

コンピュータ アーキテクチャ I 第2回

塩谷 亮太 (shioya@ci.i.u-tokyo.ac.jp)

東京大学大学院情報理工学系研究科 創造情報学専攻

質問や感想など

- 1年生の時にC言語と論理回路は習っているので、その前提で進めていただいて大丈夫です。論理回路は全加算期くらいまで習った気がしますが、正直記憶はあやふやです...

- 「CやPythonなどで書かれたソフトはどのように翻訳されて実行されるのか」という疑問は自分では持ったことがなかったのですが、とても興味深いなと思いました。

質問や感想など

- C言語を学び始めて配列・ポインタを勉強した時、コンピュータのメモリの原理を勉強したら処理する動きが理解しやすくなったので、プログラミング（特に低級言語）をする上でコンピュータアーキテクチャの理解は必要だなと感じました。

質問や感想など

- 暗記は得意ではないので授業内容を理解してついていけるのか不安ですが、なんとか毎時間ついていけるように頑張りたいと思います。

質問や感想など

- 将来やりたいことがまだ決まっていないので、アーキテクチャに関する職業も個人的に調べたいです。これからよろしくお願いいたします。

質問や感想など

- 履修登録の段階ではコンピュータアーキテクチャが何なのか全く分かっていませんでしたが、コンピュータにおける設計や様式が何なのか、なぜ学ぶのかが何となく分かりました。

質問や感想など

- 講義内で出てきた組み込みコンピュータは、小さいので高そうですが、目的が極めて限定されているため安くすんでいる、ということなのではないでしょうか。

- 広義のコンピュータアーキテクチャの範囲に、アプリケーション・ソフトウェアと集積回路/半導体デバイスが含まれないのはなぜなのか気になりました。

- 組み込みコンピュータは基本的に1つの機器に1つのコンピュータが搭載されているのでしょうか。また、車には小さなコンピュータがたくさんいくつも搭載されているのは、処理すべきことが多く、コンピュータごとに役割を分けているからなののでしょうか。

- 一年生後期の授業で、サーバーやデータベースなどとやり取りをする、ミドルウェアについて軽く触れました。ミドルウェアはコンピュータの一部分なののでしょうか。また、コンピュータの一部分だとして、ソフトウェアもしくはハードウェアの中にミドルウェアというものがあるわけではなく、完全に独立したものなののでしょうか。

- コンピュータアーキテクチャの定義である「設計や様式」というのはハードウェアの形、組み方、機能等のハードウェアを作る際に決めること全てという認識でいいのでしょうか。

- 基盤系は色々なアプリケーション一般で5%高速化できれば良いとありましたが、基盤を作り替えたなら応用系の方でそれに合わせて作り替えないのでしょうか。

質問や感想など

- 行列積の実行時間についての話にとっても興味を持ちました。先日、Google STEPのオンラインコーディングテストに挑戦したのですが、半分のテストケースでは正解になっているのに、もう半分のテストケースでは実行時間制限を超えてしまって、クリアになりませんでした。for文がいくつか回っていたので、今日の行列積の内容を理解できればもしかしたら解消できたのかな？と授業中に思いました。
- ソフトウェアに興味があったので軽くしかハードウェアに触れてこなかったのですが、実際この間コーディング問題を解いている際実行時間エラーで引っかかってしまい困ったのできちんとアーキテクチャを学ぼうと思いました。

- 確かにスマホは消費電力の縛りのためにパソコンよりも動きが遅いような気がしました。スマホに冷却ファンが使えない理由も気になりました。

- 私たちが書いているプログラミングのコードがどの様に翻訳されているかというのは難しくて関係のない話なのだろうと思っていたがどの様に翻訳されるかも知ることができると知ってもっと自分自身の理解を深めることができるようになるので今後の授業が楽しみだと思いました。

- 車に組み込まれているコンピュータと言われて想像したのが、カーナビやエンジンの制御でしたが調べてみるとボディ系制御系情報系と三つに分かれて百数十個も組み込まれているとあり驚きました。最初は排ガス規制法に対処するために小型コンピュータが導入されたようで、環境保護を目的にコンピュータ制御が発達したというのはおもしろいなと思いました。。

- 処理時間を短縮するためにはアルゴリズムを利用することだと思っていたが、アーキテクチャの観点からも処理時間を短縮できるというのは衝撃でした。なぜ実行時間が短縮されたのかとても気になります。順序を入れ替えたことでキャッシュにデータが残っているからそこから呼び出すため実行時間が短縮されたのでしょうか

- スーパーコンピュータとPCは目的とする速さが違うとありましたが、量子コンピュータはスーパーコンピュータ寄りの車でいうダンプカーのようなものなのでしょうか？それともダンプカーとスポーツカーを兼ね備えたような存在なのでしょうか？

質問や感想など

- 行列積の例にあったSIZEはどれくらいに設定しましたか？実行時間が1100秒と20秒とのことで結構な差がついているので気になりました。

- 基礎情報技術者試験の勉強をしているのでより詳しく知れそうを楽しみです

質問や感想など

- コンピュータに関わる分野の階層関係の図で、去年それぞれ別の授業で習ったことが一気につながったような感じがしました。将来ゲームを作りたいと思っているのですが、この授業が、ソフトを作る人にとってどう重要なのが分かりやすくて良かったです。

- 自動運転に関する説明がありましたが、かなり高度な計案を必要とするのなら、コンピュータの大きさを大きくする必要やコンピュータの数をふやす必要があると考えてよいでしょうか？
その場合、自動運転ができる自動車はそうでない自動車と大きさがあまり変わらないような気がするのですがどのような工夫がなされているのでしょうか。
スパコンは大きいイメージがあるのでスパコンを載せて走っているという説明を見たときに疑問に思いました。

質問や感想など

- 今後の授業の進み方として、コンピュータアーキテクチャの講義が例年と少し異なることになる可能性を懸念しています。三年次にある集中講義演習にも関わってくる教科なので、どのように学習を進めていけばよいのかなど、授業の合間などに教えてほしいです。こ

質問や感想など

- 休講情報はお茶大のポータルサイトに情報を掲載する場所がありますが、実際は教員も生徒もポータルサイトを頻繁には確認しないのでメールで送る先生が多いです。実際生徒もその方が助かります。情報科の生徒には入学時に学科のGmailが配布されているので履修者が情報科だけのようでしたらそこのメーリスが楽だと思います。

コンピュータ アーキテクチャ I 第2回

コンピュータに関わる分野の階層関係

アプリケーション・ソフトウェア
画像処理 / 音声認識 / 言語処理 / 機械制御
機械学習 / AI / WEB サービス / 暗号 ...

この講義の範囲

システム・ソフトウェア
OS / コンパイラ / インタプリタ

(狭義の) コンピュータ・アーキテクチャ

論理回路

集積回路 / 半導体デバイス

今日の講義の進め方

- まずノイマン型コンピュータの基本をわかってもらう
 - 命令やプログラム, 機械語とはなにか
 - 単純な CPU の構造と動作
 - これらを簡単な例を元に説明
- 今回はついでに以下も：
 - C 言語との対応
 - 実際の命令セットの例
 - (時間が足りないかキリが悪いかもしれないので次回にするかも)

もくじ

1. 命令やプログラム, 機械語とはなにか
2. 単純な CPU の構造と動作
3. C 言語と機械語の対応
4. 実際の命令セットの例 (やや発展)

命令やプログラム，機械語とはなにか

プログラム

- プログラムとは
 - 計算の手順を表したもの
 - 実体：メモリの上にある，計算方法を指示する数字（命令）の列
- （フォン）ノイマン型 (von Neumann-type) コンピュータ
 - プログラムに従って計算をする機械
 - メモリに格納された命令を取り出して順に実行
 - 他にもあるけど，これが今日では主流
- 次項から，簡単な例を使って説明

例 : $A + B - C$

- 「 $A+B-C$ 」の計算の手順 :

1. A と B を足す
2. 1.の結果から C を引く

- なんとなく形式的に表すと :

1. `add A, B → D` // D は一時的に結果をおいておく変数
2. `sub D, C → E` // E に $A + B - C$ の結果

例 : $A + B - C$

■ 形式的に表すと :

1. `add A, B → D` // D は一時的に結果をおいておく変数
2. `sub D, C → E` // E に $A + B - C$ の結果

■ 数字の列で表してみる :

● 変換の規則 :

意味 :	add	sub	A	B	C	D	E
数字 :	0	1	2	3	4	5	6

● 数列 :

1. 0, 2, 3, 5 // `add(0) A(2), B(3) → D(5)`
2. 1, 5, 4, 6 // `sub(1) D(5), C(4) → E(6)`

例 : $A + B - C$

■ 数列を1次元に展開すると :

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

意味 :	add	sub	A	B	C	D	E
数字 :	0	1	2	3	4	5	6

■ 先頭から順に数字を読んで, 変換規則をみながら計算する

1. 先頭は 0 なので, これは足し算
 - 続く2つの数字は足し算の入力で, その次は出力
2. 次は 2 なので, これはA. 次は 3 なので B
3. 結果を 5(E) に入れる ...
4. 次は 1 なので...

プログラムの表現と用語（1）

- バイナリ： 0, 2, 3, 5, 1, 5, 4, 6
 - 計算方法を表す数字の列
 - コンピュータが直接理解できるのは、このバイナリのみ
- アセンブリ言語： add A, B → D
 - バイナリと1：1に対応しており、基本的に「相互に」変換可能
 - 要はバイナリを人間にとって読みやすくしたもの
- 機械語：
 - 上記のバイナリないしはアセンブリ言語で表現されたプログラム

プログラムの表現と用語（2）

■ 命令：

- コンピュータが解釈できるプログラム内の計算手順の最小単位
- 「0, 2, 3, 5」 「add A, B→D」

■ オプコード (opcode)

- 命令でどういう計算をするか指定する部分
- 「0, 2, 3, 5」 「add A, B→D」

■ オペランド (operand)

- 計算の入出力対象を指定する部分
- 「0, 2, 3, 5」 「add A, B→D」
- 入力をソース, 出力をディスティネーション とよぶ

命令セット・アーキテクチャ

- バイナリの数字と実際に行う計算のルールを定めたもの：
 - どのような演算をサポートするか
 - 「add, sub ...」
 - バイナリのどの数字にどのような意味を持たすか
 - 「0 なら add」
 - 数字の順番の意味
 - 「最初の 1 桁が計算の種類, 次が入力・・・」
 - 各数字に何桁（何ビット）割り当てるか
 - 「10進数で 1 桁ずつ」

命令セット・アーキテクチャ

- ルールはコンピュータ（CPU）の種類ごとに異なる
 - このルールを「命令セット・アーキテクチャ」という
 - プログラムの「互換性」とは、上記のルールが同じであること
 - 他にも互換性に関わる要素には OS など色々な点があるが、最も重要なのはこれ

ここまでのまとめ

■ コンピュータとは

- プログラムに従って計算をする機械

■ プログラムとは

- 計算の手順を表したもの
- メモリの上にある, 命令（計算方法）の列

■ 命令とは

- コンピュータが解釈できる計算手順の最小単位
 - 最終的な実体としては, 数字の列
- 命令セット：
 - 命令の数字と, それに対応する計算方法を定めたもの

単純な CPU の構造と動作

単純な CPU の構造と動作

■ 下記のようなプログラムを処理できる, 最低限のコンピュータを説明

1. 0, 2, 3, 5 // add(0) A(2), B(3) → D(5)
2. 1, 5, 4, 6 // sub(1) D(5), C(4) → E(6)

コンピュータ

- 「プログラム = 命令列 = 数字の列」に従って計算をする機械
- 構成要素：
 - CPU （計算するもの）
 - 演算器
 - レジスタ
 - PC
 - メモリ（データを記憶するもの）

メモリ

データ :	07h	10h	30h	...	3Eh	01h	32h	20h	80h	...
アドレス :	0h	1h	2h	...	8000h	8001h	8002h	8003h	8004h	...

- メモリは命令列と、計算するデータを保持する
 - 単一の巨大な配列があると思えばよい
 - C 言語の配列は、これを切り出してユーザーに見せている
- 数字が入る箱がたくさん並んでいるイメージ
 - アドレス : 箱の通し番号 (住所)
 - データ : 箱の中身の数字

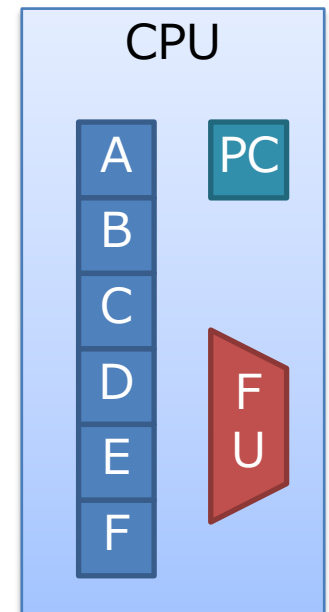
CPU

■ コンピュータの心臓部

- メモリから命令を読み出し，計算する

■ 構成要素：

- 演算器（FU: Functional Unit）
 - 加算器や AND 演算器など
 - 指示された種類の演算を行う
- レジスタ・ファイル（右図では A,B,C...）
 - メモリと同様にデータを記憶する
 - ◇ 位置を指定して読み書きする
 - CPU の演算は，このレジスタ上でのみ行う
- PC（Program Counter）
 - 現在見ている命令のアドレスを記憶している場所



CPU の動作

- PC (Program Counter) :
 - 現在処理する命令のアドレスを保持
- おおざっぱな命令の処理 :
 1. PC が指すアドレスのメモリから読む
 2. 読んできた命令に応じて処理をする
 3. PC を更新 (数字をたす)
 4. 1. にもどる

CPU の動作は料理に似ている

- レシピをみながら料理をするのに似ている
 - レシピの各手順が命令
 - 「今何個目の手順を見てるか」, を憶えているのかが PC
 - ひとつひとつ手順を取り出して, 指示に従って処理

カレーのレシピ

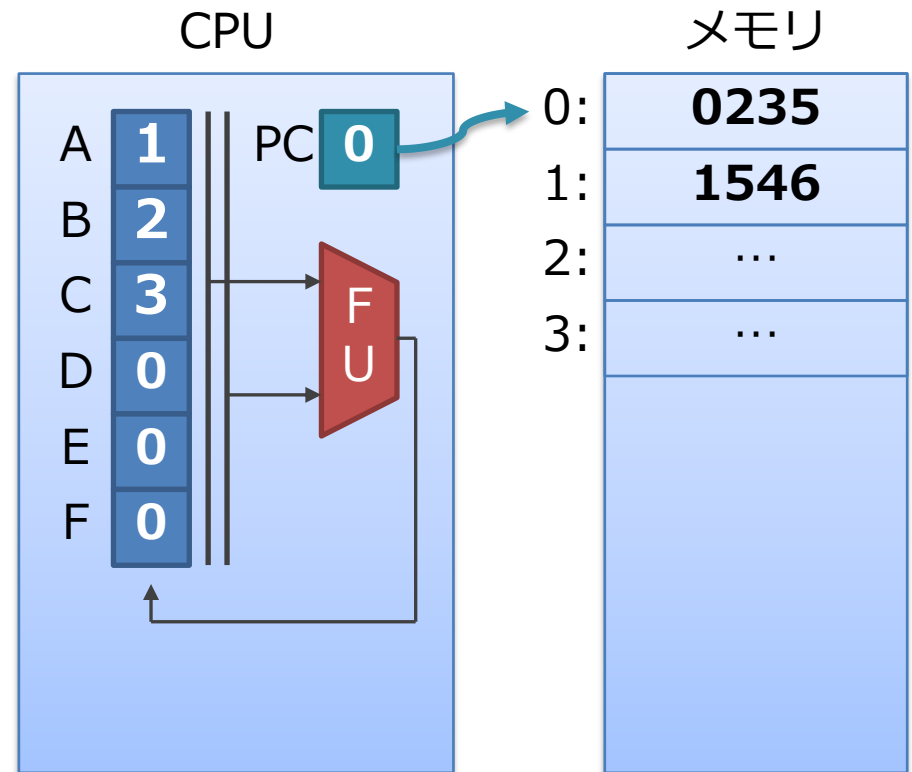


具体的な命令の処理

1. 命令のメモリからの読み出し（フェッチ）
2. 命令の解釈（デコード）
3. レジスタの読み出し
4. 演算の実行
5. 結果のレジスタへの書き戻し

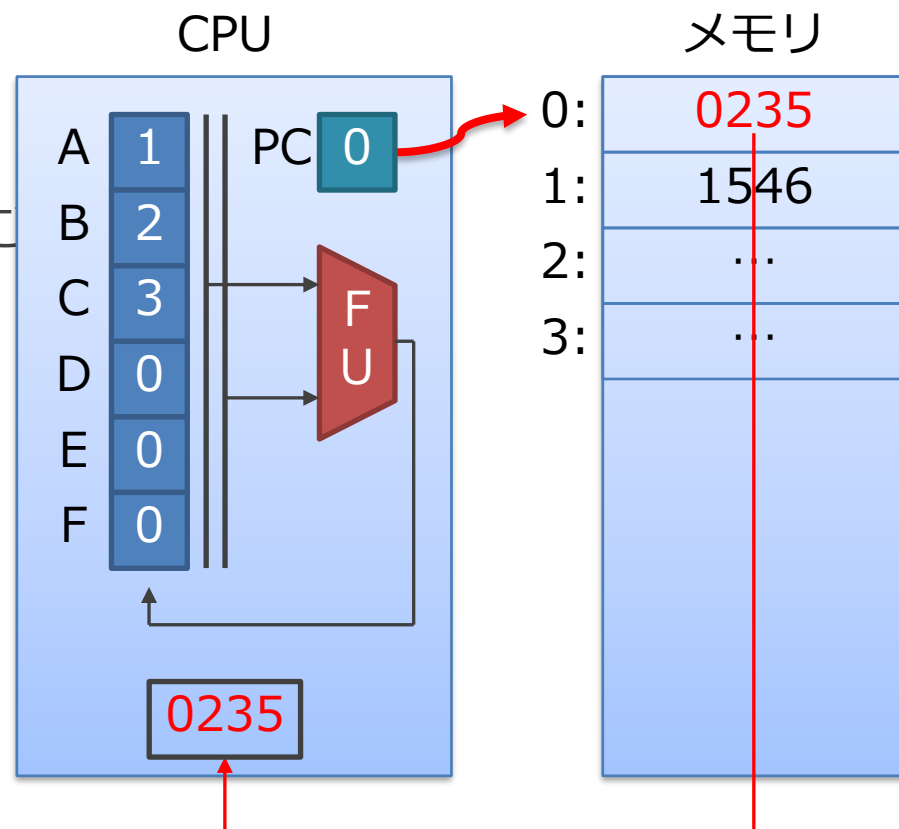
0. 初期状態

- PC はアドレス 0 を指している
- レジスタの初期値は1,2,3...
- メモリには,
 - 0番地に 0235 (add A,B→D)
 - 1番地に 1546 (sub D,C→E)



1. 命令の読み出し（フェッチ）

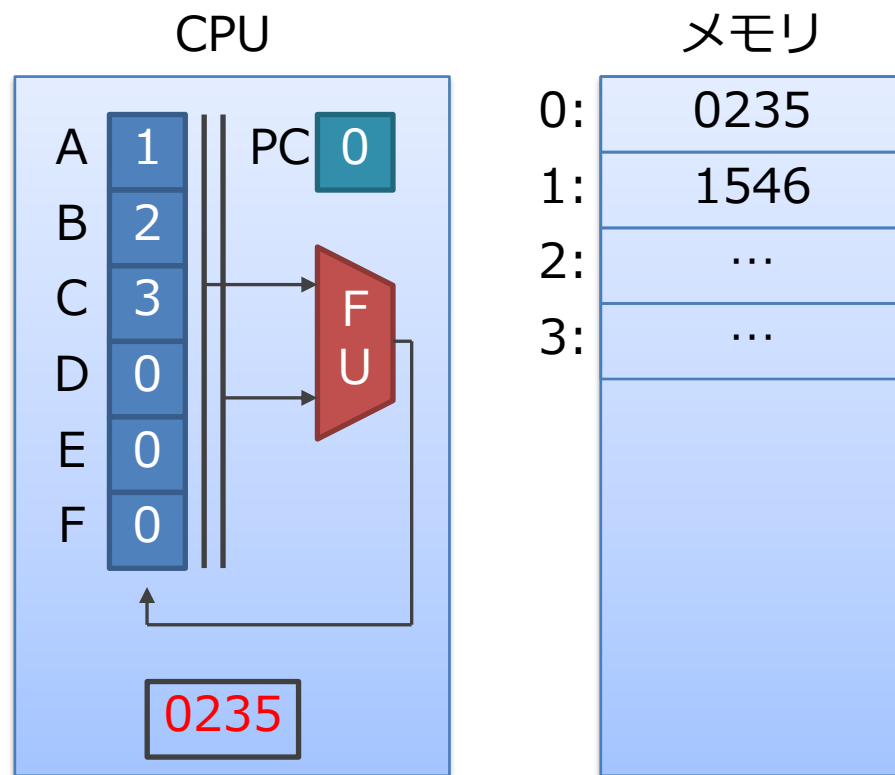
1. PC が指している命令の番地を読む
 1. 今はアドレス 0 を指している
2. 内容である 0235 が得られる
3. CPU 内にもってくる



2. 命令の解釈（デコード） オペコードやオペランドが何かを割り出す

1. 0235 の意味を解釈する

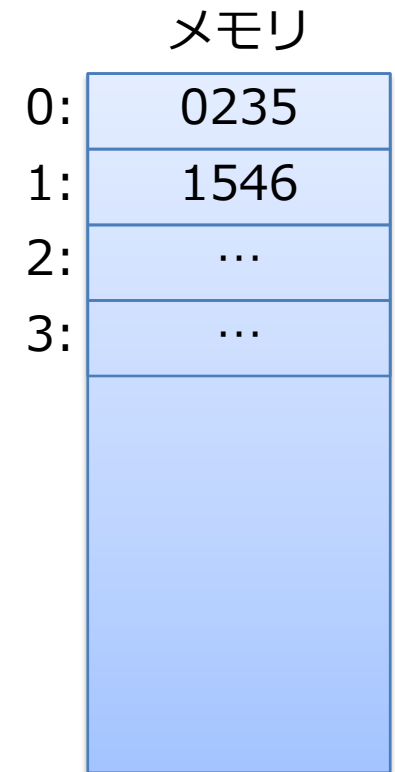
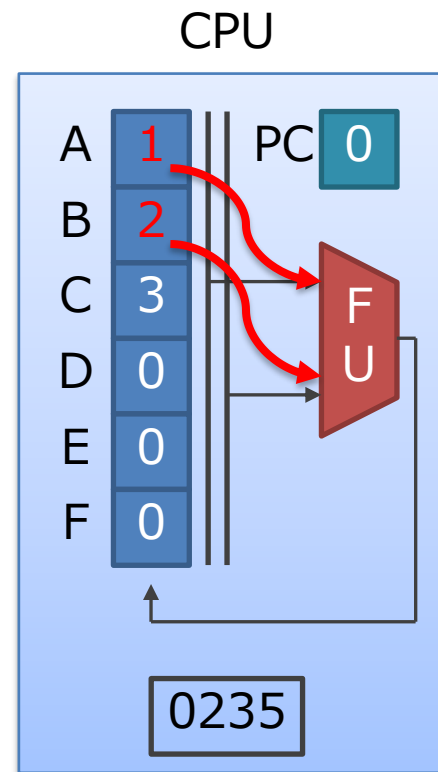
- 0: add
- 2: レジスタAを読む
- 3: レジスタBを読む
- 5: 結果はレジスタDに書く



意味：	add	sub	A	B	C	D	E
数字：	0	1	2	3	4	5	6

3. レジスタ読み出し

1. A と B をレジスタから読みだす
 - 先ほどのデコード結果に従う
2. 中身である 1 と 2 が取れる
 - 「0235」はレジスタの「読み出す場所」を表してることに注意

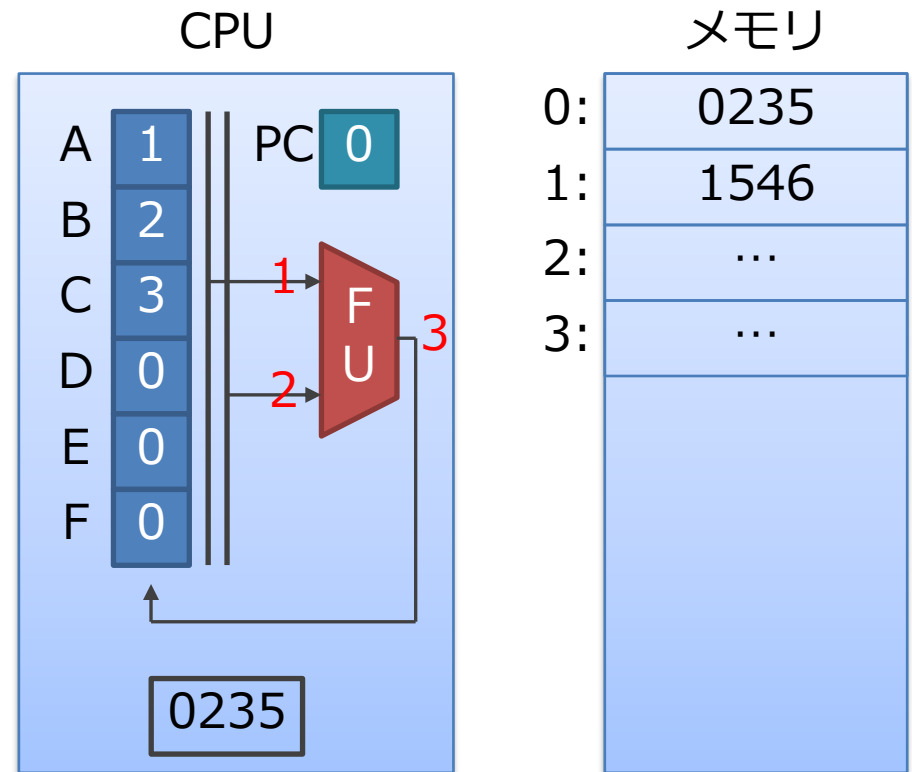


意味 :	add	sub	A	B	C	D	E
数字 :	0	1	2	3	4	5	6

4. 演算の実行

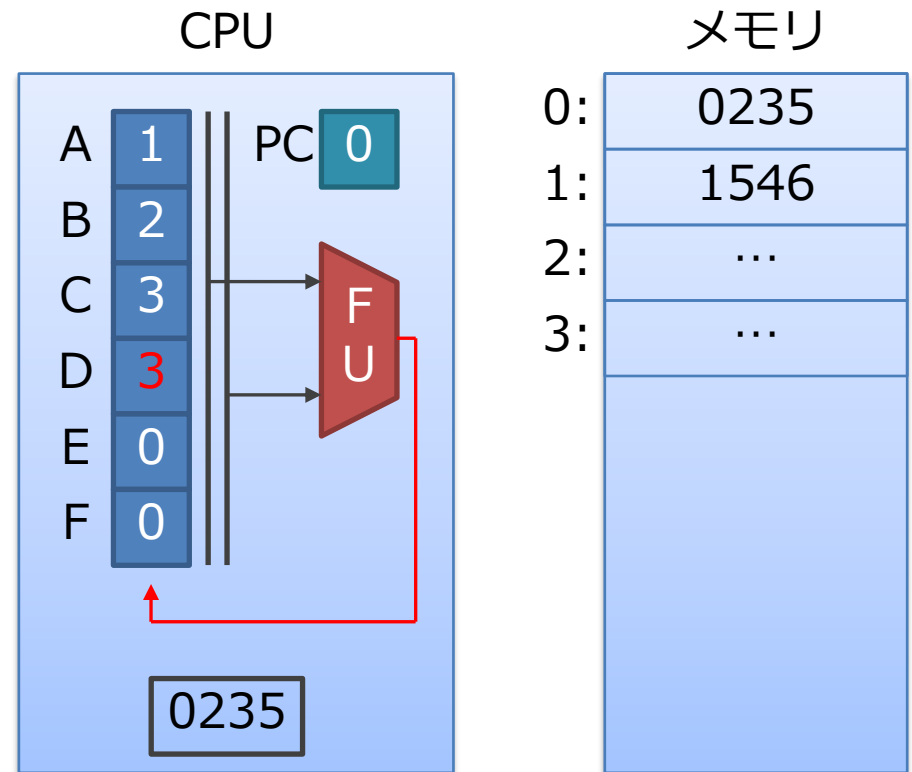
1. 演算器 (FU) で, 足し算をする

- $1 + 2 = 3$



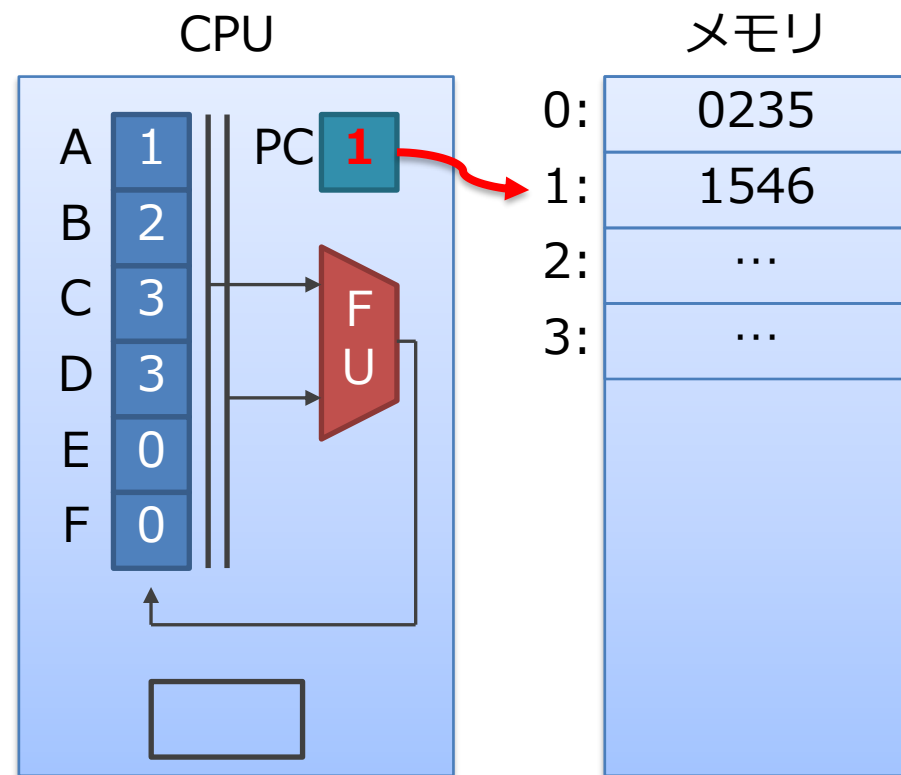
5. レジスタへの結果の書き戻し

1. D に結果の3を書き込む



6. 次の命令へ

1. PC に + 1 をする
2. 1. の命令の読み出しに戻る



その他の命令

- その他各種の演算命令 = 乗算や除算, 論理演算など
 - これらは, 演算器で行う演算の内容が異なるのみ
 - CPU 全体の制御は, 加算や減算と同様に行えばよい
- 演算とは異なる, その他の命令:
 1. メモリの読み書き
 2. 制御
 3. 即値 (レジスタの値の書き換え)

メモリの読み書き

■ メモリの読み書き

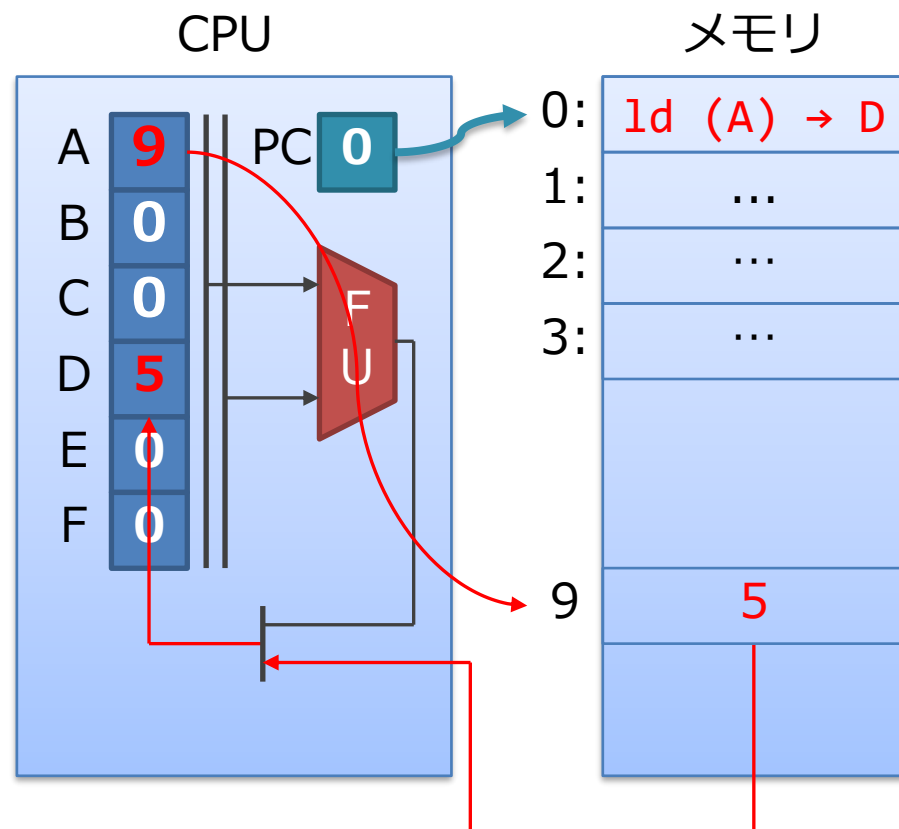
1. ロード命令：メモリからデータを読み出す
2. ストア命令：メモリへデータを書き込む

ロード命令 (ld: load)

■ ld (A) → D

- A の中が指しているメモリの場所を D に読み込む
- (A) は, C 言語 で言う *A

1. A の中身であるアドレス 9 を, メモリから読むと 5 が取れる
2. 5 を D に書き込む

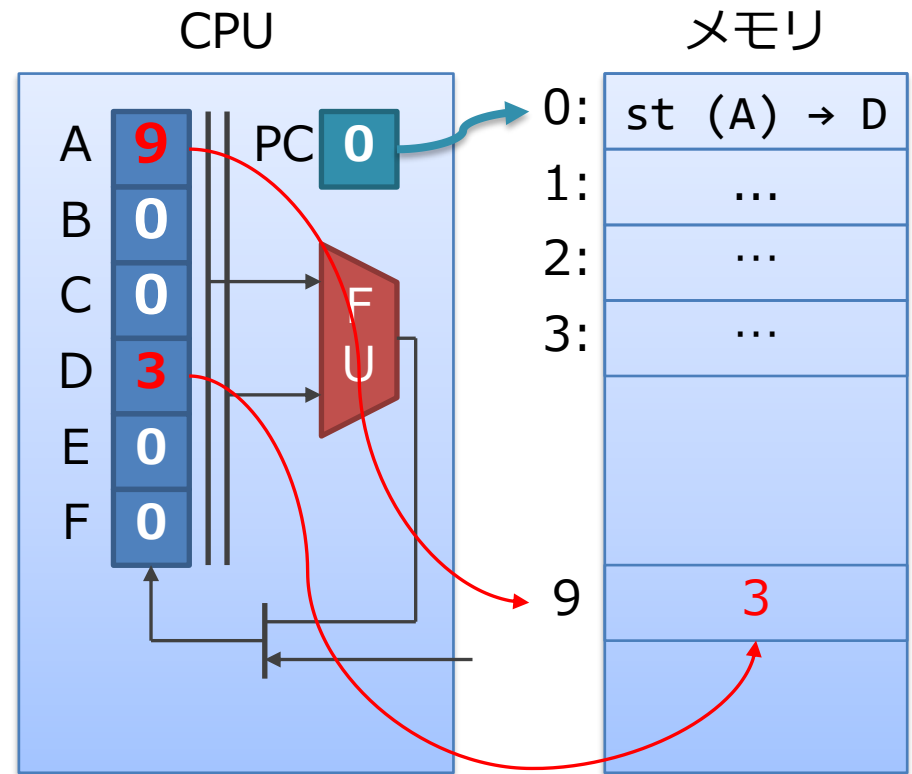


ストア命令 (st: store)

■ st D → (A)

- A の中が指しているメモリの場所に D を書き込む

1. A の中身であるアドレス 9 に,
D の中身 5 を書き込む



制御命令

■ 制御：

- PC を +1 するかわりに，任意の値に書き換えること

1. ジャンプ命令

- プログラムの任意の場所に移動する

2. 分岐命令

- 条件に応じて，プログラムの任意の場所に移動する

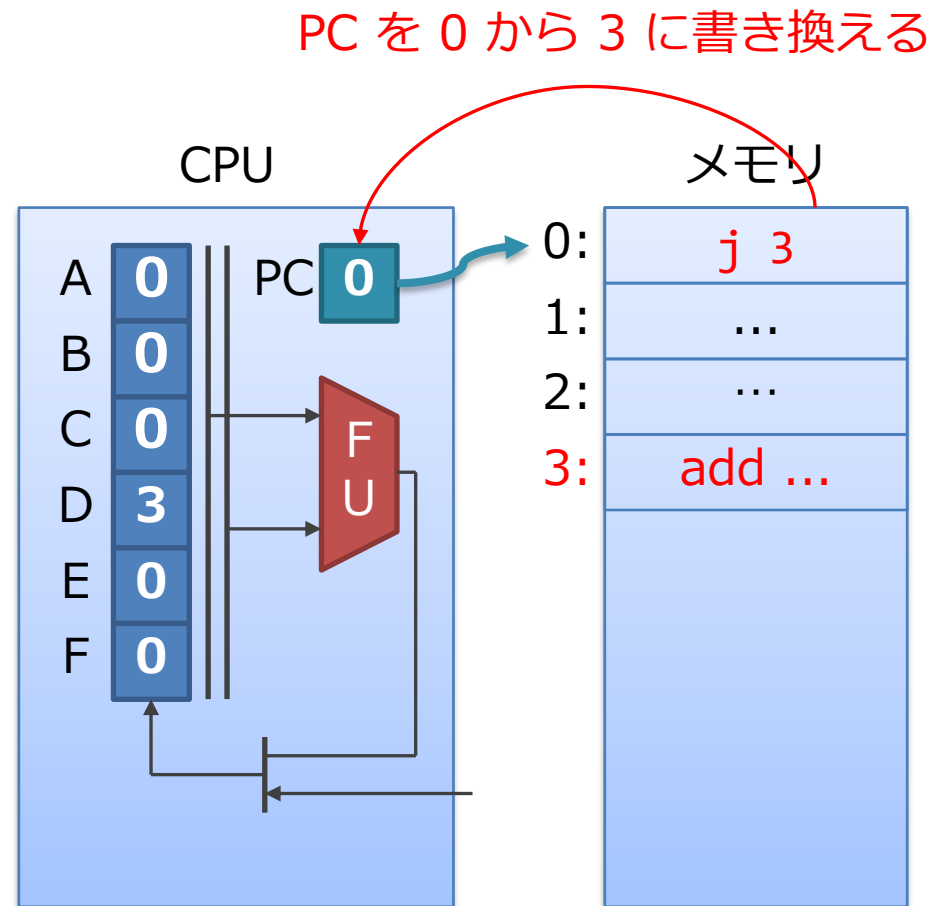
ジャンプ命令 (j: jump)

■ j N

- (N は任意の数字)
- PC を N に書き換える
- 次はアドレス N にある命令が実行される

■ 動作例

1. j 3 をフェッチ
2. PC を 3 に書き換える
3. アドレス 3 にある add を実行



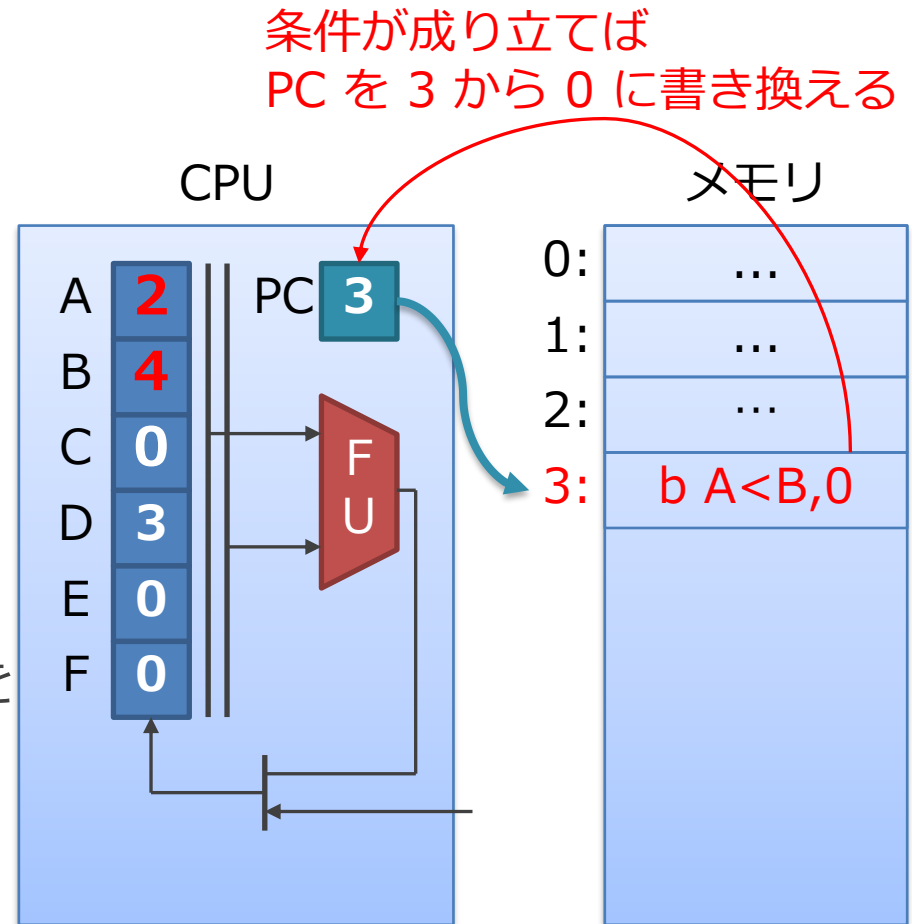
分岐命令 (b: branch)

■ b A < B, N

- レジスタを2つ読んで、
A < B なら、N に飛ぶ

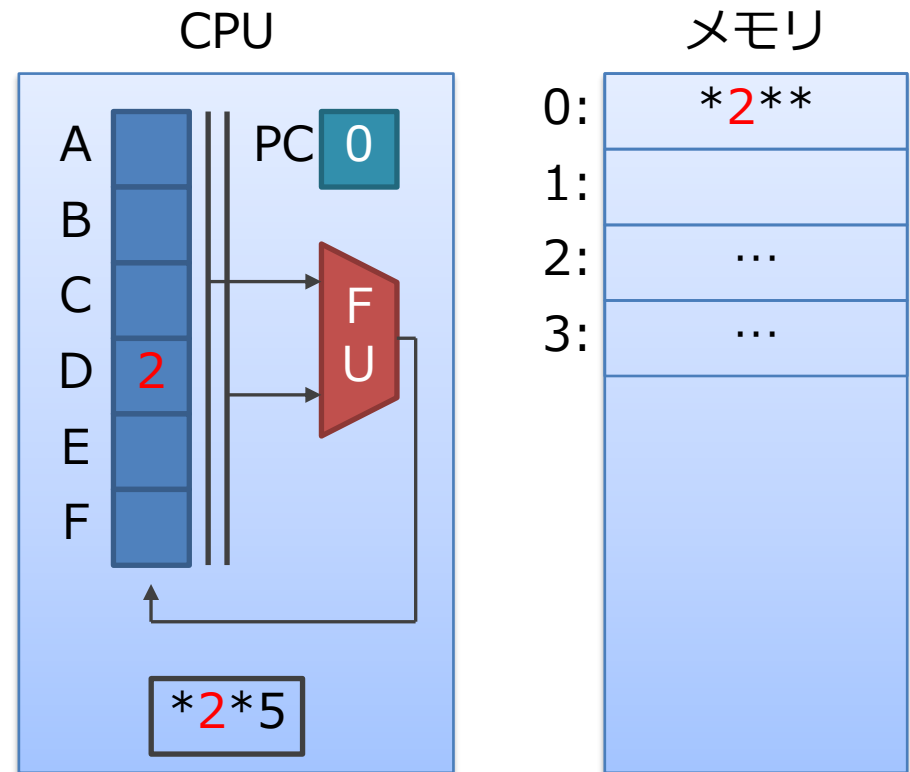
■ 動作例

1. A にある 2 と B にある 4 を
読んで比較
2. A < B なので、PC を 0 に
書き換える
3. 次は アドレス 0 にある命令が
実行される



即値（レジスタの値の書き換え）

- `li 2 → D`
 - (load immediate)
- 即値命令は命令内の数字を、直接レジスタに書き込む
- 他の命令は、命令内の数字を「レジスタの位置」と解釈
- レジスタの初期値を設定することなどに使用



プログラムの例：10回だけ回るループ

■ レジスタ初期値

- PC : 0 // 0 番地の命令から開始
- A : 0 // ループ・カウンタ
- B : 1 // インクリメント量
- C : 10 // ループ回数

■ 動作

0: add A+B→A: A に B を足して, A に書き戻す
1: b A<C,0 : A < C ならアドレス 0 に戻る
もし A > C ならアドレス 2 に

レジスタ

A	0
B	1
C	10
D	
E	
F	

メモリ

0:	add A+B→A
1:	b A<C,0
2:	
3:	



ここまでのまとめ

■ 単純な CPU の構造

- 演算器 (FU) , レジスタ, PC

■ 動作 :

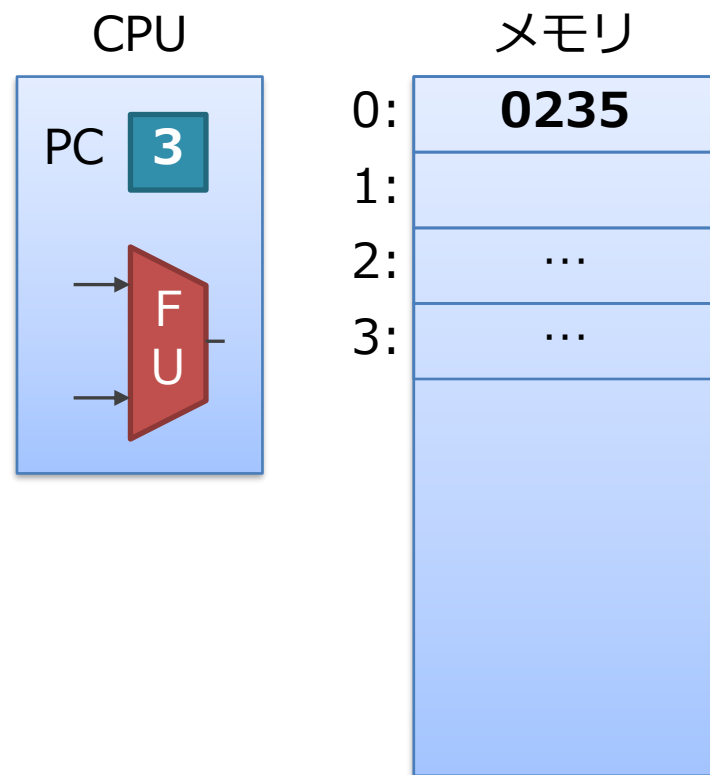
- PC に従ってメモリから命令を読み出し, それを 1 つずつ処理

■ 命令の例

- 演算, ロード/ストア, ジャンプ/分岐, 即値

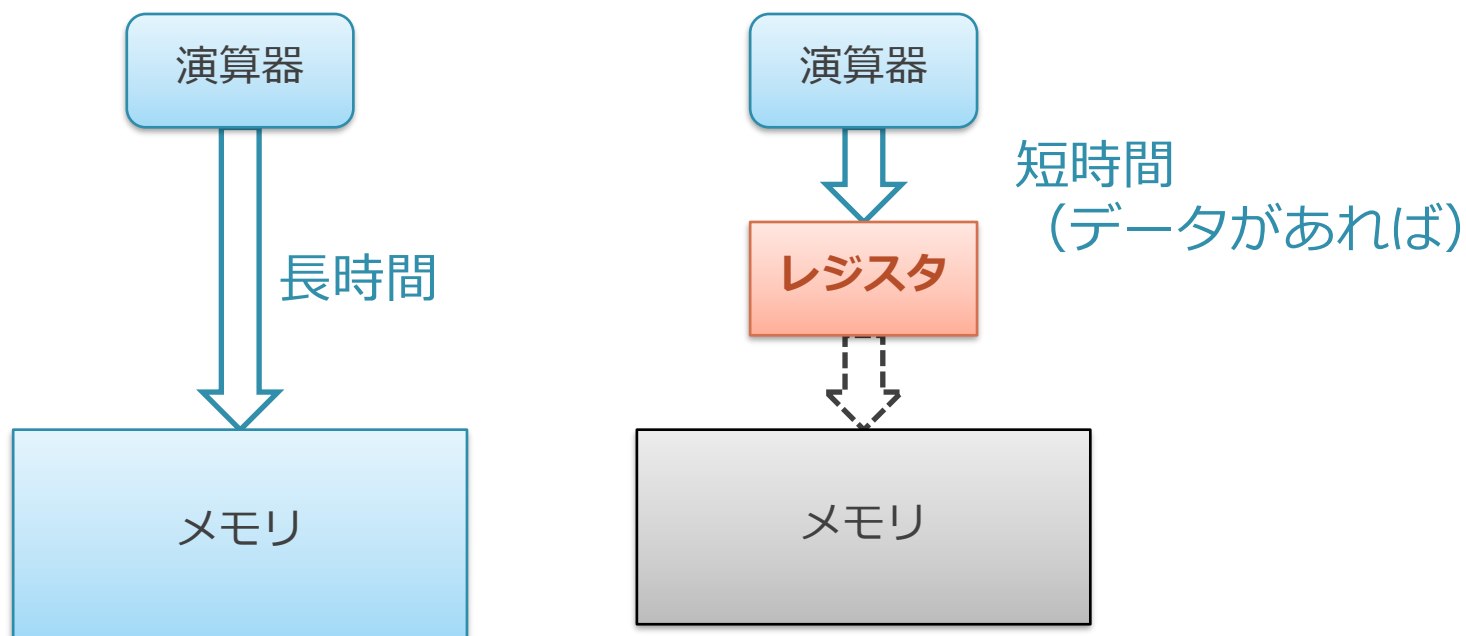
余談：メモリのみでもコンピュータは作れる

- レジスタは必須の存在ではない
- メモリのみでも等価なものは作れる
 - 命令中のレジスタの指定 (A, B, C ...) をメモリのアドレスだと思えばよい
- 昔の命令セットでは, ほぼメモリのみで計算を行うものも実際ある



なぜレジスタとメモリがあるのか？

- 問題：メモリは大容量だが，その分遅い
- 小容量だけど，高速なレジスタを用意
 - 一度利用した値を入れておくことで，2回目からは高速に
 - 一度使用したデータは，また使う可能性が高い



データをとってくるのに、どのぐらいかかるか？



C 言語と機械語の対応

C 言語で書かれたプログラムを動作させるには

- ここまでに説明した CPU でも、大概のことはできそう
 - 任意の場所のメモリの読み書き
 - ループ, 分岐
- コンパイラの処理 :
 - C 言語で書いた各ステートメントを, 対応する機械語に置き換える
 - 基本的にはパターンマッチング

C 言語で書かれたプログラムを動作させるには

- 具体的に, C 言語の構文をみていく
 - まずは例としてループを手でコンパイルしてみる
- C 言語の構文から, どのような命令が必要なのかを検討

C 言語のループ

■ C 言語のループについて考える

```
1: for (i = 0; i < 10; i++) {  
2: }
```

■ そのままだと考えづらいので、 まず上記のループを下記の形に変換して考える

```
1:      i = 0;           // 初期化部分  
2: LABEL:               // ループの先頭  
3:      i = i + 1;       // カウンタの更新  
4:      if (i < 10)      // ループの継続判定  
5:          goto LABEL;  // LABEL に戻る
```

前準備：変数の割り当て

```
1:      i = 0;
2: LABEL:
3:      i = i + 1;
4:      if (i < 10)
5:          goto LABEL;
```

変数表

変数	アドレス
i	0x0f4

メモリ

	...
0x0f0	...
0x0f4	i
0x0f8	...
	...
	...
	...
0x400	ここから命令
0x404	...
0x408	...

■ 変数 i をメモリの 0x0f4 番地に割り当てる

- グローバル変数だと思ってほしい
- 変数は1つ4バイトとする
- 適当にあいてるところを選んだだけで、番地の数字に意味はない

■ 命令は 0x400 番地から開始するとする

- 命令も1つ4バイトとする

1 行目：変数 i への 0 の代入

```
1: i = 0;
```

```
// レジスタ A に 0 を入れる
```

```
0x400: li 0      → A
```

```
// B に 0x0f4 (i の番地) を入れる
```

```
0x404: li 0x0f4 → B
```

```
// A を (B) にかきこむ
```

```
0x408: st A → (B)
```

メモリ

	...
0x0f0	...
0x0f4	i: 0
0x0f8	...
	...
	...
	...
0x400	li 0 → A
0x404	li 0x0f4 → B
0x408	st A → (B)
0x40C	...

■ グローバル変数の更新は，基本的にこのパターンでできる

- 変数のアドレスを li で読んで，そこにストア

2行目：ラベル

```
1:      i = 0;
2: LABEL:
3:      i = i + 1;
4:      if (i < 10)
5:          goto LABEL;
```

- 次の3行目は, **0x40C** から始まるので,
LABEL=0x40C として憶えておく

メモリ	
	...
0x0f0	...
0x0f4	i: 0
0x0f8	...
	...
	...
	...
0x400	li 0 → A
0x404	li 0x0f4→B
0x408	st A → (B)
0x40C	...

3 行目：変数 i のインクリメント

```
3: i = i + 1;
```

```
// B に 0x0f4 (i の番地) を入れる
```

```
0x40C: li 0x0f4 → B
```

```
// (B) を A に読み込む
```

```
0x410: ld (B) → A
```

```
// 1 を足す
```

```
0x414: add A, 1 → A
```

```
// A を (B) にかきこむ
```

```
0x418: st A → (B)
```

メモリ

	...
0x0f0	...
0x0f4	i: 1
0x0f8	...
	...
	...
	...
0x400	li 0 → A
0x404	li 0x0f4→B
0x408	st A → (B)
0x40C	li 0x0f4→B

4-5行目：ループの継続判定とジャンプ

```
2: LABEL:
3:     i = i + 1;
4:     if (i < 10)
5:         goto LABEL;
```

// B に 10 を読み込む

li 10 → B

// さっきのインクリメントの結果が残ってる A と

// 比較し, 条件がなりたっていたら LABEL に

b A < B, 0x40C

// 条件がなりたっていなかったら, 以降の命令に

メモリ

	...
0x0f0	...
0x0f4	i: 1
0x0f8	...
	...
	...
	...
0x400	li 0 → A
0x404	li 0x0f4→B
0x408	st A → (B)
0x40C	li 0x0f4→B

全体

```
1:      i = 0;
        0x400: li 0      → A    // レジスタ A に 0 を入れる
        0x404: li 0x0f4 → B    // B に 0x0f4 (i の番地) を入れる
        0x408: st A → (B)      // A を (B) にかきこむ (= i を更新)

2: LABEL:

3:      i = i + 1;
        0x40C: li 0x0f4 → B    // B に 0x0f4 (i の番地) を入れる
        0x410: ld (B)   → A    // (B) を A に読み込む (= i 読み込む)
        0x414: add A,1  → A    // A に 1 を足す
        0x418: st A → (B)      // A を (B) にかきこむ (= i を更新)

4:      if (i < 10)
5:          goto LABEL;
        0x41c: li 10 → B      // B に 10 を読み込む
        0x420: b A < B, 0x40C // 条件がなりたっていたら LABEL に
```

C 言語への変換（コンパイル）

- 基本的には1つ1つの文を，機械語に置換していけばよい
 - さっきの結果はかなり冗長なので，実際にはもっと最適化する
 - 変数 i を毎回メモリから読み書きしていたのを省略するとか
- ただし，デバッグ用にコンパイルしたコードはさっきの例に近い
 - デバッガでステップ実行するためには，元の文と1 : 1に対応していた方が都合が良い
- C 言語の演算子と制御構文の全体について見ていく

C 言語 の 演算子 (優先順位順)

	記述例	説明		記述例	説明
1	a[i]	配列アクセス	4	a * b a / b a % b	乗除算
	f(a)	関数呼び出し	5	a + b a - b	加減算
	s.m sp->m	構造体アクセス	6	a << b a >> b	シフト
	a++ a--	インクリメント, デクリメント	7	a < b a <= b a > b a >= b	比較
2	++a --a		8	a == b a != b	
	&a	アドレス	9	a & b a ^ b a b	ビットごとの論理演算
	*p	デリファレンス	12	a && b a b	論理演算
	+a -a	単項 + と -	14	c ? l : r	条件
	~a	ビット反転	15	a = b	代入
	!a	論理否定		a += b a -= b	演算 と 代入
	sizeof a	サイズ	16		
	(t)a	キャスト		a, b	コンマ

算術・論理演算

アドレス (ポインタ)

その他言語上の作用

変数, アドレス, ポインタ

■ 変数, 配列, 構造体アクセス

● 変数の出現

○ $x \Rightarrow *(&x)$

● 配列

○ $a[i] \Rightarrow *(a + i)$

● 構造体

○ $s.m \Rightarrow *(&s + offset)$

○ $sp \rightarrow m \Rightarrow *(sp + offset)$

■ 下記があればよい :

● アドレスに対する演算

● アドレスを指定したロードとストア

変数表

変数	アドレス
x	0x0fc
a	0x100
s	0x110
m	0x4

メモリ

	...
0x0fc	...
0x100	...
0x104	...
0x108	...
0x10C	...
0x110	...
	...
	...
	...
	...
	...

C言語 の 実行順序の制御

■ C 言語の制御構文

- if ~ else
- for, while, do ~ while
- switch ~ break, continue
- return
- goto

■ これらは基本的にぜんぶ if ~ goto に書き換え可能

- return だけ, これまでに説明した命令では作れない
- ジャンプするときに, PC が戻るアドレスを保存する命令が必要

C 言語を実行するためには

- おおよそ CPU にはこれだけの命令あればよい
 - 各種演算, アドレス計算
 - アドレスを指定した読み書き
 - if ~ goto 的な分岐 + return

C 言語と機械語は結構近い

- 「C 言語がこうだから、コンピュータをこう作ろう」ではなく,
 - 「コンピュータがこうだから、C言語 がこうなった」
 - 「C 言語は、読みやすいアセンブリ言語」とか言われる

ここまでのまとめ

- C 言語 コンパイラの処理：
 - 各ステートメントを，対応する機械語に置き換える
 - 基本的にはパターンマッチング
- C 言語を動かすためには，たとえば下記があればよい
 - 各種演算，アドレス計算
 - アドレスを指定した読み書き
 - if ~ goto 的な分岐 + return

実際の命令セットの例（やや発展）

実際の命令セットの例

- 「RISC-V」 を例としてとりあげる
 - 比較的最近登場した, CPU の命令セットのオープンな規格
 - 「オープン」とは, 自由に互換品を作ってもよいということ
 - 他の商用の命令セットの互換品を作って公開すると訴えられる
- やや内容は発展的
 - 大ざっぱにわかっているだけで OK

<https://riscv.org/members/> より



- 左上がプレミアムメンバー
- 右上が戦略メンバー
 - 日本からは日立、ソニー、ルネサス、NSITEXE

RISC-V 命令セットの基本

■ 基本的な特徴：

- 各命令は 4 バイトのデータで構成
- レジスタは 32本 ある
 - A,B,C ... ではなく, x0, x1, x3... と表記
 - うち 1 つはゼロレジスタ
(常にゼロが入っている)
- 各レジスタの幅は 32/64/128 bit が規定されている
 - ここでは 32bit のものを取りあげる

■ 基本的には、今日前半で話した命令セットを 2進数ベースできちんと整理した感じ

- 前半の話では、なんとなく適当に10進数で話していた



RISC-V の 基本整数命令

■ 概要

- 加減算, 論理演算,
ロード・ストア,
即値, 分岐とジャンプなど
- 各命令は 32bit 幅

- 前半の講義で 4 桁の数字で
表していたことと同じことを
32bit の中を細かく区切って
やっている

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20:10:1 11:19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12:10:5]		rs2	rs1	000	imm[4:1:11]	1100011	BEQ
imm[12:10:5]		rs2	rs1	001	imm[4:1:11]	1100011	BNE
imm[12:10:5]		rs2	rs1	100	imm[4:1:11]	1100011	BLT
imm[12:10:5]		rs2	rs1	101	imm[4:1:11]	1100011	BGE
imm[12:10:5]		rs2	rs1	110	imm[4:1:11]	1100011	BLTU
imm[12:10:5]		rs2	rs1	111	imm[4:1:11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
0000000000000			00000	000	00000	1110011	ECALL
0000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

画像は下記より

The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2

RISC-V の 基本整数命令の構造

- エンコーディング : R, I, S, U の4タイプがある
 - opcode によって, 32 bit 中をどう区切って解釈するかが変わる
 - funct は追加の opcode (opcode が大分類, funct が小分類)
- rs1, rs2, rd はオペランド
 - それぞれ 5bit: $2^5=32$ 本のレジスタを指定可能
 - imm は即値

31	25 24	20 19	15 14	12 11	7 6	0	
funct7	rs2	rs1	funct3	rd	opcode		R-type
imm[11:0]		rs1	funct3	rd	opcode		I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode		S-type
imm[31:12]				rd	opcode		U-type

R-Type の演算命令



ADD : $x[\text{rd}] \leftarrow x[\text{rs1}] + x[\text{rs2}]$



SUB : $x[\text{rd}] \leftarrow x[\text{rs1}] - x[\text{rs2}]$



■ ADD や SUB は R-Type となる

- opcode = 0110011 は R-Type
- funct7 の部分で, さらに ADD や SUB を判別

I-Type の演算命令



ADDI : $x[rd] = x[rs1] \leftarrow \text{immediate}$



- レジスタを読んだ値ではなく, immediate の部分をそのまま演算する

ADD と ADDI の違い

ADDI : $x[rd] \leftarrow x[rs1] + \text{immediate}$

immediate[11:0]	rs1	000	rd	0010011
-----------------	-----	-----	----	---------

ADD : $x[rd] \leftarrow x[rs1] + x[rs2]$

0000000	rs2	rs1	000	rd	0110011
---------	-----	-----	-----	----	---------

SUB : $x[rd] \leftarrow x[rs1] - x[rs2]$

0 <u>1</u> 00000	rs2	rs1	000	rd	0110011
------------------	-----	-----	-----	----	---------

- immediate の部分はあるべくビット幅を大きく取りたい
 - その方がより大きな数が扱える
 - ADDI には専用の opcode: 0010011 を割り当てる
- ADD や SUB はレジスタ番号が表せる 5bit があれば足りる
 - なので, opcode にまとめて funct7 で判別していた

I-Type のロード命令



LW : $x[rd] \leftarrow (x[rs1] + \text{immediate})$



- LW : Load Word 命令 (4バイトをロード)
 - opcode: 0000011 は ロード命令で I-Type
 - funct3 部分がかわると, バイト数が異なる他のロードに
- $(x[rs1] + \text{immediate})$ と加算が入っている
 - レジスタ値に即値を加算してアドレスとできると便利だから
 - $x[rs1]$ に構造体の先頭, immediate がメンバへのオフセットとか

S-Type の命令



SW : (x[rs1] ← immediate) = [rs2]



■ SW : Store Word 命令

- opcode: 0100011 はロード命令で S-Type
- funct3 部分がかわると, バイト数が異なる他のストアに

RISC-V の命令フォーマット

- 残りの命令は, 大体これのバリエーション

まとめ

■ 今日の内容

1. 命令やプログラム, 機械語とはなにか
2. 単純な CPU の構造と動作
3. C 言語と機械語の対応
4. 実際の命令セットの例 (やや発展)

- 感想や質問を投稿してください
 - Moodle の「感想や質問」のところからお願いします
 - 締め切り：水曜日中