -c hello.c
tusedls08$ ls hello.*
hello.c hello.o
tusedls08$
```

※ `hello.o` の中身はもう人が読める形式(ASCII)ではない

# リンカ

- `***.o` に標準ライブラリのオブジェクトファイル等を結合させて実行可能なオブジェクトファイル(実行可能ファイル)にする
  - 今回の場合は`printf.o`
  - ライブラリに関しては前回のスライド参考
- `main`関数のあるオブジェクトファイルから芋づる式に保留された関数の実装(未解決の外部参照)を探索



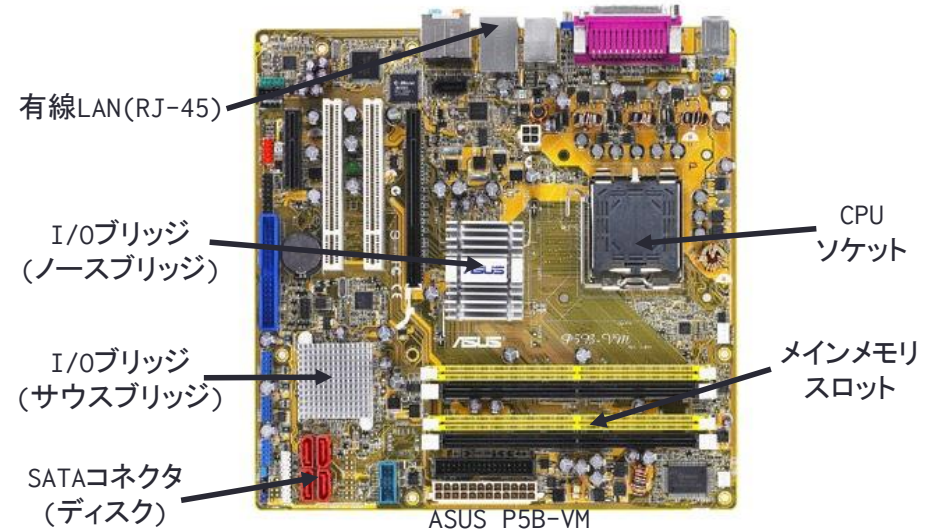
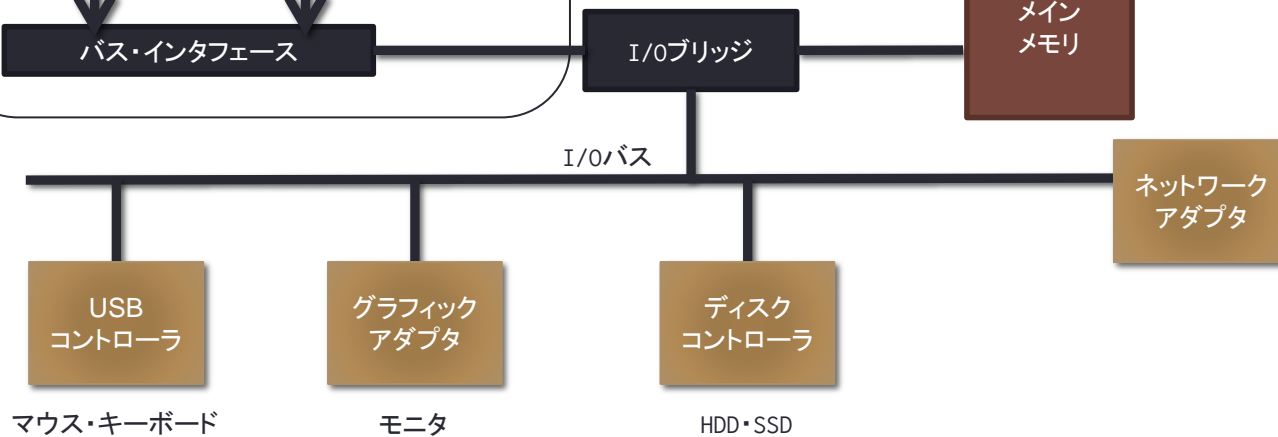
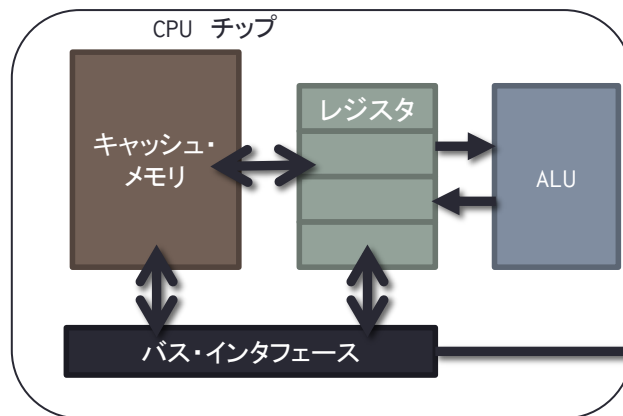
```
tusedls08$ gcc hello.o
tusedls08$ ./a.out
Hello, world
tusedls08$
```

# 余談 gccのオプションまとめ

- `gcc hello.c` → リンカまでまとめて行う(実行ファイル生成)  
`gcc hello.c -o hello` 実行ファイル名をhelloにして生成
- `gcc -E hello.c` → プリプロセスのみ行った結果を標準出力へ
- `gcc -S hello.c` → アセンブリ言語へ翻訳(hello.s生成)  
`gcc -fno-asynchronous-unwind-tables -S hello.c` デバッグ情報なしファイル生成
- `gcc -c hello.c` → 機械語命令に翻訳(hello.o生成)

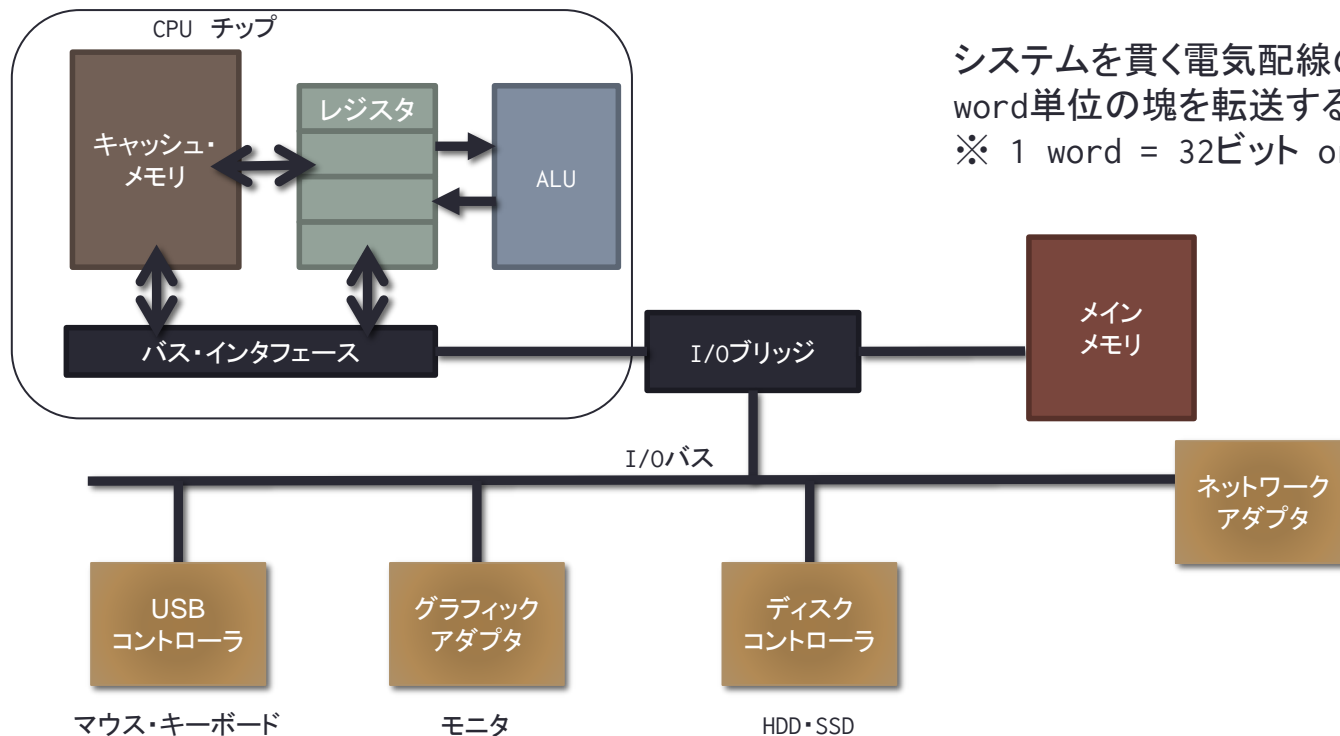
# プログラムの実行

## システムのハードウェア構成



# プログラムの実行

## システムのハードウェア構成

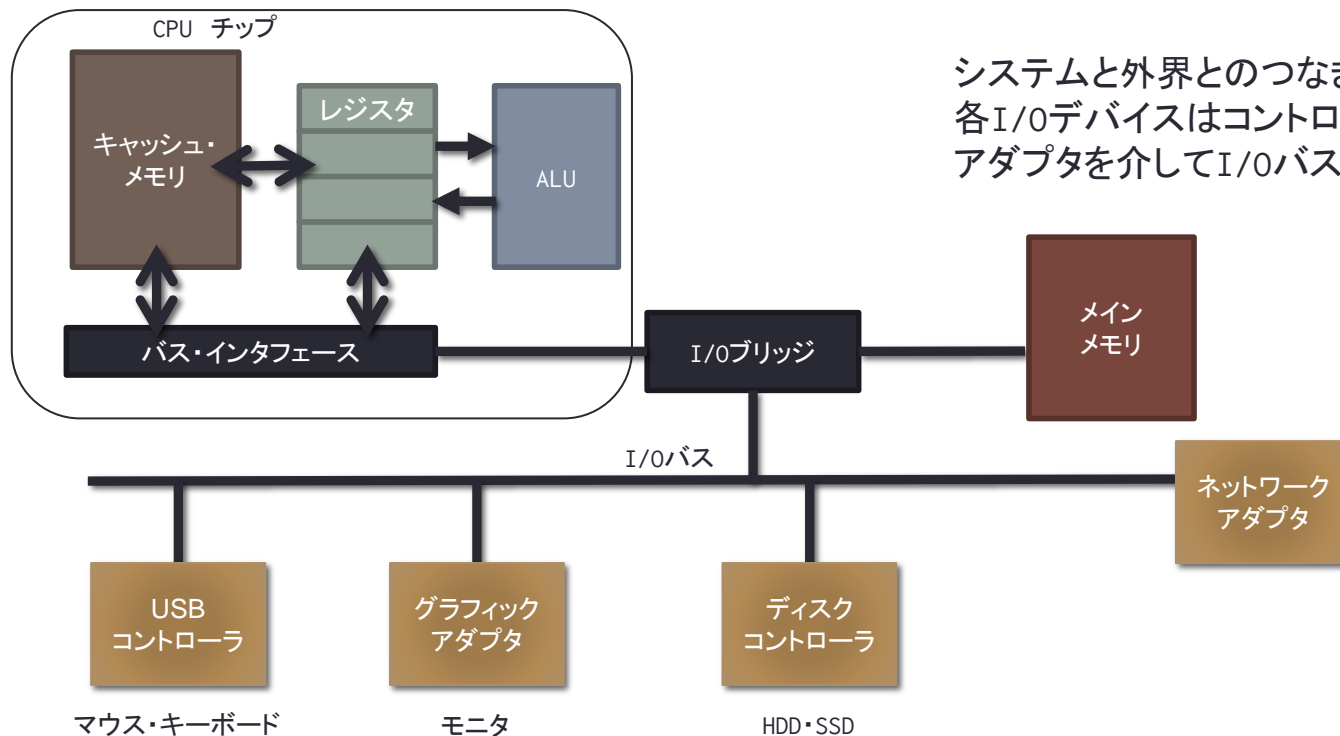


### バス

システムを貫く電気配線の集まり  
word単位の塊を転送する  
※ 1 word = 32ビット or 64ビット

# プログラムの実行

## システムのハードウェア構成

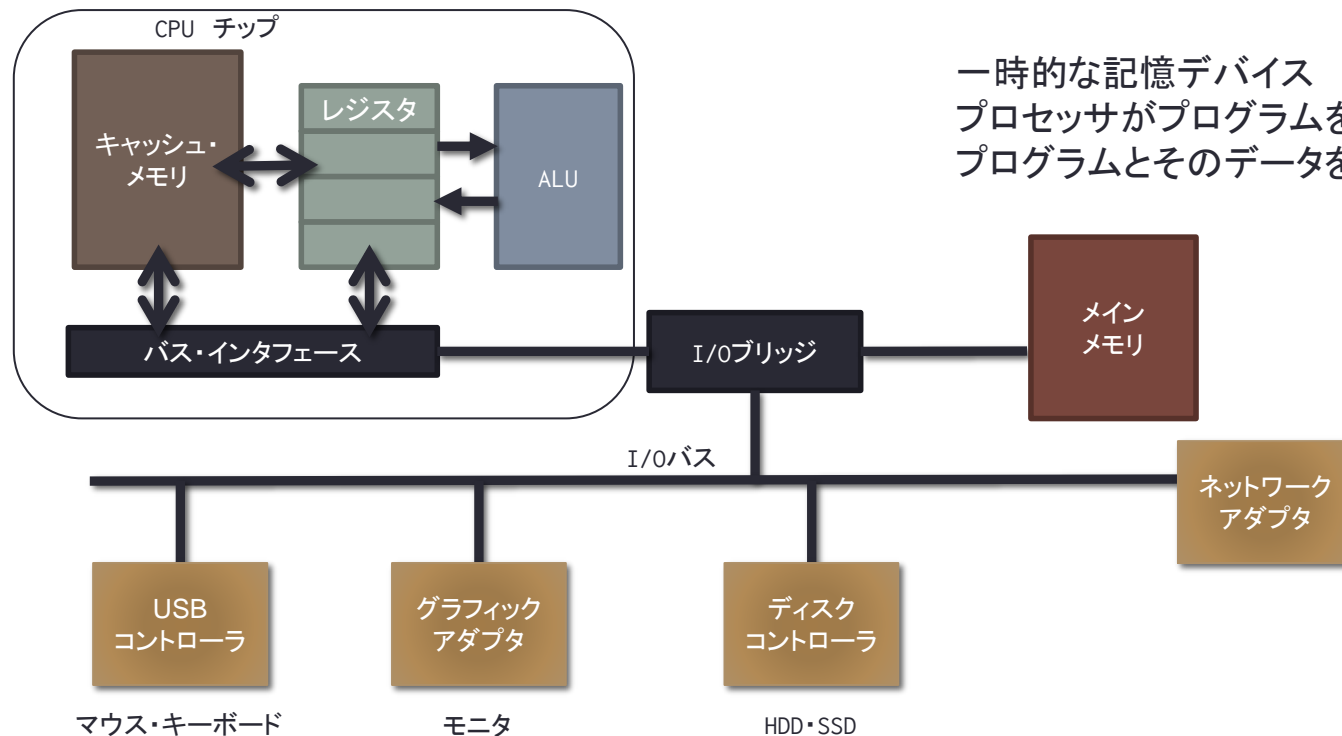


### I/O(入力/出力)デバイス

システムと外界とのつなぎ目  
各I/Oデバイスはコントローラまたは  
アダプタを介してI/Oバスに接続

# プログラムの実行

## システムのハードウェア構成



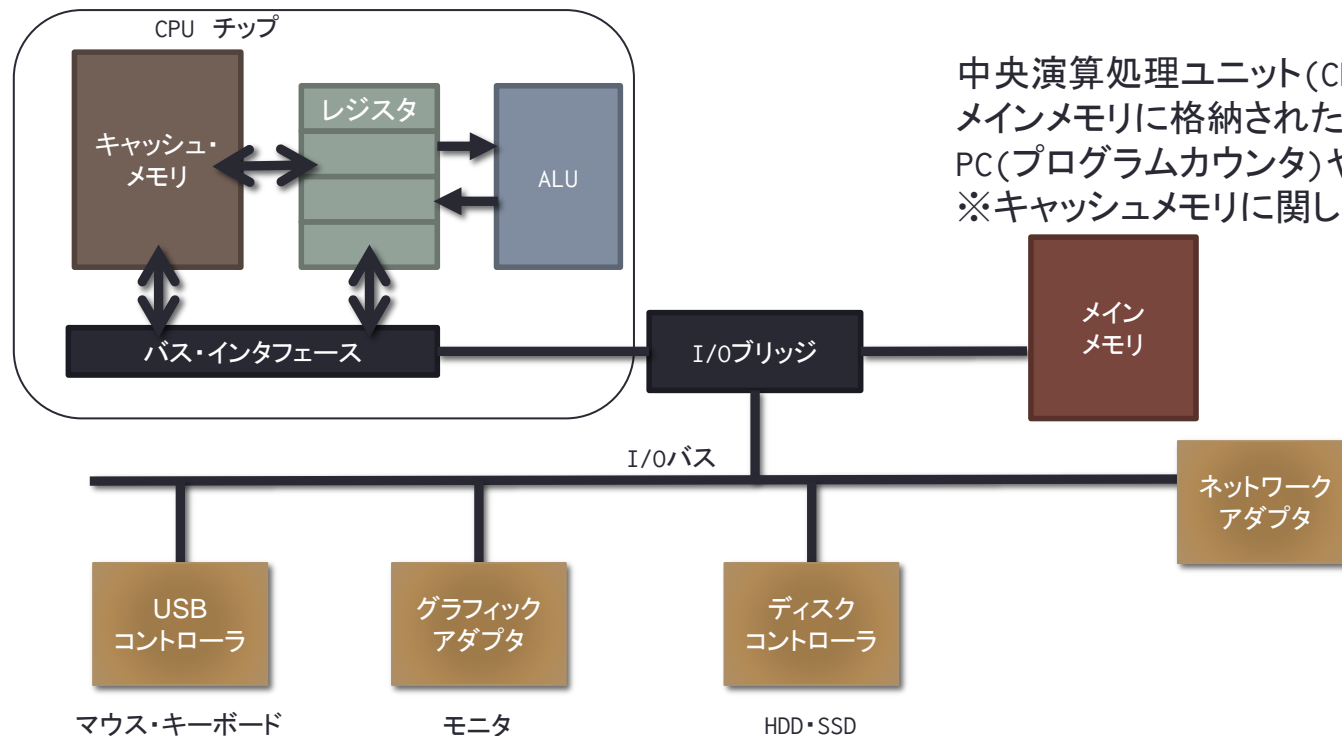
### メインメモリ

一時的な記憶デバイス  
プロセッサがプログラムを実行する間、  
プログラムとそのデータを格納する



# プログラムの実行

## システムのハードウェア構成

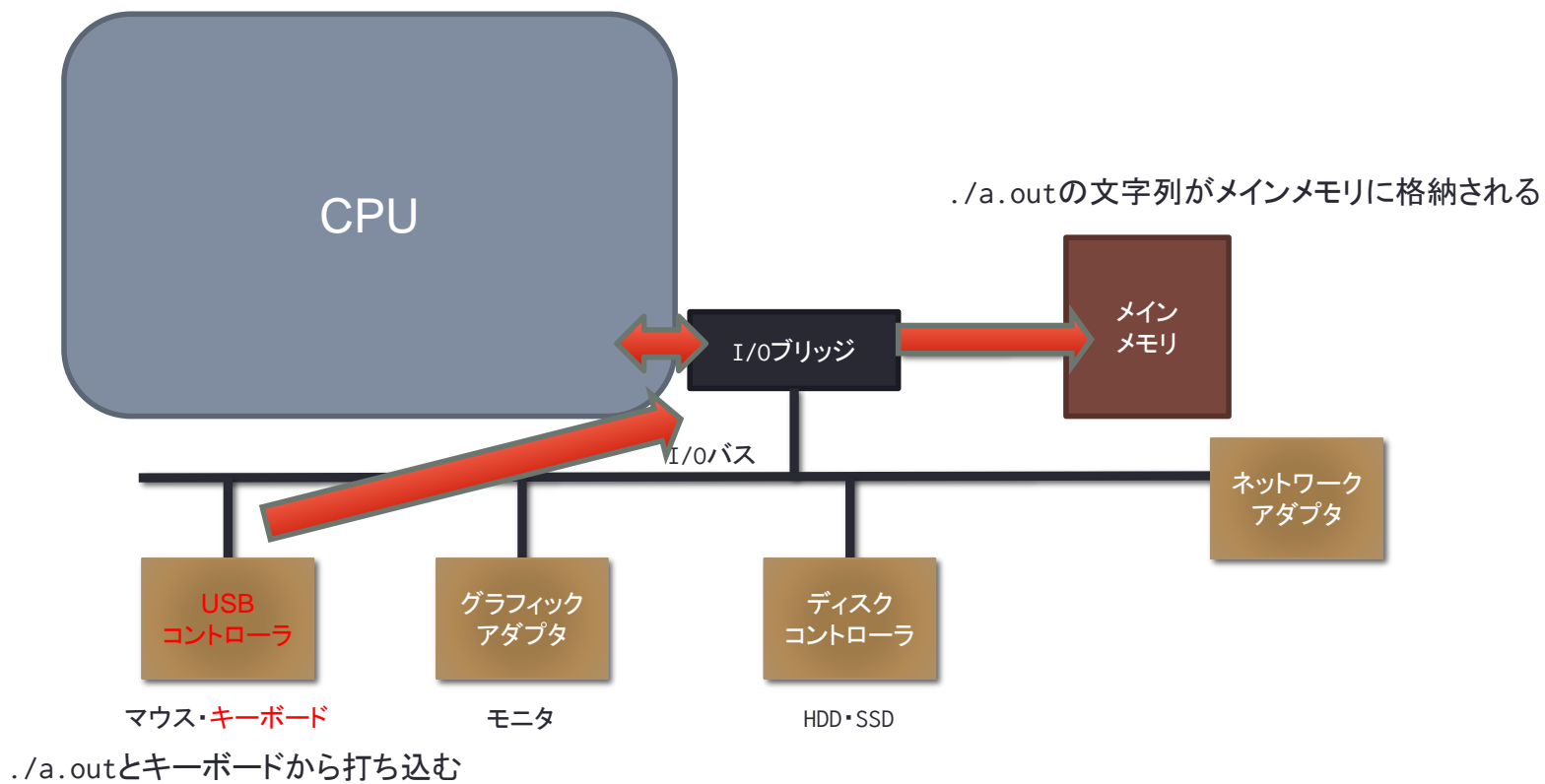


### プロセッサ

中央演算処理ユニット(CPU)とも呼ぶ  
メインメモリに格納された命令を解釈(実行)する  
PC(プログラムカウンタ)や汎用のレジスタを持つ  
※キャッシュメモリに関しては後述

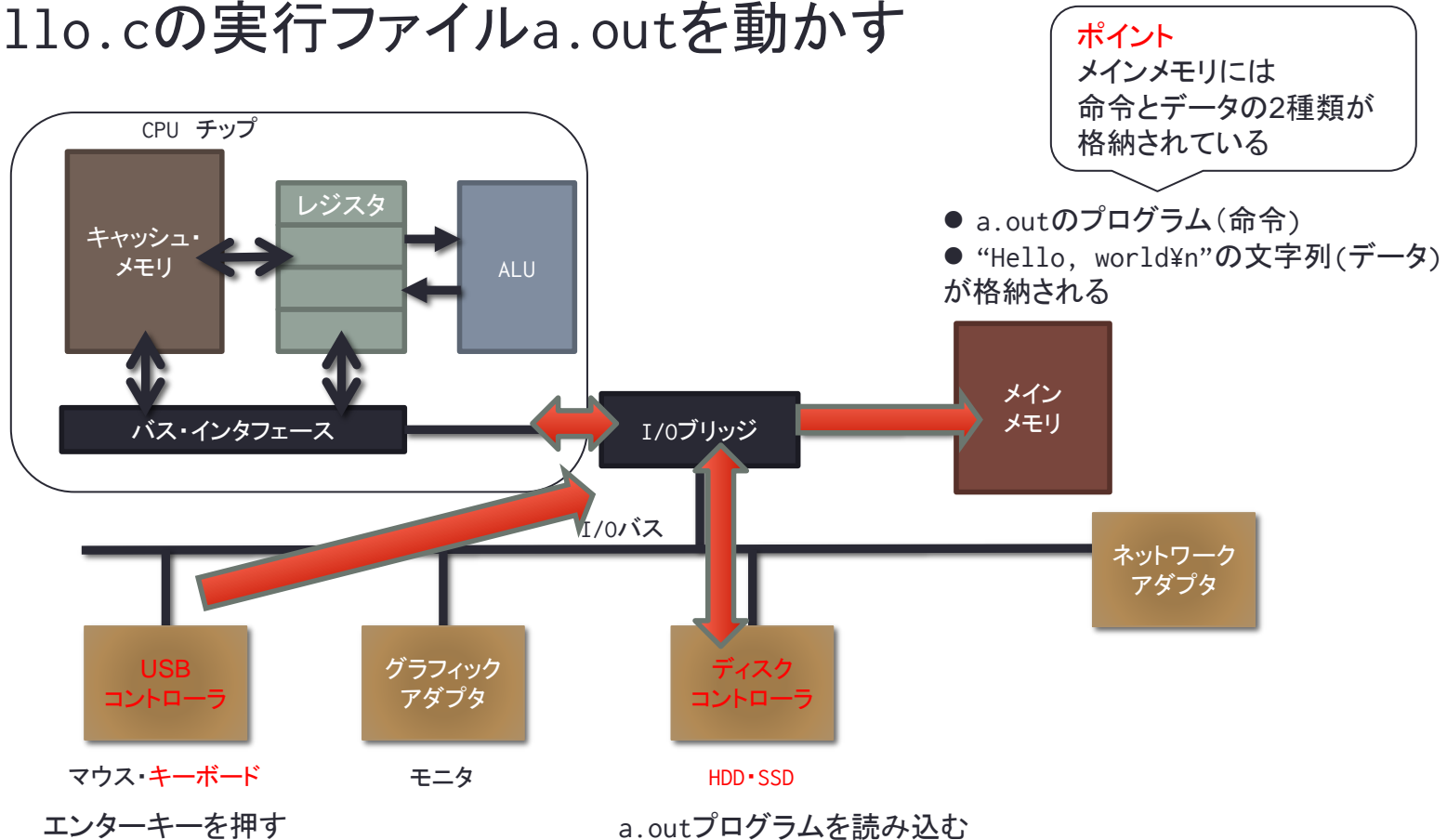
# プログラムの実行

## hello.cの実行ファイルa.outを動かす



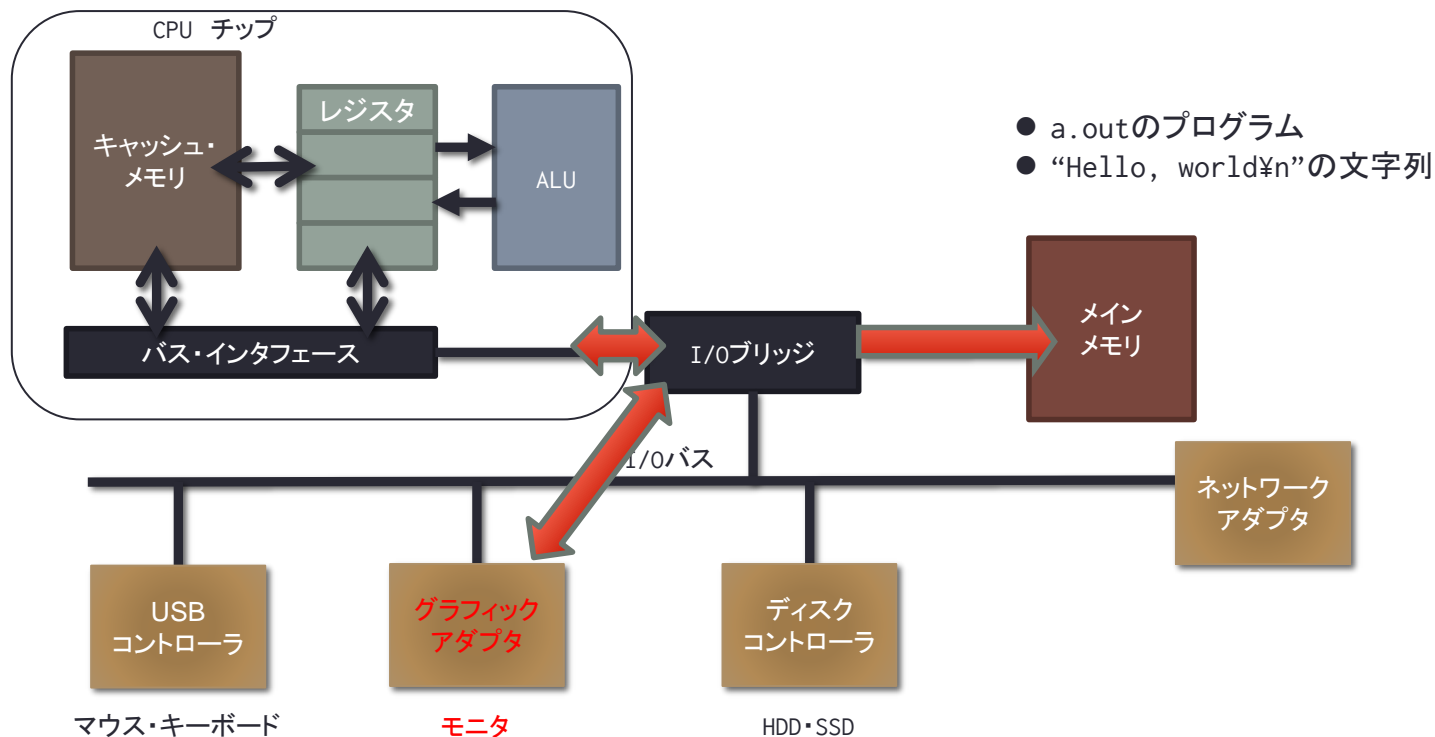
# プログラムの実行

hello.cの実行ファイルa.outを動かす



# プログラムの実行

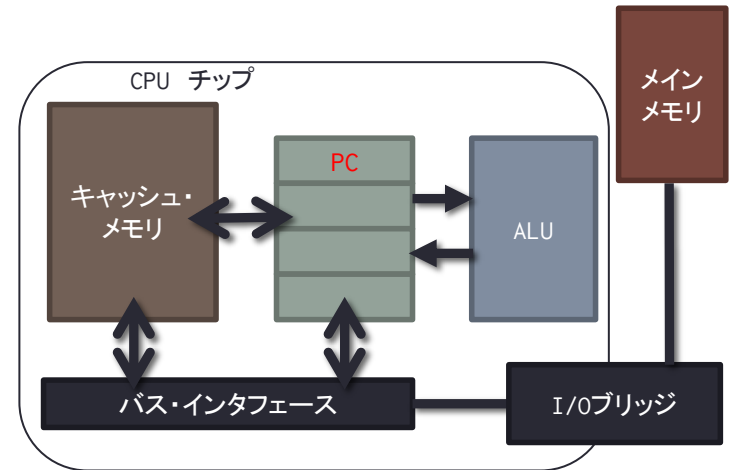
hello.cの実行ファイルa.outを動かす



Hello worldがモニタに出力される

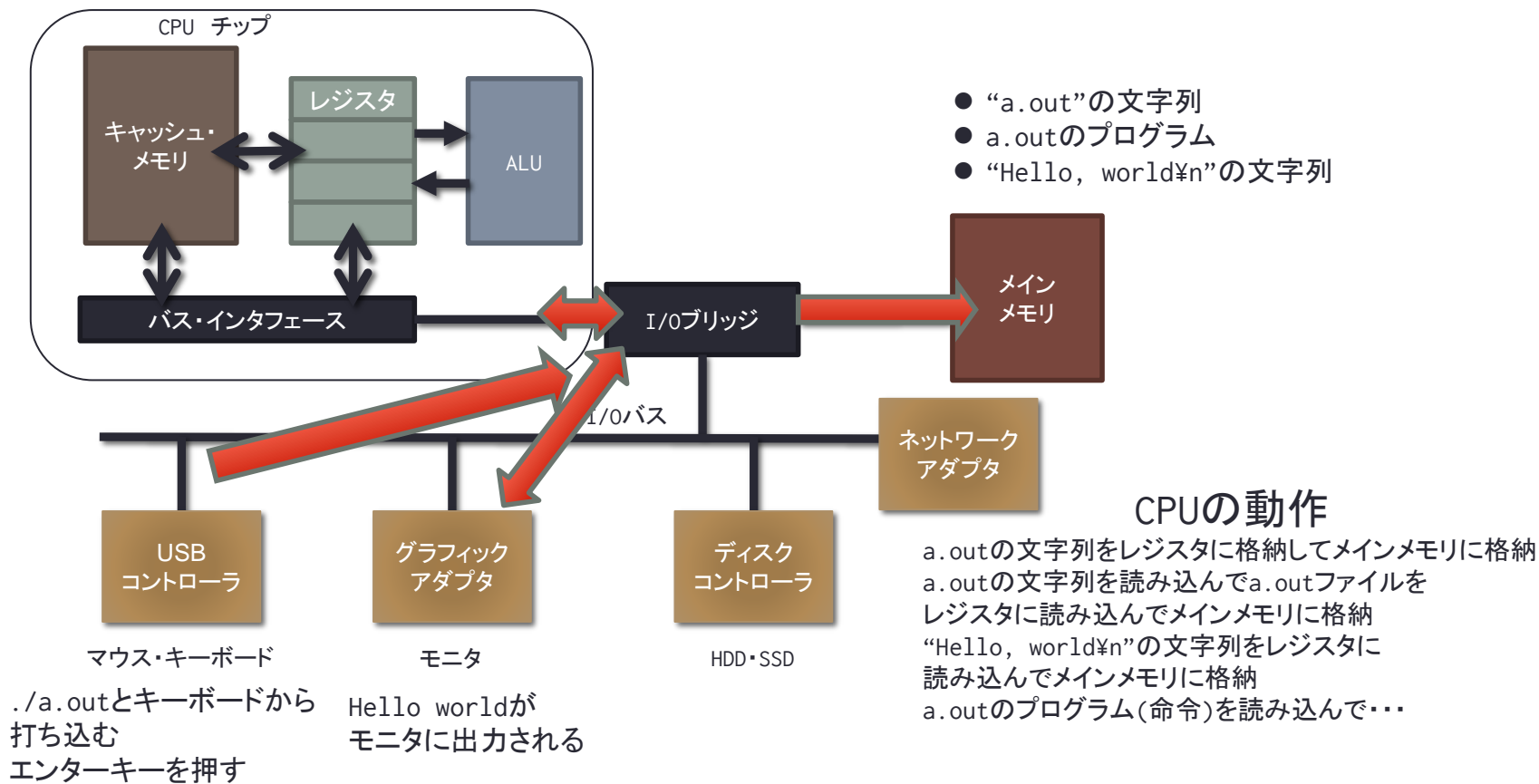
# プロセッサの動作

- レジスタの一つのPC(プログラムカウンタ)にメインメモリ内にある命令のアドレスを保持
  - 次に実行する命令(アドレス)をPCに置き, 次の命令を指すようにPCを更新する
- 命令セットアーキテクチャ(CPUにより異なる)によって定義されたシンプルな命令を実行可能
  - ロード メインメモリからレジスタにコピー
  - ストア レジスタからメインメモリの指定された場所にコピー
  - 演算 2つのレジスタの内容をALUにコピーし, 算術演算の結果をレジスタに上書き
  - ジャンプ 指定した値(アドレス)をPCにコピー
- 現在のプロセッサは高速動作させるためにいくつかの複雑なメカニズムを実装している
  - 追加の命令セット
  - メモリとファイルの抽象化(仮想メモリ等)
  - DMA(ダイレクト・メモリ・アクセス)・・厳密にはプロセッサの機能ではない



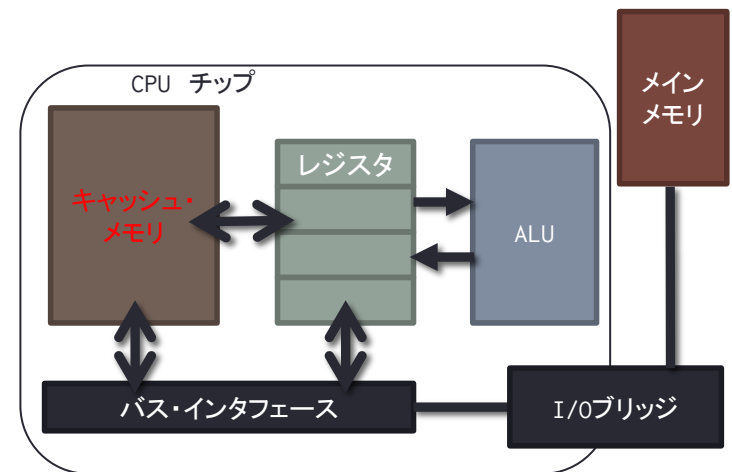
# 再度:プログラムの実行

hello.cの実行ファイルa.outを動かす

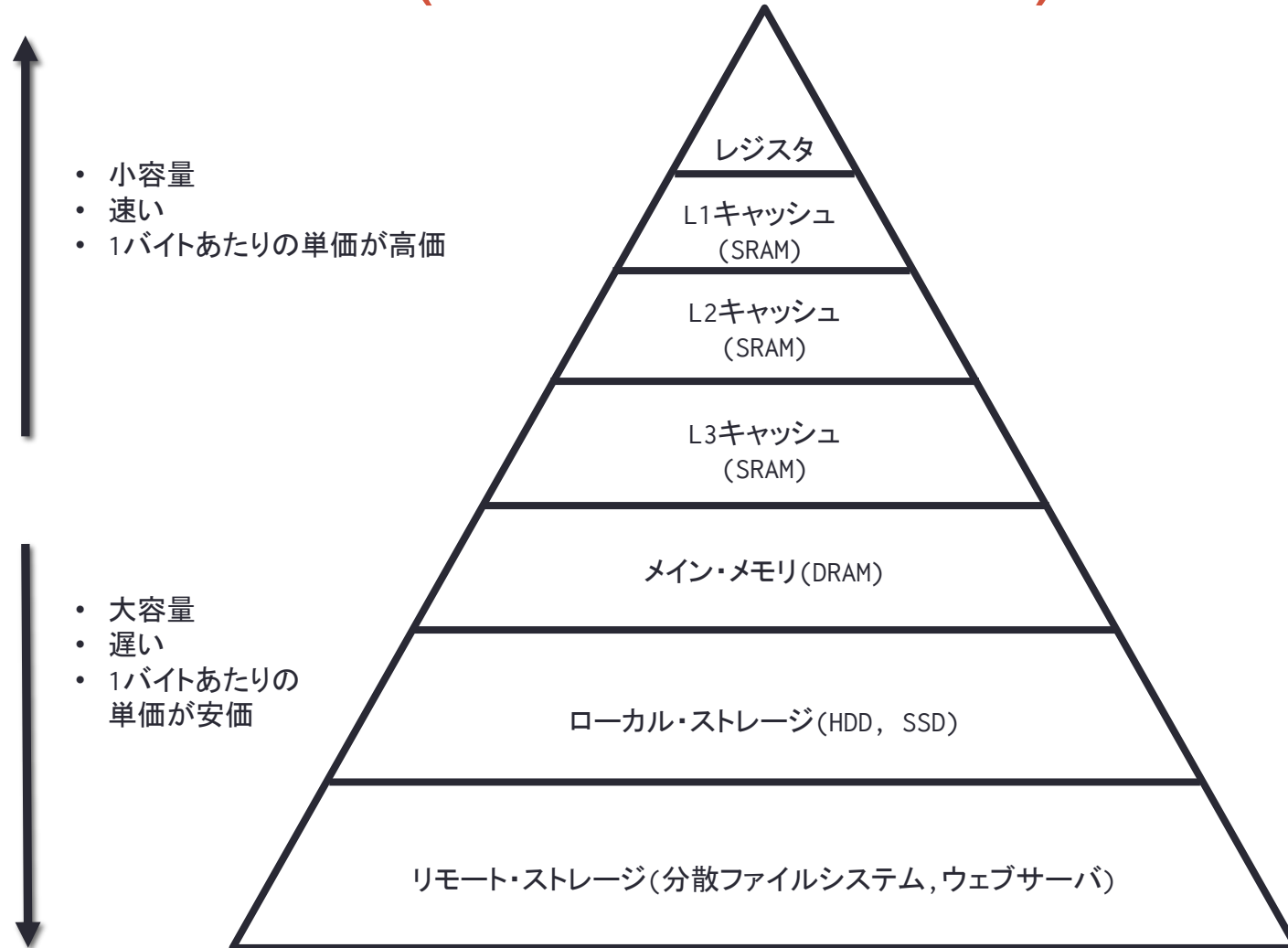


# キャッシュ・メモリ

- 情報のある場所から別の場所に移す処理が多い
  - ディスクからレジスタを経由してメインメモリへ
  - メインメモリからレジスタへ
  - など
- レジスタの容量は64ビット×数十本であるが、メインメモリより100倍以上速く読み書きできる
  - メインメモリへの読み書きがネックとなる
  - プロセッサ-メモリギャップという  
年々加速中
- CPUチップ内にメインメモリより高速な記憶媒体を用意する(キャッシュメモリ)

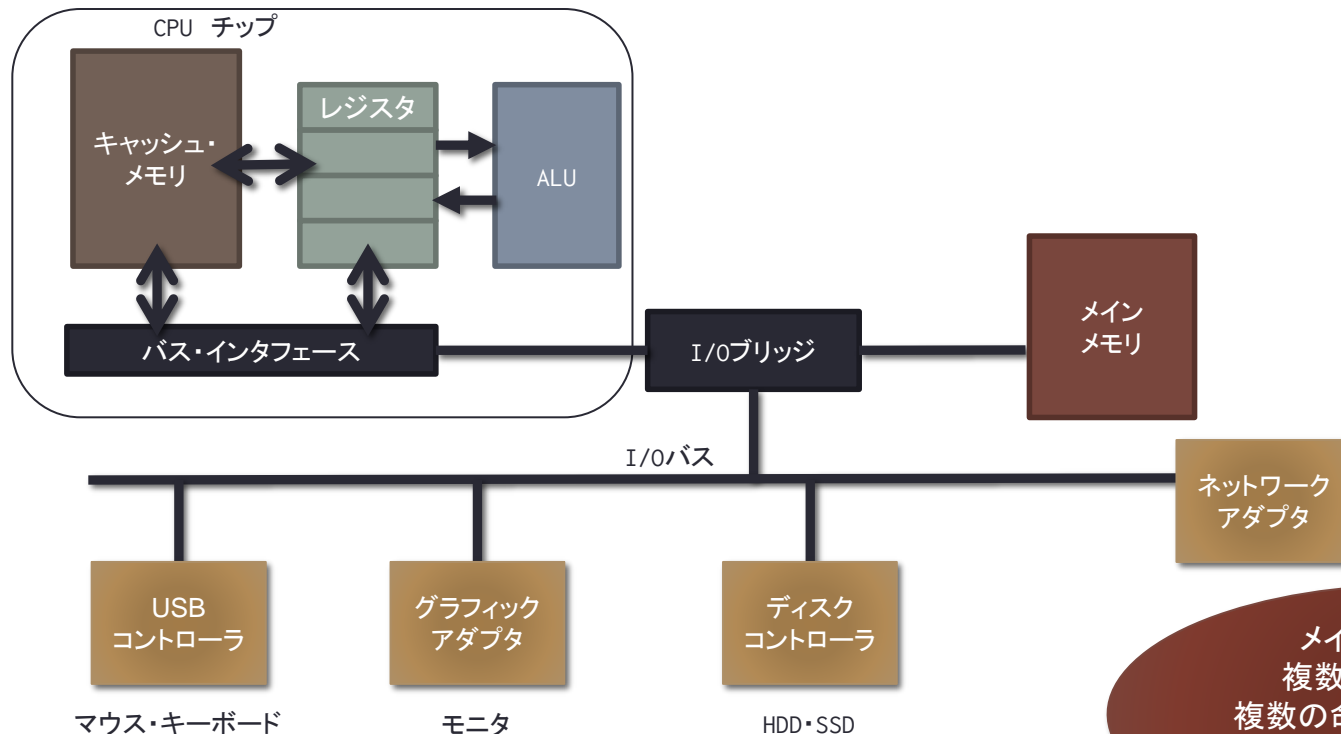


# ストレージ(メモリ, ディスク)階層





# 再再度:プログラムの実行



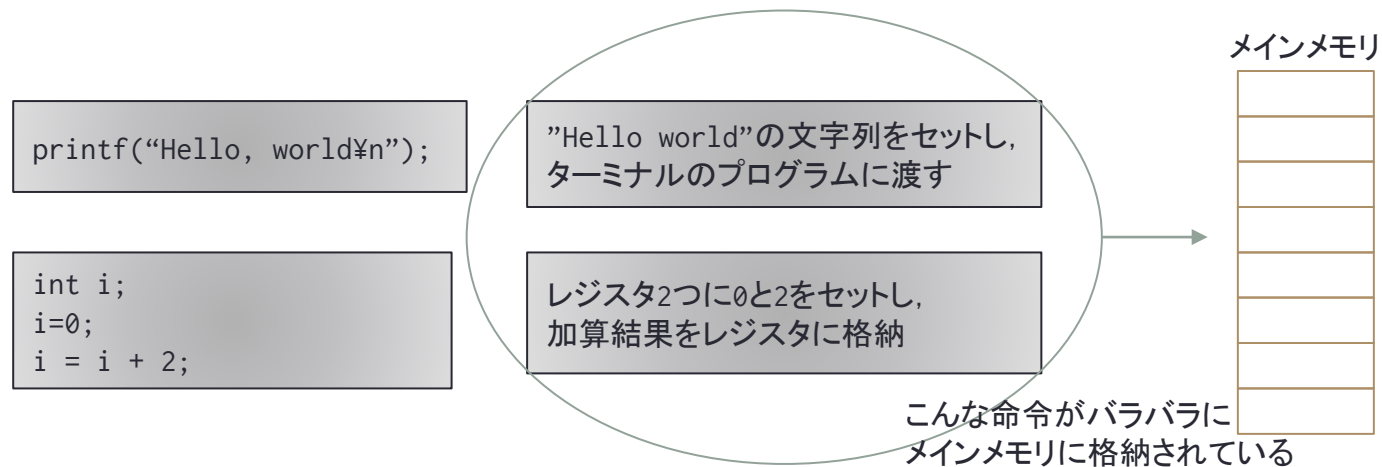
メインメモリには、  
複数のプログラムの  
複数の命令が存在している  
また、必ずしも順番に  
並んでいるとは限らない

## hello.cが動くまで(かなり省略)

- ディスクにあるOSのプログラム(+必要なデータ)がメインメモリに格納される→OS動作中
- キーボードやマウスからの入力はOSのプログラムによって検知され、CPUやメインメモリに送付
- shell(bash等)のプログラムも格納され動作中
- emacsの入力後、shellがコマンドの解釈をし、OSのライブラリ等を用いてemacsのプログラムをロードしてメインメモリへ送付
- OS(ライブラリやシステムコール)やWindowManagerの機能を利用してemacsのエディタ画面をモニタに描画
- キーボードやマウスからemacs上で保存を選んだ際にはOSの機能を経由してemacsプログラムにその信号を送付
- emacsプログラム内の保存機能(ファイル保存はOSのシステムコールを使用)を利用してディスクに保存
- gccの入力後、shellがコマンドを解釈・・・(上と同じ) して、a.outを生成してディスクへ保存(保存の方法も上と同じ)・・・

# プロセス

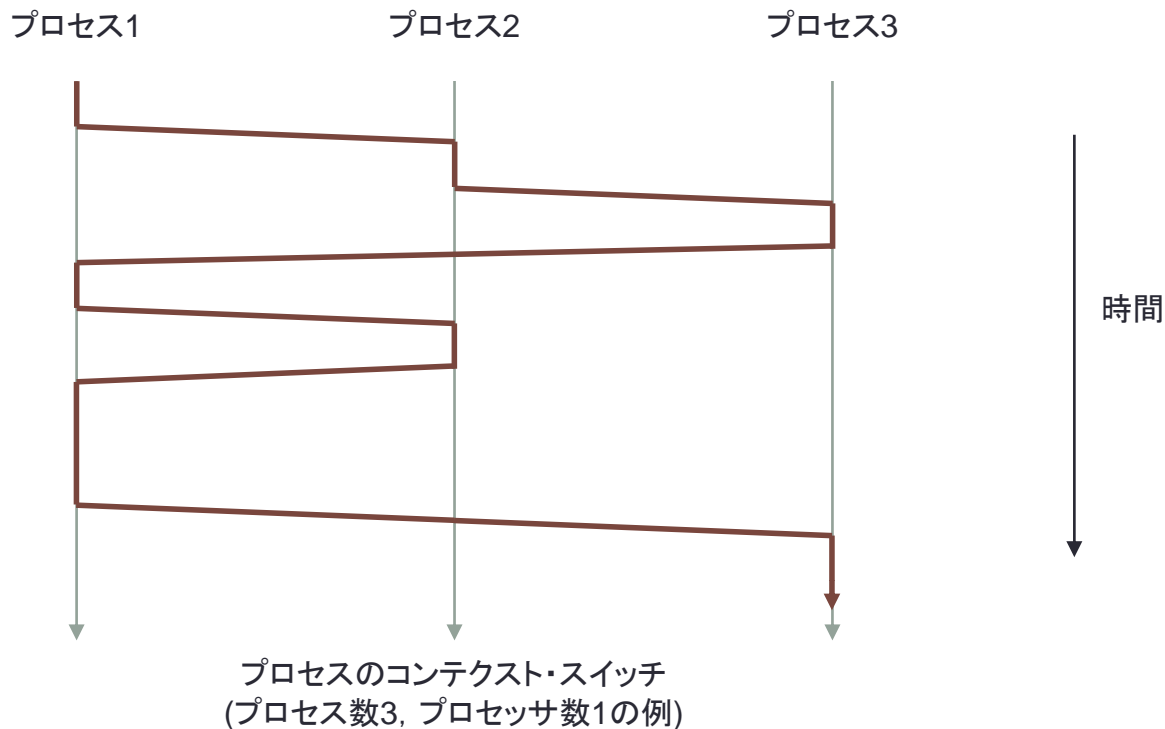
- メイン・メモリには, 複数のプログラムの命令が存在する
- CPUは簡潔な命令を1つずつ実行する



- 1つのプログラムの実行を1プロセスとして管理するOS内のプログラム
  - 各プロセスにはプロセスIDという識別子を付与
- 状況に応じてどの命令をCPUに実行させるかを判断するプログラム

# プロセス

- 実行中のプログラムをOS(カーネル)が抽象化したもの
- 各プロセスはハードウェアを独占的に使用しているように見える
- 多くのシステムでは「プロセスの数>実行するプロセッサの数」
- コンテキストスイッチにより並行動作  
(あたかも独占的に動作しているように見せる)



# コンピュータ・システムの抽象化

- プロセスは実行中のプログラムをOSが抽象化したもの
- 抽象化の概念はOSの至る所で使用されている
  - 仮想メモリ
    - 物理メモリとは異なる仮想アドレス空間で管理
  - ファイル
    - 異なる仕様のディスク等でも同一の操作
- OS以外での抽象化
  - プログラミング言語
  - API

# まとめ

- コンピュータシステムはハードウェアとシステムソフトウェアからなり、それらが連携してアプリケーションソフトウェアを動作させる
- プログラムは別のプログラムによって別の形に翻訳される
  - ASCIIテキストとして書かれた後、コンパイラやリンカによりバイナリの実行可能ファイルに変換される
- プロセッサはメインメモリに格納された命令を読みだして解釈する
  - メインメモリとI/Oデバイスとレジスタ間でデータコピーが頻繁に行われる
  - コピーや読み出しを高速に行うためにキャッシュメモリが用意される
- OSのカーネルは、アプリケーションとハードウェアの仲介者として働き、3つの抽象化を提供する
  - プロセス
  - ファイル
  - 仮想メモリ

質問あればどうぞ