コンピュータ アーキテクチャ I　第３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

もくじ

前回の振り返り
（プログラムと簡単な CPU）

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

- 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)

プログラムの表現と用語（１）

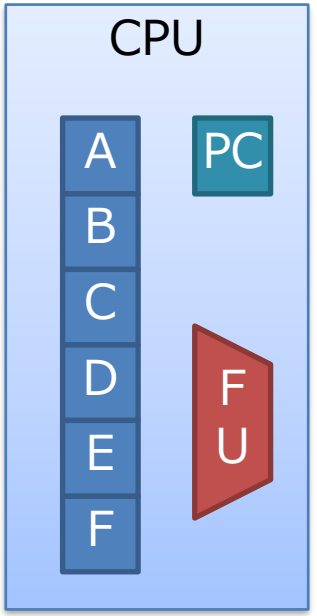
- バイナリ：　　0, 2, 3, 5, 1, 5, 4, 6
  - 計算方法を表す数字の列
  - コンピュータが直接理解できるのは，このバイナリのみ
- アセンブリ言語：　`add A, B → D`
  - バイナリと１：１に対応しており，基本的に「相互に」変換可能
  - 要はバイナリを人間にとって読みやすくしたもの
- 機械語：
  - 上記のバイナリないしはアセンブリ言語で表現されたプログラム

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

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

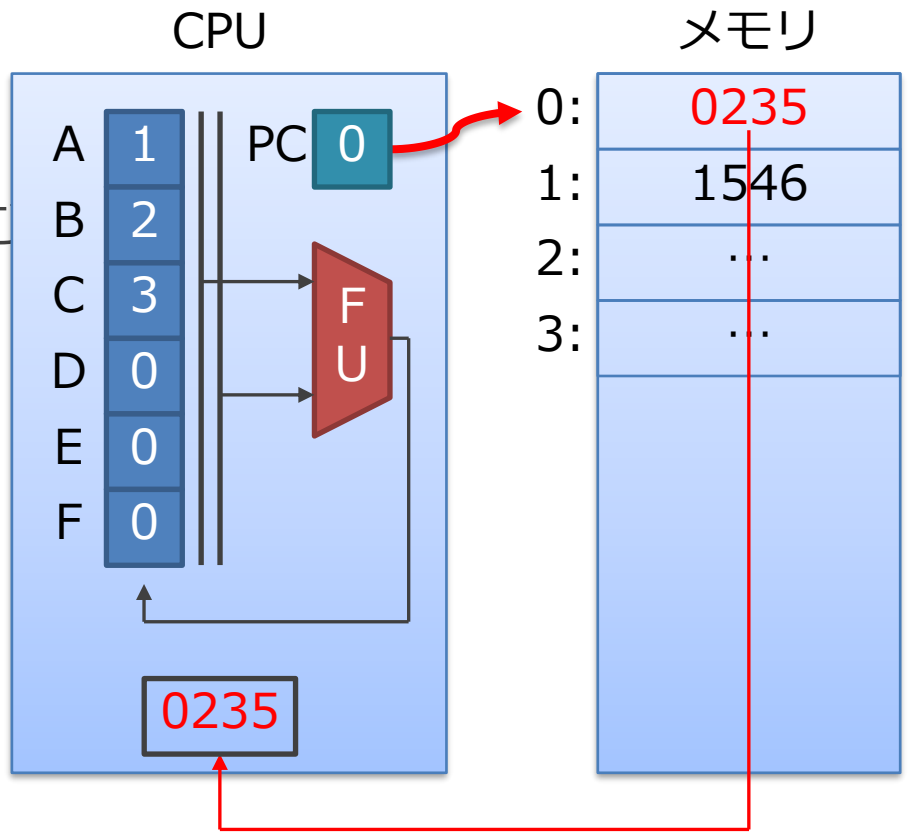


CPU の動作

- 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

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



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

課題２の解説

課題２で使用する命令セット

- レジスタ：
  - A, B, C, D, E, F の６つがあるものとする
- 命令：
  - 次のページで説明する li, add, sub, b の命令がある

課題２で使用する命令セット

- li：即値をレジスタに読み込む
  - 例：`li 0 → A　　// レジスタ A に 0 をいれる`
- add：加算
  - 例：`add A,1 → A　// レジスタ A に 1 を足して A に書く`
- sub：減算
  - 例：`sub B,C → D　// レジスタ B から C を引いて D に書く`

課題２で使用する命令セット

- b：条件分岐命令
  - 例： `b A < B, LABEL  // もし A<B であれば LABEL に飛ぶ`
  - 例： `b LABEL  // 条件式がなければ無条件に LABEL に飛ぶ`

課題２で使用する命令セット

- ラベルについて
  - C 言語のラベルと同様に，任意の命令の場所を表すためにラベルを使うことができる
  - ラベルには任意の名前をつけることが可能であり，「ラベル名：」で表記する

- 例：
  - ```
      li A←0
    LABEL_EXAMPLE:    // LABEL_EXAMPLE というラベルをここに定義
      add A←A+A
      b LABEL_EXAMPLE // LABEL_EXAMPLE に無条件で飛ぶ
                      // 具体的には add に飛ぶ
    ```
    ```

課題２ (1,2)

- (1) 以下の C 言語で書かれたプログラムをアセンブリ言語で表せ.

- (2) アセンブリ言語で書いたプログラムを実行した際の，実行される命令の系列を列挙せよ
  - (1) で書いたプログラムを実行した場合に，実行される命令を１つづつ，全部書いてくださいと言う意味でした

```
1: i=1;
2: if (i > 0)
3:   i++;
4: i--;
```

補足 1

- 「add A←A+1」「add A+1→A」「add A,1→A」はどれが正しいのか？
  - 意味がわかれば，どれでも良いです
    - このオリジナルのアセンブリ言語は実物ではないので，概念的な意味が表せて通じればそれでいいです
  - とはいえ，こちらの表記が一定していなくてすいません

補足２

- LABEL は定義する前に使っても構いません
  - 分岐の後に LABEL が出てきても問題ないです
  - 実は C 言語でも同じで，以下の様に書けます

```
int main() {
    int i = 1;
    goto LABEL;
    i++;
LABEL:
    return 1;
}
```

解答例

```
1: i=1;
2: if (i > 0)
3:   i++;
4: i--;
```

(1)

```
  li A←1
  b A<=0, LABEL
  add A←A+1
LABEL:
  sub A←A-1
```

(2)

```
  li A←1
  b A<=0, LABEL
  add A←A+1
  sub A←A-1
```

- if (i > 0) i++ は，逆に i<=0 の時に飛ばすようにするとシンプルに書ける

とてもダメな解答例

- この課題はオリジナルのアセンブリ言語を使って出題しています.
- しかし ChatGPT はそんなこと知らないので，一番代表的な x86 で書いてきます. しかも，元のプログラムの意味に含まれてないものまで入ってます.
- これは不正です. 絶対やめてください.

(1) 以下の C 言語で書かれたプログラムをアセンブリ言語で表せ.
(2) アセンブリ言語で書いたプログラムを実行した際の，実行される命令の系列を列挙せよ

```
1: i=1;
2: if (i > 0)
3:  i++;
4: i--;
```

アセンブリ言語で表すと以下のようになります：

assembly　コードをコピーする

```
section .data
    i dd 1      ; i の初期値を1に設定

section .text
    global _start

_start:
    ; i > 0 の比較
    mov eax, [i]    ; i の値をeaxにロード
    cmp eax, 0      ; eaxと0を比較
    jle end_if      ; i <= 0 の場合は end_if へジャンプ

    ; i++ の部分
    mov eax, [i]    ; i の値をeaxにロード
    inc eax         ; eaxをインクリメント
    mov [i], eax    ; eaxの値をiにストア

end_if:
    ; i-- の部分
    mov eax, [i]    ; i の値をeaxにロード
    dec eax         ; eaxをデクリメント
    mov [i], eax    ; eaxの値をiにストア

    ; 終了
    mov eax, 1      ; システムコール番号1 (exit) をeaxにセット
    xor ebx, ebx    ; 戻り値として0をebxにセット
    int 0x80        ; システムコール実行

section .bss
    resb 4          ; i のための領域を確保
```

２進数や１６進数による数値表現

２進数や１６進数による数値表現

- まず数値表現について軽く説明する
  - 実際の命令セットでは２進数や１６進数が多く出てくるため
  - 慣れてないと理解しづらい

表記方法

- ここから先では C 言語の表記方法に従う
  - １０進数：なにもつけない　　（143）
  - ２進数：先頭に 0b をつける　　（0b10000011）
  - １６進数：先頭に 0x をつける　　（0x83）

- 由来：
  - 0b = binary（英語の２進数）
  - 0x = hexadecimal（英語の１６進数）
  - bit = binary digit の短縮系

- 備考：
  - 0b による２進数表記は C23/C++14 規格から導入されたので，古い C 言語の資料では書かれていない

現代のコンピュータは基本的に２進数ベースで出来ている

- 電圧が「高い=1」or「低い=0」の２進数で表現されている
  - たまに１or 0 の割り当てが逆のこともある
- なぜ２進数なのか？
  - 回路を作るのが簡単だから

現代のコンピュータは基本的に２進数ベースで出来ている

- たとえば３進数の場合，
  - 電圧が「高い」「中間」「低い」の３つを区別することになる
  - この判別は，単に高いか低いかよりもかなり難しい
    - （後で実例を紹介
- 手の指で表した場合でも同じ？
  - 指が「折れている=0」「 立っている=1 」で２進数を表す
    - 2^{10}=1024 通りを表す事ができる
  - ３進数にすると「半分折れている」が加わる
    - 3^{10}=59049通りを表すことができる
    - しかし，「半分折れてる」の判別がむずかしい

情報分野では２進/１６進/１０進が混じって出てくる事が多い

- 注意：
  - これらは単に表記方法の違い
  - 表し方が違うだけで，それが示す数字そのものは同じ
- たとえば１０進の「12」は３通りに書けるが，意味は同じ
  - １０進： 12
  - ２進： 0b1100
  - １６進： 0xc
- 「リンゴ」を「林檎」「りんご」「Apple」と書いても意味するものは変わらないのと同じ

ではなぜ複数の表記を使うのか？

- それぞれの表記方に利点があるから
  - １０進は人間が普段使っているので，人間が考えやすい
  - ２進や１６進は？

２進数で表記する利点

- 前回の講義でだした CPU では１０進で命令を定義していた
  - 人間に分かりやすくするため
  - １０進数の各桁を見れば意味がわかる
    1. ０２３５　　// add(0) A(2), B(3) → D(5)
    2. １５４６　　// sub(1) D(5), C(4) → E(6)
- コンピュータは２進数の形で数字を保持している
  - ２進数の各桁に意味を持たす事が多い
  - ２進数表記した方が，色々と考えやすい

たとえば４ビットのデータを考える

- ４ビットのデータは 0b0000～0b1111 → 0 ～ 15 を表す事ができる
- ２進数で下から３桁目が add/sub を示すとする
  - ２進の場合：３桁目だけを見れば判定できる
    - add：0bx1xx （x は 0 or 1 の任意）
    - sub：0bx0xx
  - １０進の場合：かなり意味が分からない
    - add：4,5,6,7,12,13,14,15 のいずれか
    - sub：0,1,2,3,8,9,10,11 のいずれか

２進数の欠点

- 人間にわかりにくい
  1. 桁数が大きくなりすぎる
  2. １０進でどのぐらいの数字になるのか，検討をつけにくい
- たとえば
  - `0b1101010010000111 = 54407`

１６進数

- １６進数
  - １０進数の 0～15 を
    `0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7,`
    `0x8, 0x9, 0xa, 0xb, 0xc, 0xd, 0xe, 0xf` で表す
- 利点：
  - 表記上の桁数が少ないので，２進数よりは人間にわかりやすい
  - ２進数と１６進数は相互変換が簡単

２進数と１６進数の相互変化

- ２進数の４桁が１６進数の１桁に１：１：に直接対応する
  - ２進：0b 1101 0100 1000 0111
  - １６進：0x D 4 8 7
  - １０進：54407
- ２進数の４桁ごとに対応をとれば簡単に変換できる
  - １０進との相互変換はかなり難しい

（余談）手計算でやるとき

- ２進→１６進の手計算は，以下のように憶えると便利
  - ４桁ずつに区切る
  - 下から見て，１が 立っているところに１２４８をかけて足す
- たとえば，
  - ２進：`0b 1101 0100 1000 0111`
  - １６進：`0x D 4 8 7`
  - 一番上位の４桁は `1101 → 1+4+8=13 → D`
  - 次の４桁は `0100 → 4`

なぜ１６なのか？

- ２進数との相互変換のしやすさだけなら，８進や３２進でも良い
  - ２の累乗の進数なら，同様に各桁が１：１に対応する
- １６進数では１桁を表すのに４ビットが必要
  - ここのビット数も２の累乗の方が何かと都合がよい
    - ８進数だと３ビットになり，２の累乗から外れる
  - １０進に近いので，考えやすい

（余談）二進化十進表現（Binary-coded decimal: BCD）

- ２進数の４桁（１６進数の１桁）で１０進数の各桁を表す方法
  - 0b0000(0x0) ～ 0b1001(0x9) が 0～9 を示す
  - 例：0x1234 は 1234 を示す
- BCD の利点と欠点
  - 利点：桁が１：１に対応するので２進数と１０進数の相互変換が楽
  - 問題：0xa ～ 0xf は使わないので同じ桁数で表せる数字の数が減る

BCD

- 基本的に無駄
  - なので，現代ではあまり使われない
- ただし，お金を扱う場面では今でも使われる事がある
  - １０進でキリの良い数字は２進では循環小数になったりする
    - 0.1 は２進だと 0.0001 1001 1001 1001 1001 …
    - ２進では有限の桁数で表せないので誤差が出る
  - 税金や利子の計算などは１０進数前提でルールが組まれている
    - 消費税 10% のたびに微妙な誤差が出たらまずい
    - 利子が微妙に多くなったり小さくなるのもまずい

論理回路による実装

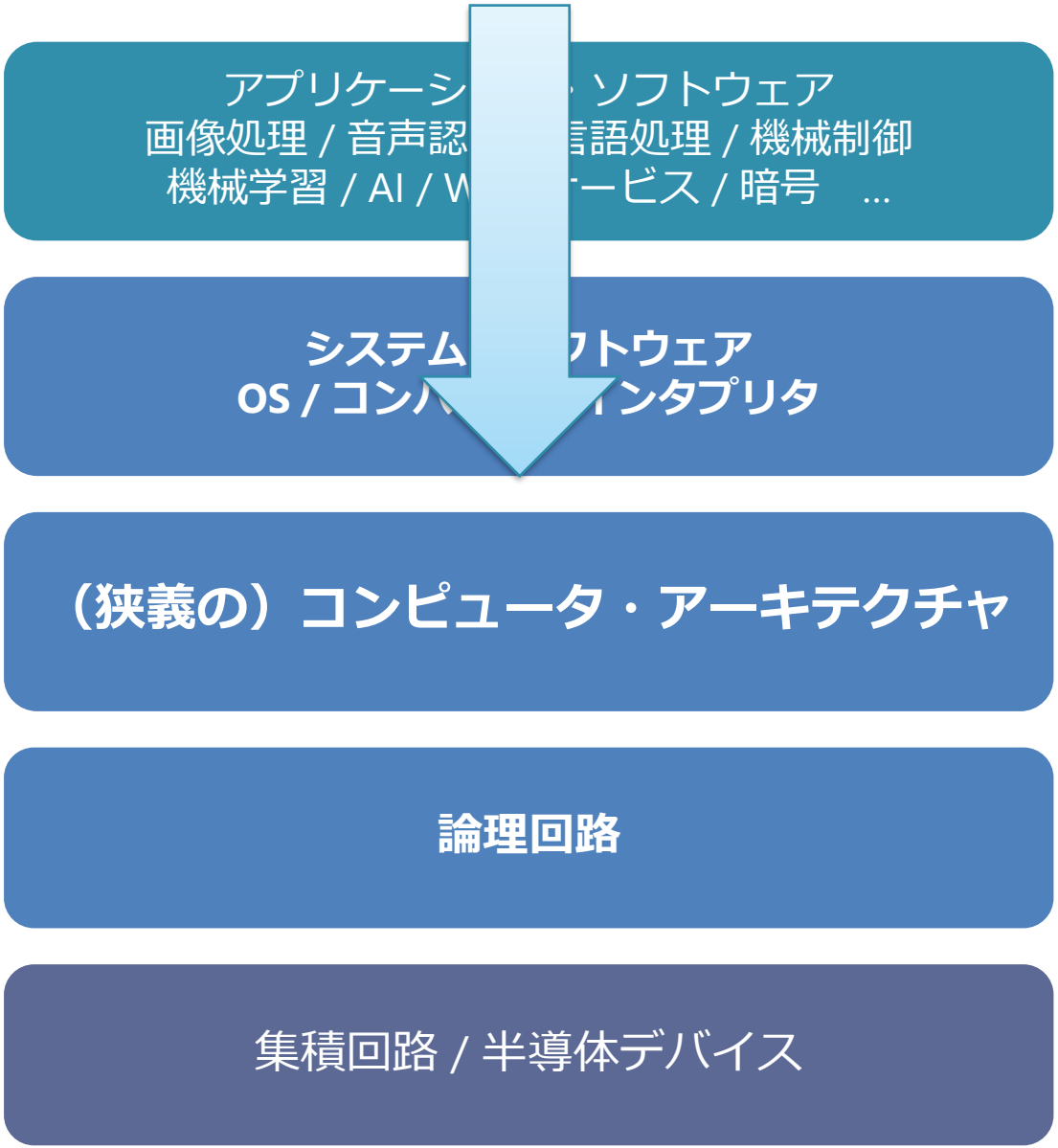
回路と遅延

- 目的：これらの具体的なイメージを持つ
  - CPU の論理的な動作と，それを実現する回路の繋がり
  - （それら論理回路の遅延や消費エネルギー

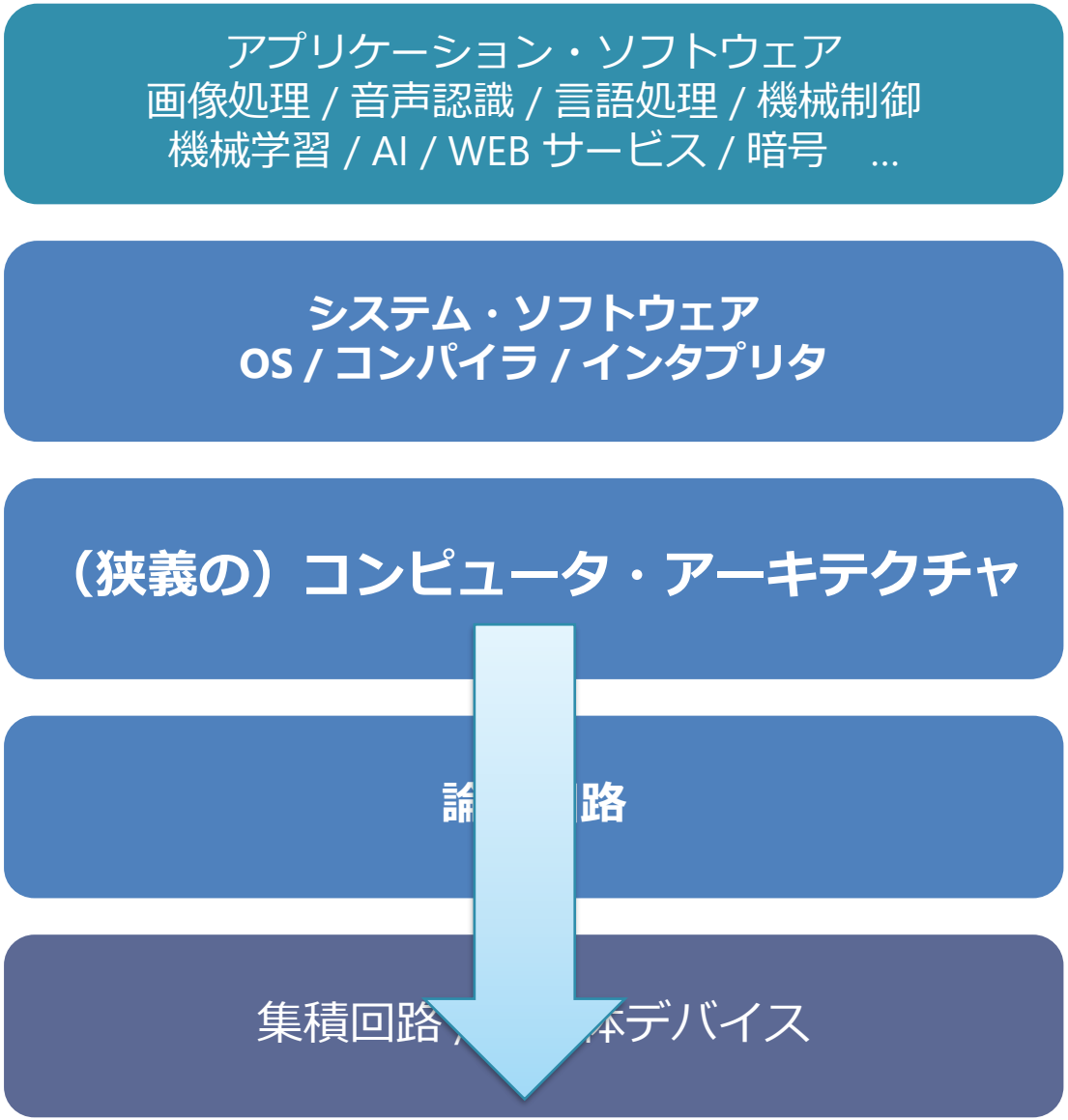
- 論理回路の復習から始めて説明
  - 論理回路

（次回以降：
  - CMOS による実現
  - 遅延と消費電力がどのように決まるのか

前回は，「C 言語で書かれたプログラムを動かすためには」という視点で上から迫っていた



今回は，「コンピュータのハードを作るためには」という視点で，さらに下がっていく

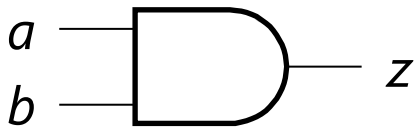
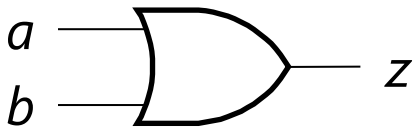
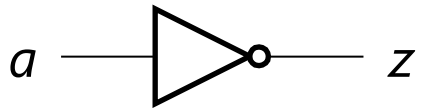


論理回路の復習

組み合わせ回路と順序回路

1. 組み合わせ回路
   - 出力が，現在の入力のみにより決定される論理回路
2. 順序回路
   - 出力が，過去の入力（の履歴）にも依存する論理回路

組み合わせ回路の例：2入力論理ゲート

	AND （論理積）	OR （論理和）	NOT （論理否定）
MIL記号 MIL symbol			
論理式 logic expression	$z = a \cdot b$	$z = a + b$	$z = a'$ $z = \overline{a}$ $z = \neg a$
真理値表 truth table	a b z 0 0 0 0 1 0 1 0 0 1 1 1	a b z 0 0 0 0 1 1 1 0 1 1 1 1	a z 0 1 1 0

完全性（Completeness，完備性）

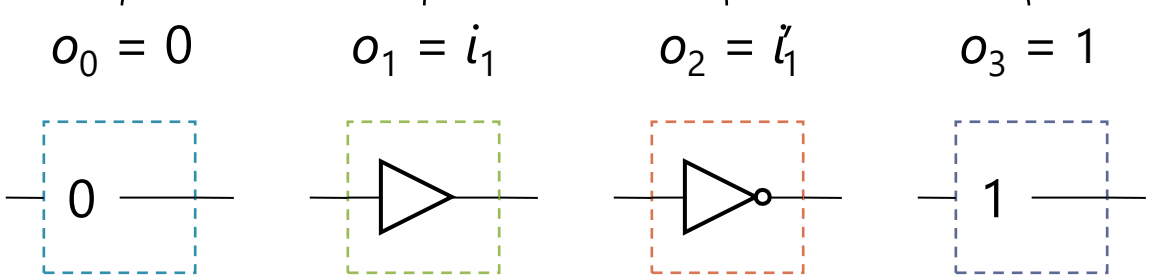
- 完全集合 (Complete Set)：
  - その組み合わせによって，すべての論理関数を表現できる論理関数の集合
- 完全集合の例
  - {AND, OR, NOT}
  - {AND, NOT}
  - {OR, NOT}
  - {NAND}
  - {NOR}
- たとえば，{AND, OR, NOT} を組み合わせると任意の論理関数が作れる

完全性の証明 {AND, OR, NOT}

- 数学的帰納法：
  1. 1入力の論理関数は {AND, OR, NOT} の組合せで表現できる
  2. n 入力の関数を {AND, OR, NOT} の組合せで表現できたと仮定して，$(n+1)$ 入力の関数が表現できることをいう

i_1	o_0	o_1	o_2	o_3
0	0	0	1	1
1	0	1	0	1

- i_1 =0 or 1 の入力に対して
  4通りの論理関数があり得る
- 4つとも NOT と数字直結で作れる

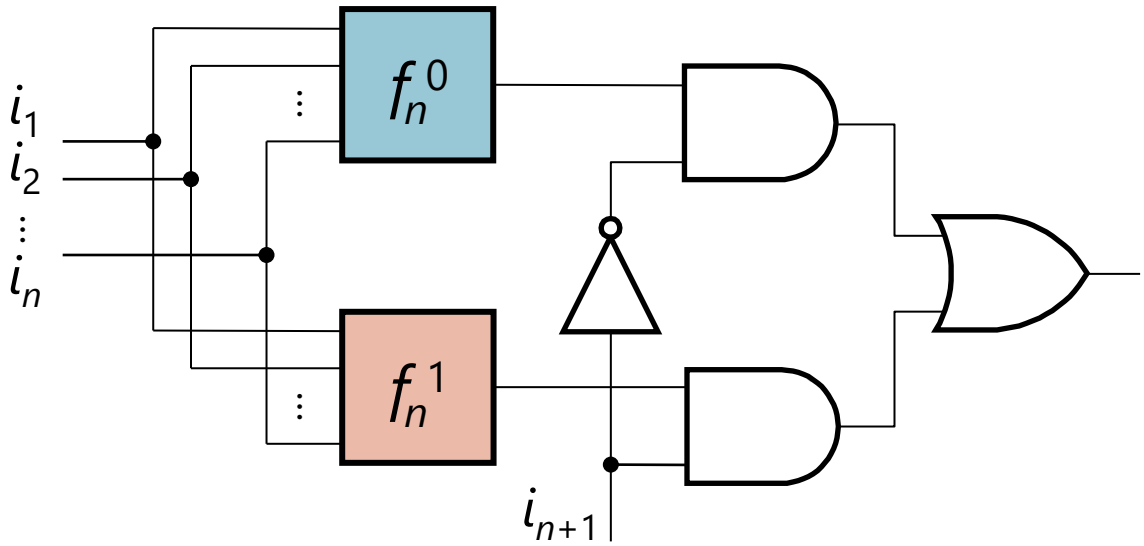


完全性の証明 {AND, OR, NOT}

- 数学的帰納法：
  1. 1入力の論理関数は {AND, OR, NOT} の組合せで表現できる
  2. n 入力の関数を {AND, OR, NOT} の組合せで表現できたと仮定して，$(n+1)$ 入力の関数が表現できることをいう

i_{n+1}	i_n	…	i_2	i_1	o
0	0	…	0	0	
	0	…	0	1	
	⋮	⋮	⋮	⋮	⋮
	1	…	1	1	
1	0	…	0	0	
	0	…	0	1	
	⋮	⋮	⋮	⋮	⋮
	1	…	1	1	

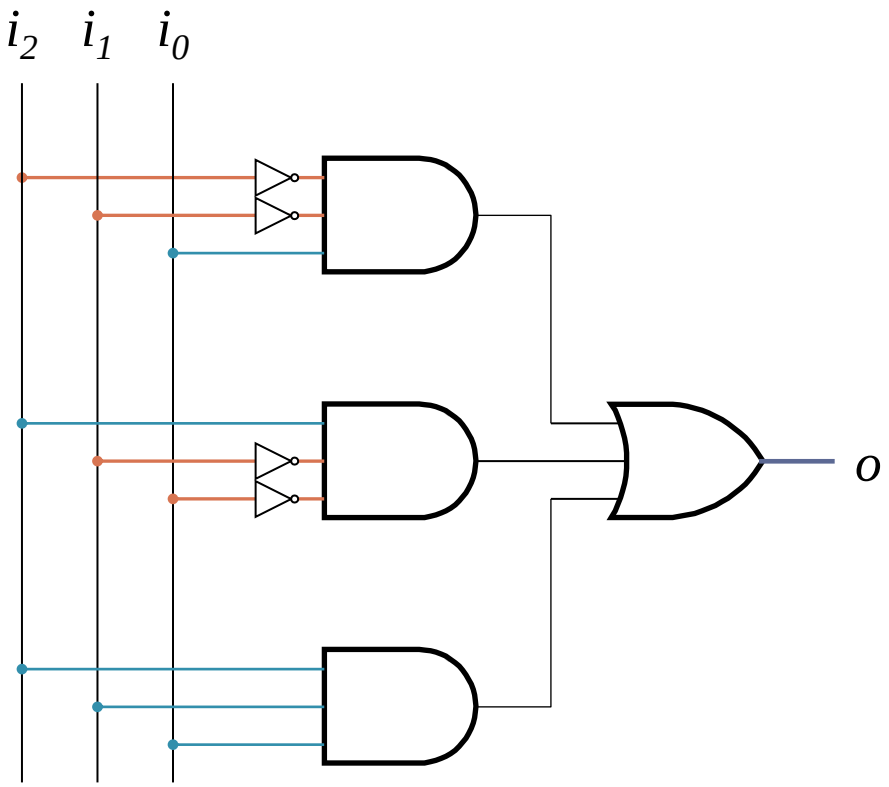
$$f = i'_{n+1} \cdot f_n^{\,0} + i_{n+1} \cdot f_n^{\,1}$$



- 任意の論理関数 $f_n^{\,0}$ $f_n^{\,1}$ が表現できるので，それらの結果をi_{n+1}を使って選択
- この選択部分は AND/OR/NOT で作れる

真理値表による表現と，積和標準系による回路

i_2	i_1	i_0	o
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



$$o = i_2'i_1'i_0 + i_2i_1'i_0' + i_2i_1i_0$$

- 真理値表が与えられれば，
  - {AND, OR, NOT} を使った積和標準系に機械的に置き換えできる
  - つまり，{AND, OR, NOT} を使って対応する回路が生成できる

回路の例：RISC-V の AND/OR/XOR 命令の演算

XOR: x[rd] ← x[rs1] - x[rs2]

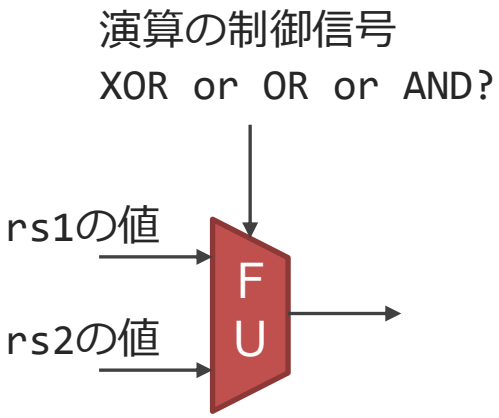
OR: x[rd] ← x[rs1] - x[rs2]

AND: x[rd] ← x[rs1] + x[rs2]

- RISC-V の AND/OR/XOR 命令
  - まず右端の opcode が 0110011 であれば，この 3 つのどれかということにする
    - 本当はほかの命令との識別がさらにあるが，ここでは忘れる
  - 真ん中の赤い3ビットの違いで識別する

制御の例：RISC-V の AND/OR/XOR 命令

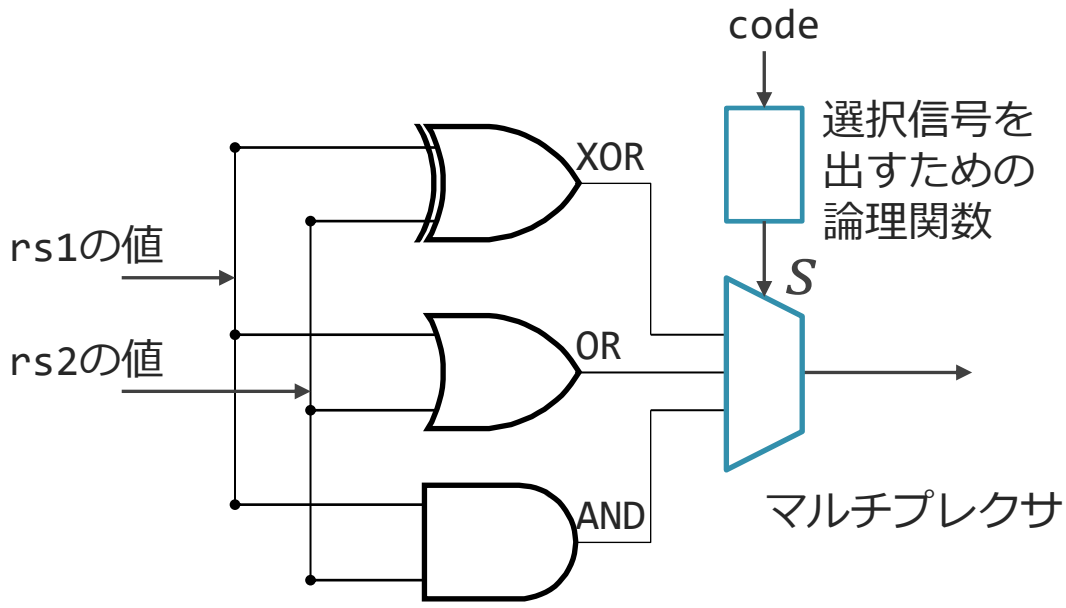
```
if code == 0b100:
    rs1 xor rs2
elif code == 0b110:
    rs1 or  rs2
else:
    rs1 and rs2
```



- 各命令の 3bit の code の違いに応じて
  - 演算器に，AND/OR/XOR 演算をさせることを考える

制御の例：RISC-V の AND/OR/XOR 命令

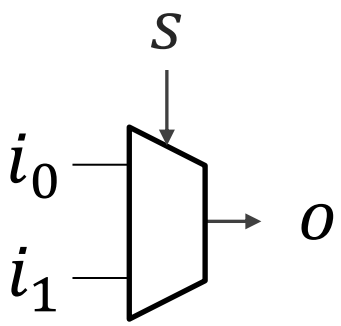
```
if code == 0b100:
    rs1 xor rs2
elif code == 0b110:
    rs1 or  rs2
else:
    rs1 and rs2
```

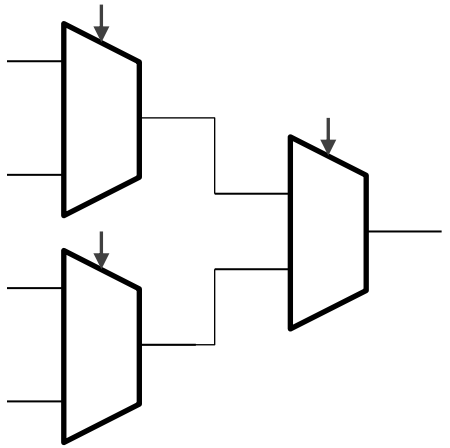


- 典型的には,
  - 各場合ごとの回路を用意して並列に配置
    - XOR, OR, AND ゲートを並べる
  - 制御に従ってマルチプレクサで出力を選択

マルチプレクサ：複数入力から1つを選ぶ回路

s	i_0	i_1	o
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

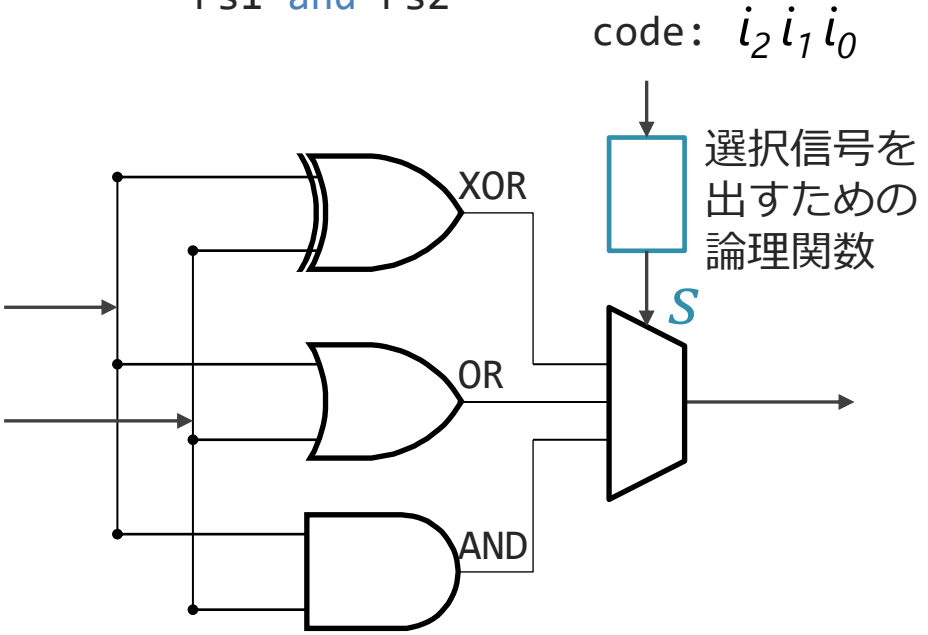




- 以下により，回路が生成できる
  - ２：１マルチプレクサは真理値表でかける = 回路が作れる
  - 多入力マルチプレクサは，カスケードすれば良い

選択信号を出すための論理関数

```
if code == 0b100:
    rs1 xor rs2
elif code == 0b110:
    rs1 or  rs2
else:
    rs1 and rs2
```

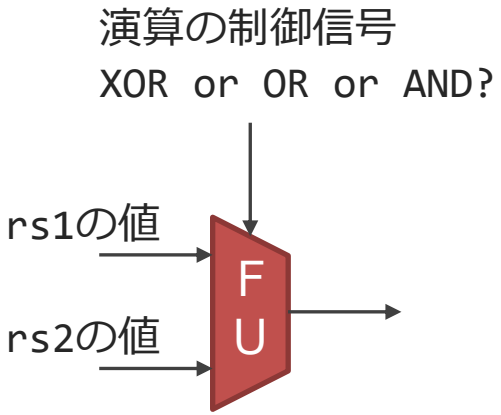


	i_2	i_1	i_0	s_1	s_0
	0	0	0	x	x
	0	0	1	x	x
	0	1	0	x	x
	0	1	1	x	x
XOR	1	0	0	0	0
	1	0	1	x	x
OR	1	1	0	0	1
AND	1	1	1	1	0

- 場合わけの制御は，そのまま真理値表にして回路を生成すればよい
  - XOR なら 00，OR なら 01，その他は 10

回路の生成のまとめ

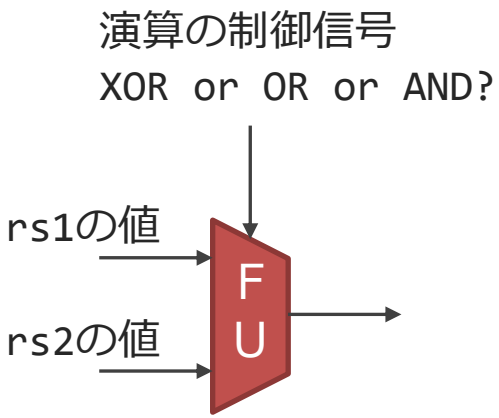
```
if code == 0b100:
    rs1 xor rs2
elif code == 0b110:
    rs1 or  rs2
else:
    rs1 and rs2
```



- 結局この手の，「条件に応じて異なる演算を出力する回路」は，
  1. 各場合ごとの回路を用意して並列に配置
  2. 制御に従ってマルチプレクサで出力を選択
  - ･･･というように分解すれば，AND/OR/NOT 回路に落とし込める

回路の生成のまとめ

```
if code == 0b100:
    rs1 xor rs2
elif code == 0b110:
    rs1 or  rs2
else:
    rs1 and rs2
```

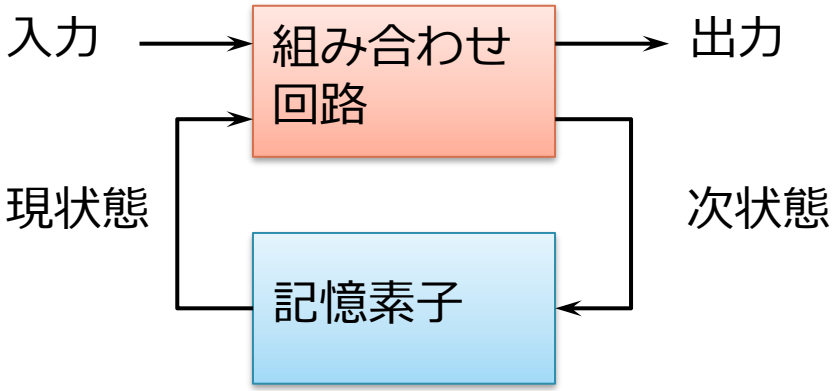


- 原理的には，code, rs1, rs2 を全て含む真理値表を作れば，そこから直接回路に落とし込むこともできる
  - 表が大きくなりすぎて (2 の 3+32+32乗)，現実的には無理

組み合わせ回路と順序回路

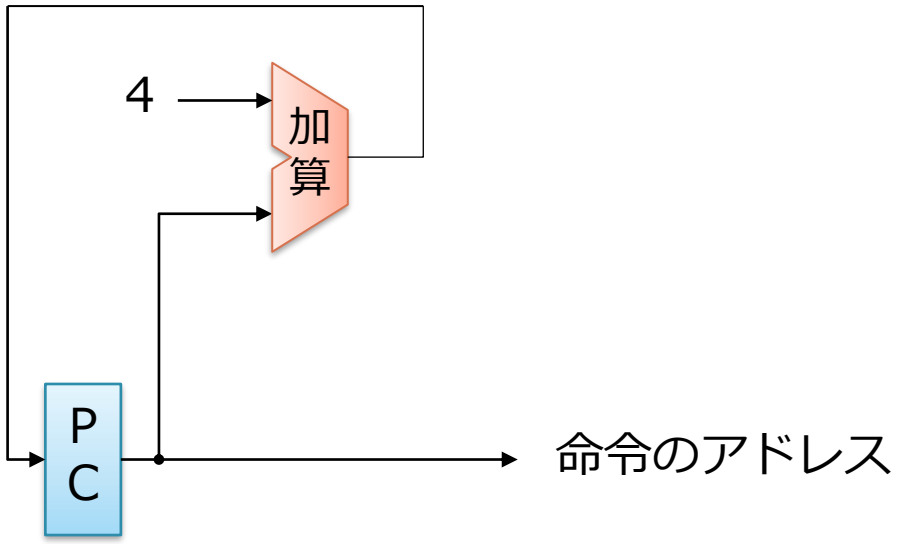
1. 組み合わせ回路
   - 出力が，現在の入力のみにより決定される論理回路
2. 順序回路
   - 出力が，過去の入力（の履歴）にも依存する論理回路

順序回路



- 出力が，過去の入力（の履歴）にも依存する論理回路
  - 記憶素子と，組み合わせ回路から成る

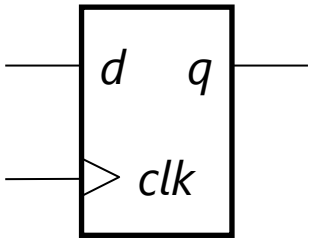
CPU の PC 部分



- 毎サイクル PC が加算される部分は，典型的な順序回路

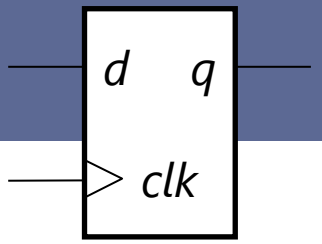
記憶素子の例：D-FF（Flip Flop）

- 入出力端子
  - ◇ データ入力　　：*d*
  - ◇ データ出力　　：*q*
  - ◇ クロック入力：*clk*

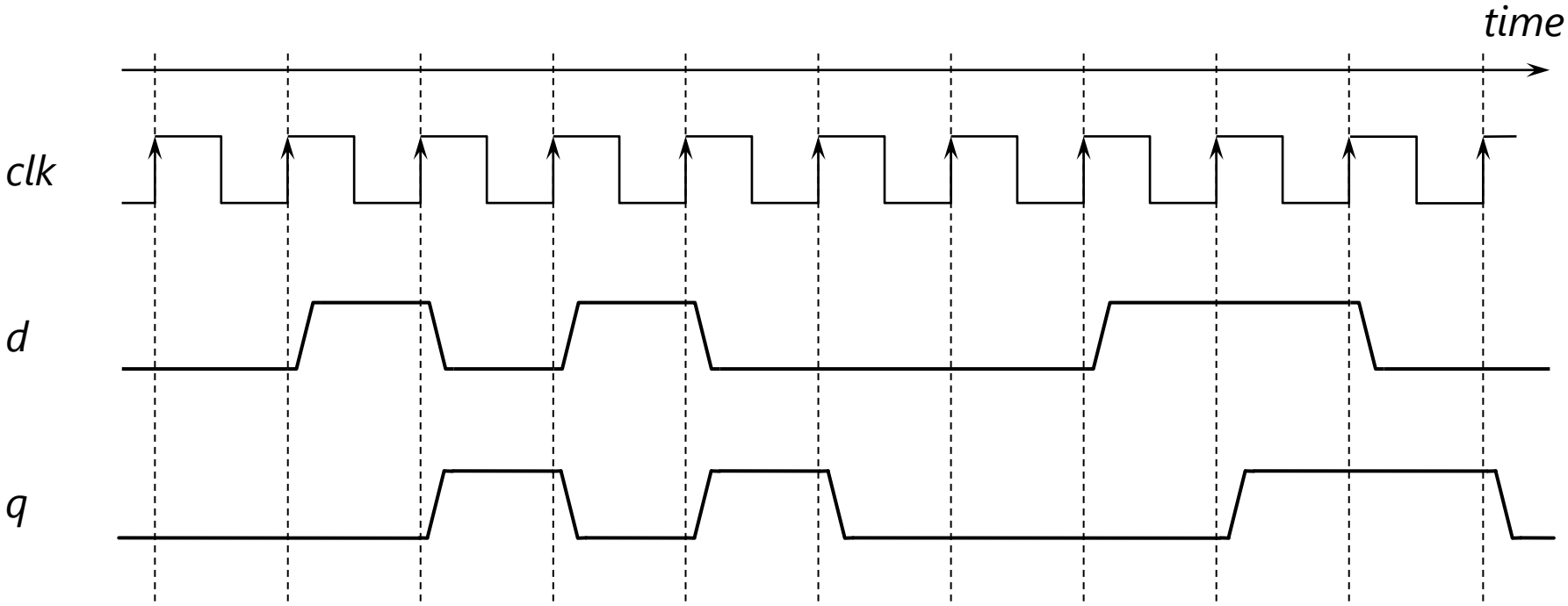


- 働き：
  - ◇ クロックの立ち上がりのたびに，*d* の値がサンプリングされる
  - ◇ その値が次のサイクルの間 *q* から出力される

D-FF の動作

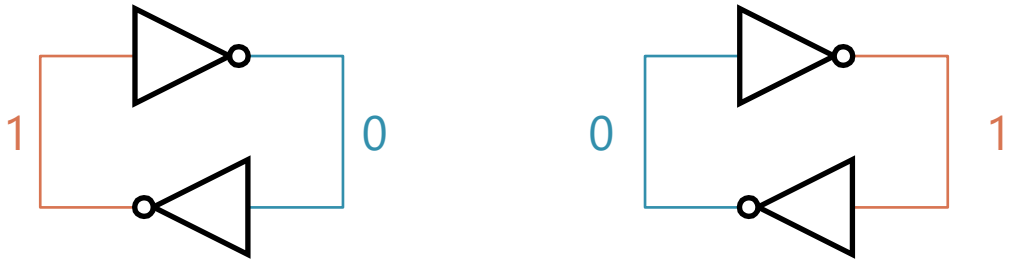


d	0	1	0	1	0	0	0	1	1	0
q	0	0	1	0	1	0	0	0	1	1



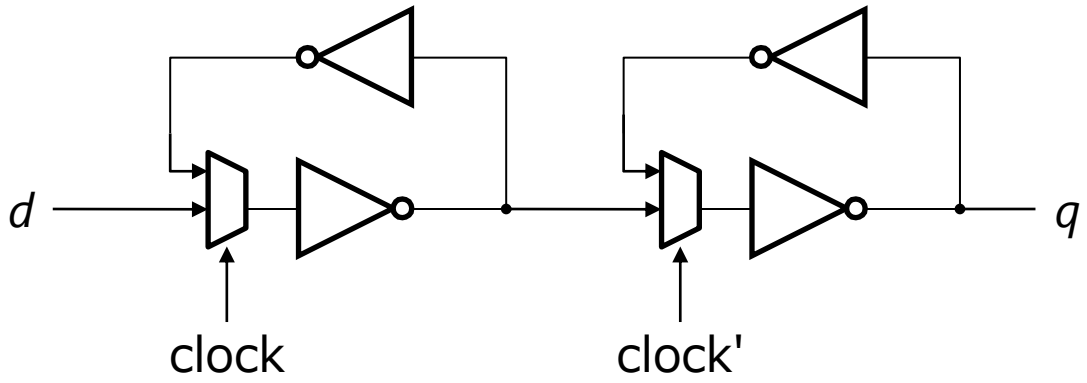
記憶素子の原理

- 記憶
  - 2 つの NOT ゲートをループさせた回路により記憶
  - 2 通りの安定状態がある：1 bit を記憶



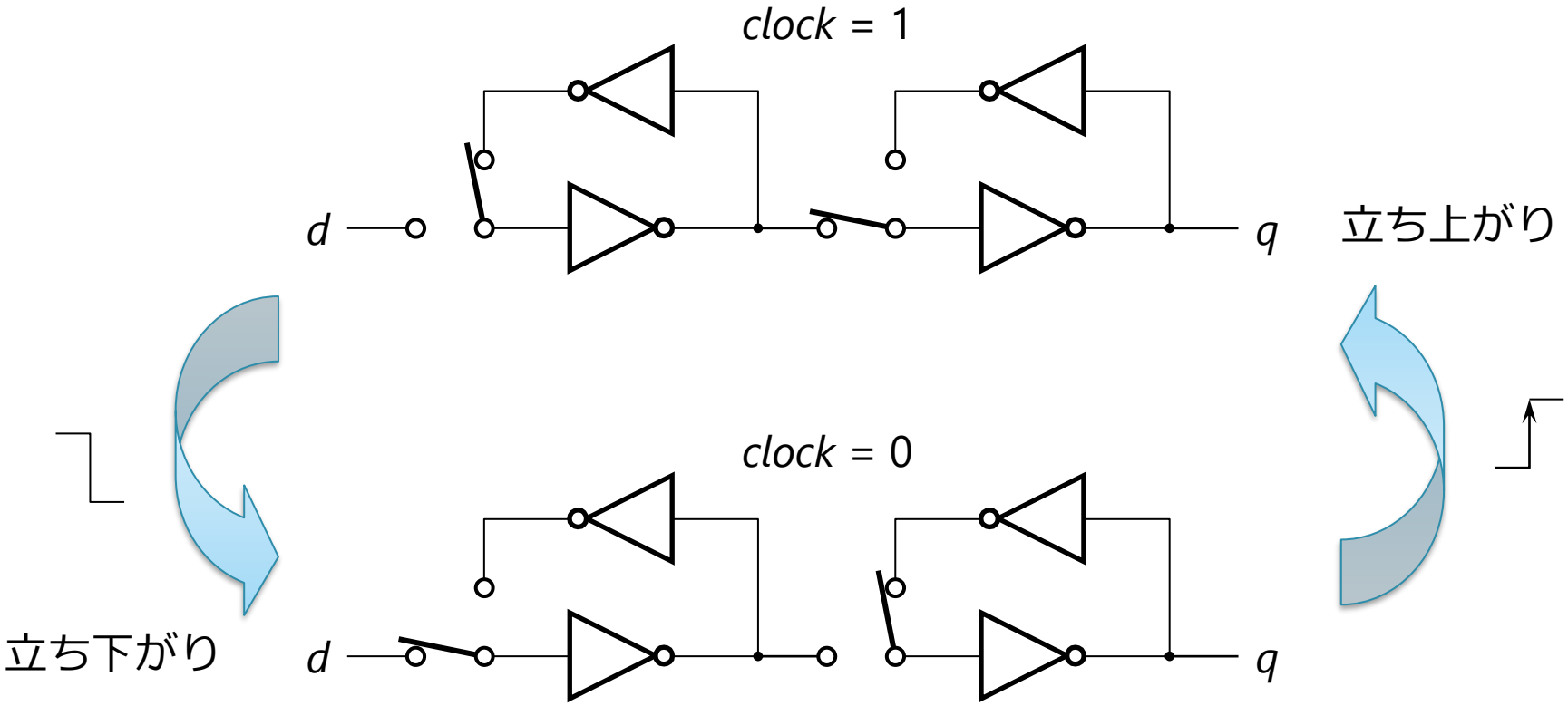
D-FF の実装

- 構造：
  - ◇ NOT ゲートのループを二段に接続
  - ◇ 各ループにはマルチプレクサが入っている
- この構造は，クロックのエッジで記憶を更新するため

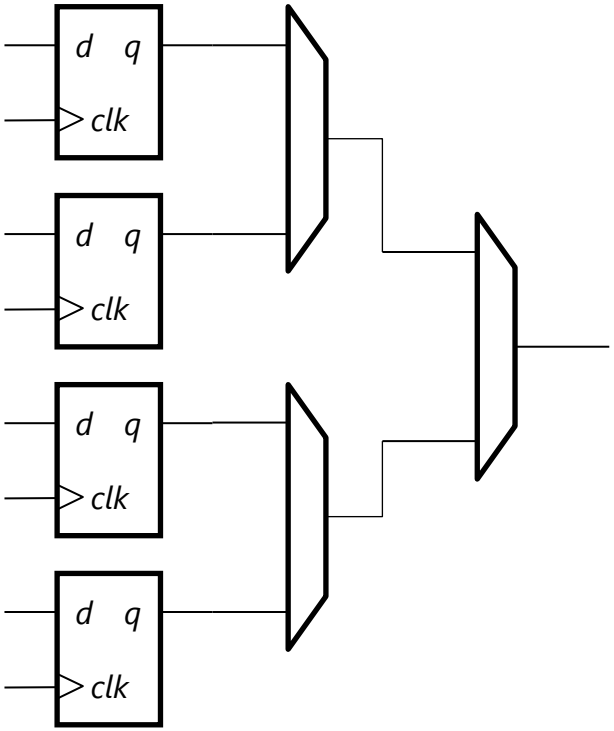


D-FF の実現例

- マルチプレクサを，切り替えスイッチとして説明
  - ◇ クロックの立ち上がりのたびに，*d* の値がサンプリングされる
  - ◇ その値が次のサイクルの間 *q* から出力される



メモリやレジスタ・ファイル

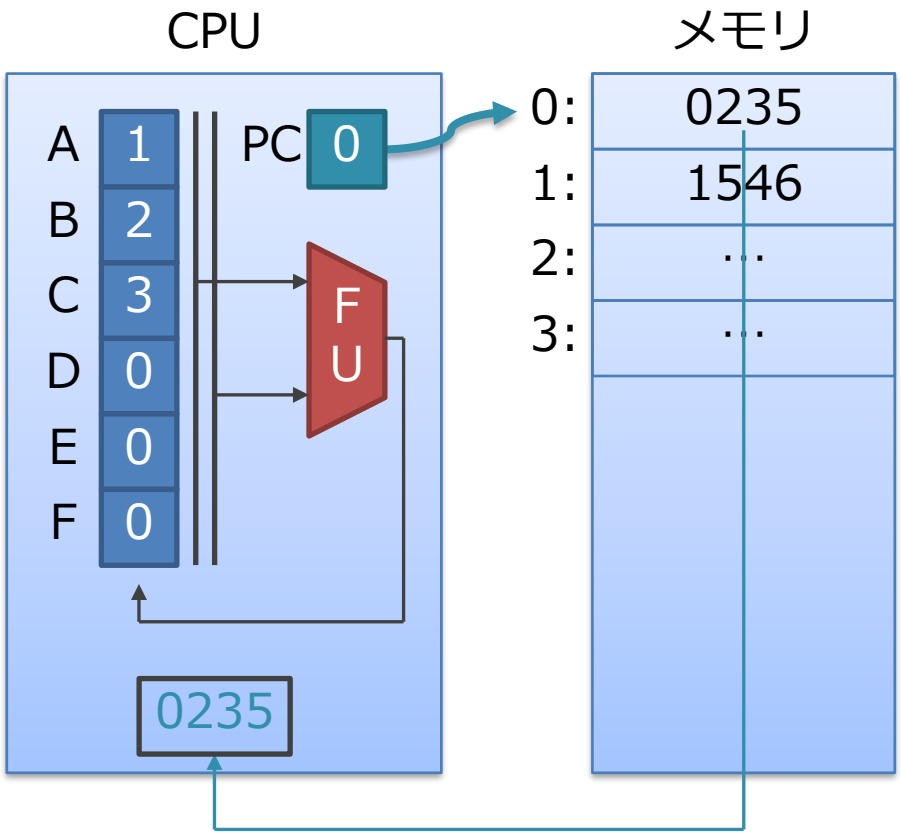


- D-FF を必要なだけ並べて，マルチプレクサで選択することで実現できる
  - 実際には，もっと最適化された回路（SRAM）が使用される事が多い
  - （後の講義で詳しく説明する予定

組み合わせ回路/順序回路のまとめ

- 記憶素子も，NOT ゲートなどの論理ゲートで構成される
- 結果として，任意の組み合わせ回路/順序回路は
  - 原理的には，完全集合の要素の組み合わせに落とし込める
  - たとえば {AND, OR, NOT} を組み合わせれば，全部つくれる

これまでに説明した CPU の要素は，全てこれでカバー可能



まとめ

1. ２進数や１６進数による数値表現
2. 論理回路による実装
   - CPU の論理的な動作と，それを実現する物理的な回路の繋がり

- この講義資料では（ていうか，今後の資料も結構），一部，五島先生の「ディジタル回路」の講義資料の図を使用しています

課題

1. 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
2. 以降で説明する課題を「課題」に提出してください
   - 課題は自力でがんばって提出することに意義があります
   - 途中で力尽きてもある程度頑張った様子があれば，別に正解していなくても満点がつきます

- 締め切りはそれぞれ来週の講義開始です

課題３

- 以下では 2進数は 0b... 16進数は 0x... と表記するものとする
- 以下の１６進数を２進数で表記せよ
  - (1) 0x1010
  - (2) 0xab1234cd
- 以下の２進数を１６進数で表記せよ
  - (3) 0b111010
  - (4) 0b111111111111
- 以下の演算を行え
  - (5) 0b10110110 / 0b10 (「/」 は割り算)

感想や質問

質問や感想など

- アセンブリ言語が難しかったです。特に、条件分岐命令のbがどのようなものかわからず苦戦しました。また、LABELも初めて見たものだったので難しかったです。

  - 料理のレシピに無理矢理あてはめると，
    「温度が～度以下の時は，『3. ぬるま湯で暖める』に戻る」とか
    「すでによくまざってる場合は，『5. 盛り付け』に進む」
    みたいな，条件がなり立ってる時に特定の手順に移るイメージです
  - LABEL は，レシピの途中に張れる付箋やシールのイメージです
    - 印をつけておくと，そこを指すときにわかりやすい

質問や感想など

- 「li 0→A」と、「li A←0」は同じことを表しているということでいいのでしょうか。書き方はどちらでもいいのですか？
- li 0 → Aはその後加算してAにその値を代入するときなどにli A←0となるのですか？それとも全く同じ意味でしょうか？
  - おなじ意味です

質問や感想など

- 課題2を取り組んでいるときに気になったのですが、条件分岐命令が出てくる前にLABELを定義しないといけないのでしょうか。それとも条件分岐命令の下にLABELを定義する式が出てきてもいいのでしょうか。
  - 定義がある前に LABEL を使っても構いません

質問や感想など

- スライドのp.53で、0x408までの命令を行った後に、次のメモリにあるLABEL部分は1度実行されてしまうのか、それともLABEL部分は飛ばされて b a<B , LABELの後にLABELが実行されるのか、を疑問に思いました。i = 0 のときLABELが先に実行されてしまったら、i = 1 となってしまうので、LABELは飛ばされるのかとも思いましたが、命令はメモリの上から順に実行されるので、LAVEL部分が先に実行されてしまうのかなとも思いました。

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
  （初回の範囲チェックはややこしいのでないものとする）

```
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
```

全体

```
1:      i = 0;
        0x400: li 0      → A    // レジスタ A に 0 をいれる
        0x404: li 0x0f4 → B    // B に 0x0f4 (i の番地) をいれる
        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) をいれる
        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
```

質問や感想など

- C言語を勉強していた時、C言語はほかの言語（RubyとPythonをちょっと触っただけですが）と比べて読みにくい、難しいなと感じていたのですが、その理由が機械語に近い言語だったからだと今日分かりました。機械語の方がC言語より手順が細かく書かれているので一見分かりやすそうにも思いますが、実際には読みにくいというのが面白いです。

質問や感想など

- 手順を辿れば意外と簡単にC言語からアセンブリ言語に直すことができた

質問や感想など

- 実際の命令セットの例に入った途端に全然分からなくなった…。

質問や感想など

- メモリに変数を割り当てるとき、授業では適当に空いているところを選んでいましたが、実際には上から順番に埋めていきますか？
  - 一定の領域を上から順に埋めていくことがおおいです

質問や感想など

- 容量が大きくても高速なものは作れないのでしょうか。

質問や感想など

- コンピュータの仕組みやメモリに関しての授業をさまざまな授業で何度も聞いてきたが、いまだになんとなくでしかわからなく、ふわふわとした理解のままです。とっても難しいですね。

質問や感想など

- アセンブリ語はcよりも直感的にわかりやすくて親しみが湧きました
- 今まではc言語は難しいと思っていましたが、アセンブリ言語を見てみたら、c言語はとても見て理解しやすい言語なのだなと思いました。
- C言語はすでに習ったからか、見たら英語の意味で内容を理解できるがアセンブリ言語はまだ直感的には理解できない。

質問や感想など

- データをとってくる時間の例がわかりやすく、時間の差のイメージを感じられました。思っていたよりも、データをとってくる時間はCPU、メモリ、ハードディスク、光学ドライブによって差が大きくて驚きました。何が速度に影響しているのか気になります。

質問や感想など

- 今まで、CPUやメモリの働きについてあまり理解しておらず、なんか難しいものだと思っていたので、今回の授業で改めて学習して、大まかな部分は意外と単純な動きをしているんだなと思いました。

質問や感想など

- 課題についてですが、
  LABEL: (実行文)
  b (条件) LABEL
  …

  と書くと、bで分岐してLABELに飛び命令を実行して、もう一度bで分岐して…とfor文のようにループしてしまうのではと思いましたが、他の書き方が思いつかないので、bでLABELに飛んで命令を実行したあと更にbが実行されることはないと仮定して課題を解きました。この仮定は正しいでしょうか。

質問や感想など

- 授業とは逸れるかもしれませんが、単純なCPUの構造において、パイプライン処理がどのように性能向上に寄与しますか？またその際に生じうる問題はありますか？

質問や感想など

- ポインタがすごく苦手なのですが、単純化して説明していただいたおかげで、メモリについての理解が少し深まった気がします。今回の授業は新しく出た単語も多かったですが、ついていけるよう頑張りたいです。

質問や感想など

- C言語はコンパイラ言語なので機械にとって結構わかりやすいと思っていたからアセンブリ言語というのを知ってさらにバイナリに近い表現があるんだと驚きました。アセンブリ言語はC言語のように習得する必要があるのですか？（アセンブリ言語でプログラムを実行することができるのか）

質問や感想など

- if～gotoさえあればC言語の制御文書き換えできるとありましたが、それだけで書かないのはどうしてでしょうか。単に視認性のためですか？
  パターンマッチングとは具体的にどういうことでしょうか。

質問や感想など

- 0にaddという意味があったり1にsubという意味があるのは、数字に意味があるという考え方が新鮮で面白かったと同時にやはりコンピュータの世界では0と1は重要な意味を持つ数字なんだと思った。

質問や感想など

- 「C言語の制御構文のほとんどがif～gotoで置き換えられる」という記述にいささか衝撃を受けています。納得はできるのですが……いざ字面で表されるとなかなかのパワーを感じました。

質問や感想など

- C言語やPythonのプログラムをアセンブリ言語で表すと同じ表現になりますか？

質問や感想など

- アセンブリ言語はemacsで動かせますか？