先進計算機構成論 02

東京大学大学院 情報理工学系研究科 創造情報学専攻
塩谷 亮太
shioya@ci.i.u-tokyo.ac.jp

質問と回答

- addとaddiの違いのところで、immediateを大きくとるために設定されていると説明されていましたが、オペランドで差された場所のビット数と違うとプログラムを書く上で何か不便が生じたりしないのでしょうか。
  - ◇ 特に不都合はない
  - ◇ このあたりのビットは幅は通常アセンブラが全自動で計算してくれるので，人間が意識することもほとんどない

ADD と ADDI の違い

ADDI : x[rd] ← x[rs1] + **immediate**

ADD : x[rd] ← x[rs1] + x[**rs2**]

SUB : x[rd] ← x[rs1] - x[rs2]

- immediate の部分はな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

質問と回答

- RISC-Vの説明を聞いていたときに思ったのですが、自分で命令セットを作ろうとして命令幅が決まっているにopcodeにより多くのビット数を割くことは可能なのか気になりました。
  RISC-Vの場合には32bitでR-typeでは17bitをopcodeとして残りの15bitをオペランドとしていました。
  CISCのような命令セットを考えたときにより多くの命令を定義するために17bit以上をopcodeとして割くというような設計方法は可能でしょうか"

  - ◇ やってもいいが，2^17 = 131,072 個なので，普通は足りる
  - ◇ むしろオペランドが足りない場合がある
    - □ ARM SVE（スーパーコンピュータの富岳のやつ）では 4 オペランド命令がある
      - ✳ rd = rs1 + rs2*rs3
    - □ ２つの命令（オペランド設定+演算）に分けて実現

質問と回答とか

- 時間のあるときにRISCVのエミュレータ作成に挑戦してみたい気持ちを抱きました。
  - だいたい以下のような感じで作れる
    - たぶん RV32I 対応なら1000行いらない

```
while(true) {
    code = binary[pc++];
    switch(get_opcode(code)) {
    R_TYPE:
        switch(get_funct7(code)) {
        ADD: reg[get_rd(code)] = reg[get_rs1(code)] + reg[get_rs2(code)]; break;
        }
        break;
    I_TYPE_ADDI:
        ...
    }
}
```

  - バイナリを得るには riscv 用 gcc でコンパイル →
    objdump コマンドでバイナリをテキストで出力 →
    スクリプトかなんかで加工して配列で定義　とかすると比較的楽

質問と回答とか

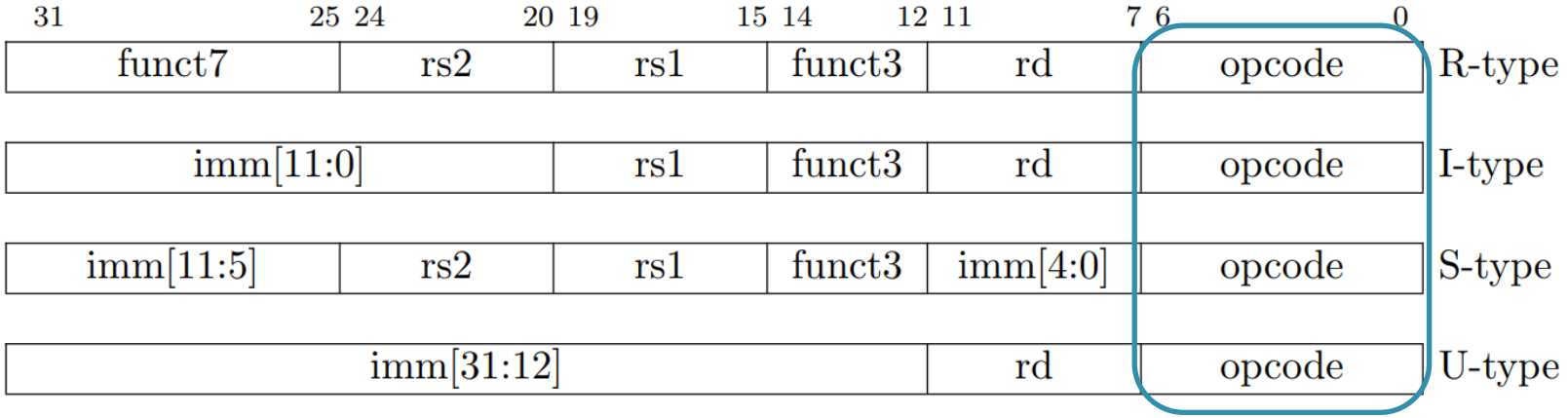
- 今日の内容は学部で習った内容でしたので、理解できました。自作PCを作っているので、よりPCに対する理解を深めたいです。
- ゲームボーイアドバンスが好きで、多少それ用のアセンブリ言語を触ったことがあったので、今日の内容はほぼ理解できました。

前回の続き：RISC-V について

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



画像は下記より
The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2

RISC-V のロードとストアの違い

```
LW : x[rd] ← (x[rs1] + immediate)
```


```
SW : (x[rs1] + immediate) ← [rs2]
```

- どちらもレジスタのオペランドは２つ
  - ◇ しかし，使用するビット位置が違う
    - □ LW: rd, rs1　/　SW: rs1, rs2
  - ◇ SW では immediate が [11:5] と [4:0] に分断されている
- これの説明が大ざっぱすぎたので，補足したい

RISC-V と MIPS のロード/ストア命令

- RISC-V

LW : x[rd] ← (x[rs1] + immediate)

SW : (x[rs1] + immediate) ← [rs2]

ADD : x[rd] ← x[rs1] + x[rs2]

- MIPS

LW : x[rt] ← (x[rs] + immediate)

SW : (x[rs] + immediate) ← [rt]

ADDU : x[rd] ← x[rs] + x[rt]

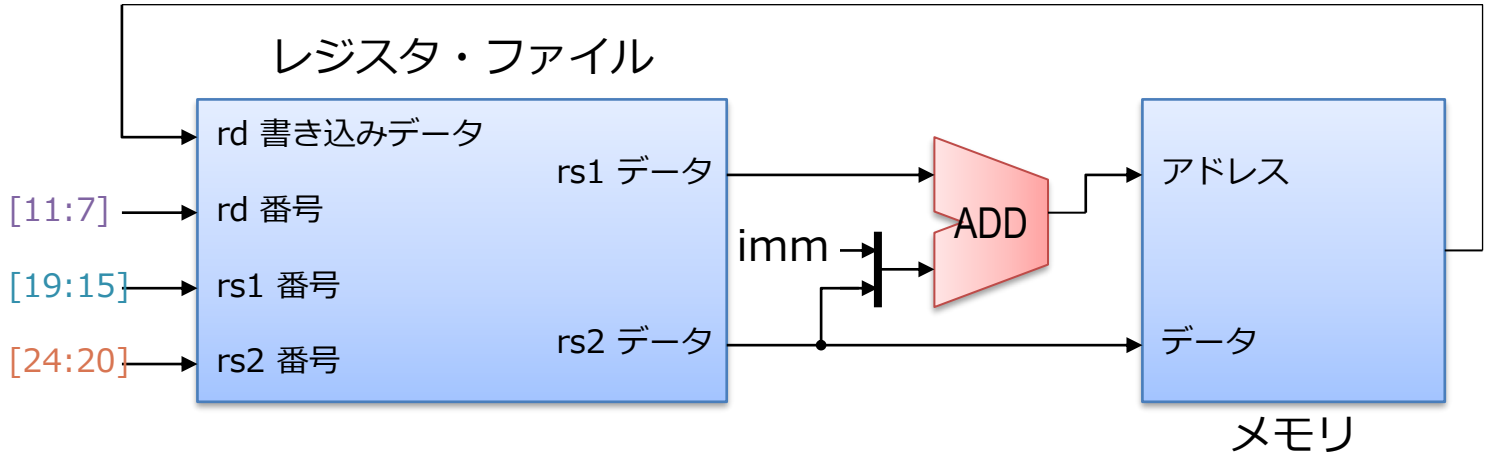
- MIPS の方が一見きれいにも見えるが・・・

RISC-V の場合

- RISC-V

```
LW :  x[rd] ← (x[rs1] + immediate)
SW : (x[rs1] + immediate) ← [rs2]
ADD:  x[rd]  ← x[rs1] + x[rs2]
```

- レジスタ番号を指定する，命令内のビット位置が一貫している
  - ◇ LW ：書き込み rd [11:7]　　読み出し：rs1 [19:15]
  - ◇ SW ：　　読み出し：rs1 [19:15]，rs2 [24:20]
  - ◇ ADD：書き込み rd [11:7]　　読み出し：rs1 [19:15]，rs2 [24:20]

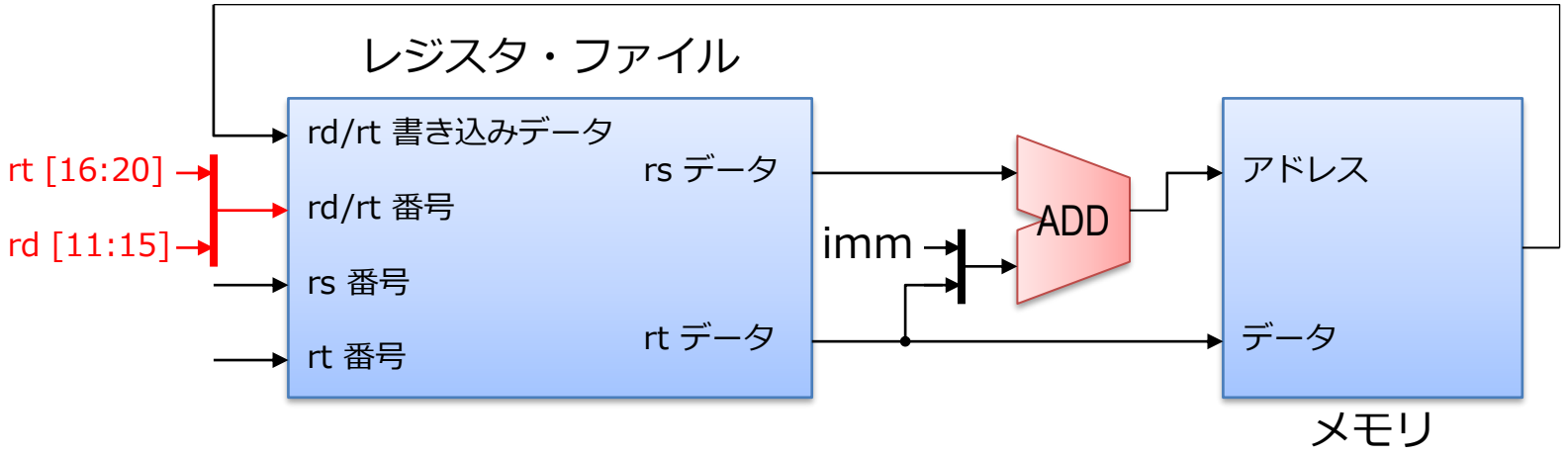


MIPS の場合

- MIPS

```
LW : x[rt] ← (x[rs] + immediate)
SW : (x[rs] + immediate) ← [rt]
ADDU : x[rd] ← x[rs] + x[rt]
```

- レジスタ番号を指定する，命令内のビット位置が一貫していない
  → 書き込み番号の切り替え（rt or rs）が必要
  - ◇ LW ：書き込み rt [16:20]　　読み出し：rs [25:21]
  - ◇ SW ：　　読み出し：rs [25:21]，rt [16:20]
  - ◇ ADD：書き込み rd [11:15]　　読み出し：rs [25:21]，rt [16:20]



RISC-V と MIPS のロード/ストア命令

- RISC-V のロード/ストアのエンコーディング
  - フォーマットだけみると，一貫していないように見える
  - 回路を考えると，このエンコーディングの方がよい
    - MIPS との比較により説明

RISC-V と MIPS のロード/ストア命令

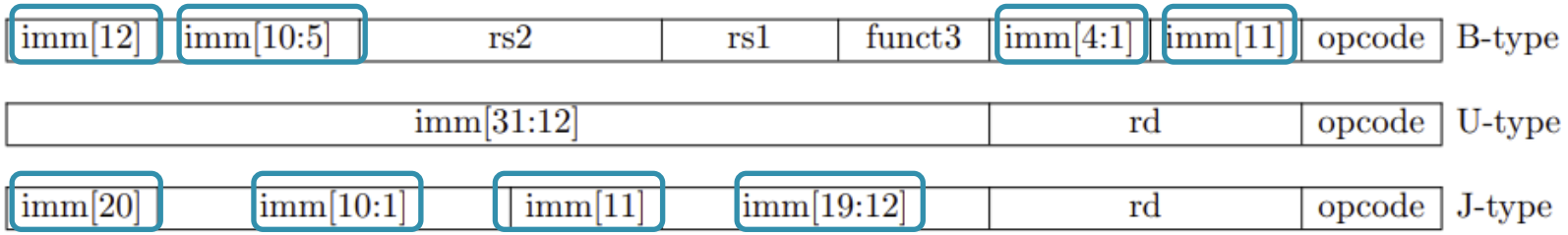
LW : x[rd] ← (x[rs1] + immediate)

SW : (x[rs1] + immediate) ← [rs2]

- 疑問：RISC-V は，その代わりに immediate 部分で選択がいるのでは？
  - ◇ ロードとストアで immediate 部分のフォーマットが異なる
  - ◇ 同様の選択が必要
- しかしこれは，レジスタへのアクセス中に並列してできる
  - ◇ MIPS のレジスタ番号の選択は，処理に直列に入ってしまう

RISC-V のエンコーディング

- この手の過去の反省や，細かい工夫がたくさん練り込まれている
  - ◇ 他の例：分岐命令の即値のビット配置が飛び飛び
    - □ 一見，頭がおかしいとしか思えない
    - □ 実際，自分で実装すると，よくバグの元になる
  - ◇ しかしこれはちゃんと意味がある
    - □ さっきの LW/SW の例のように，選択のための回路が減る
    - □ マニアックなのでこの講義では割愛

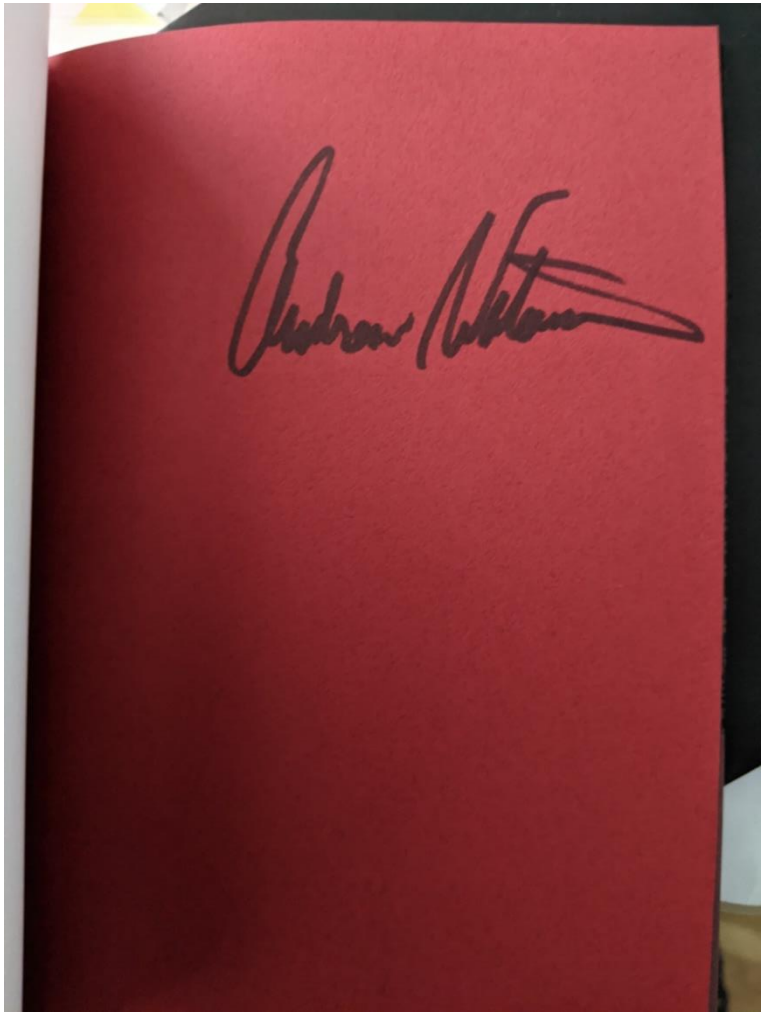


RISC-V

- Andrew Waterman,
  「Design of the RISC-V Instruction Set Architecture」
  https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf

  - ◇ RISC-V の仕様を主に決めた人達の中の１人の博士論文
  - ◇ いろいろ書かれているので，一部でも読んでみると面白いと思う
    - □ なぜ RISC-V は，そうなっているのか
    - □ x86 や ARM はここがクソ
  - ◇ さっきの分岐のエンコーディングについても説明されてるので，興味がある人は読んでみると良い

前に東大にきてたので，サインもらった



余談：RISC-V の何がうれしいのか？

- 企業サイド：
  - ARM とかに金払いたくねぇ
  - 自由にカスタムした CPU を作りたいが，OS や コンパイラとかを全部自分で作るのはしんどい
    - カスタム：専用の命令の追加とか
    - RISC-V ベースであれば，既存のインフラを使いつつ自由にカスタムできる

余談：RISC-V の何がうれしいのか？

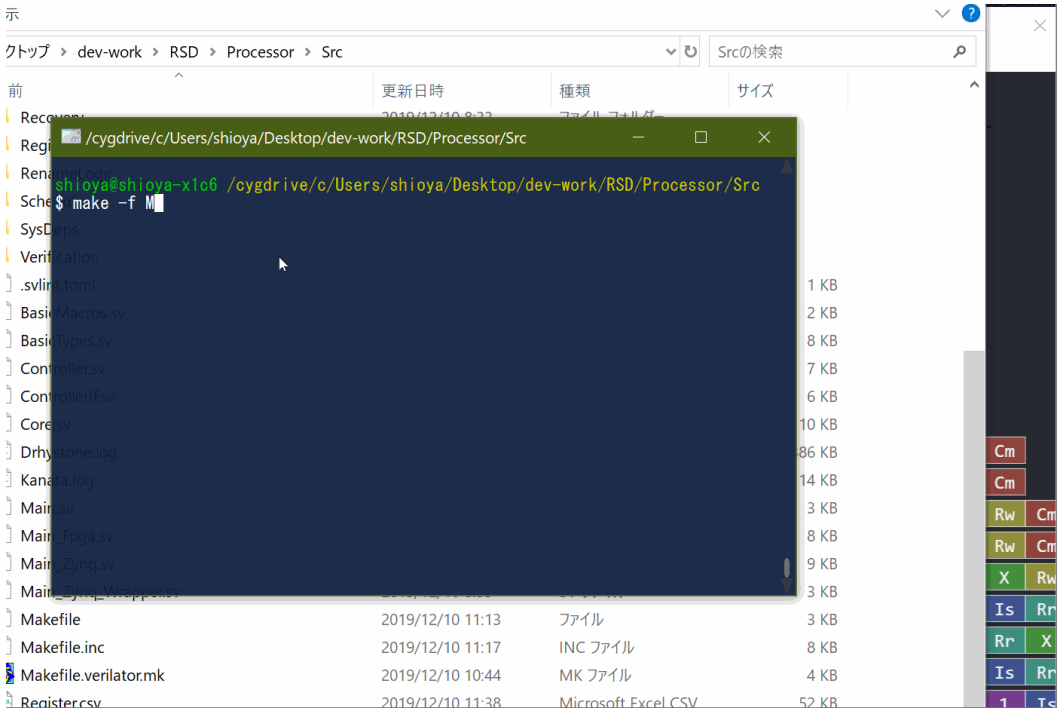
- 研究者サイド：
  - 研究がしやすい
    - 研究で使うときはなるべく命令セットは単純であってほしい
      - シミュレータなどの実装が楽
      - 本質的ではない変なことに苦しまないで済む
        - x86 で研究はマジで死ねる
    - いま広く使われている最先端の CPU は大概どれもかなり複雑
      - x86, ARM, POWER …
      - 最新 OS やコンパイラを使いつつ，実際のアプリなどの評価をしようと思うと大変
    - RISC-V であれば単純な割に最新のインフラが使える
  - 作ったものが自由に公開できる
    - 人が作った設計も利用できる

余談：RISC-V の何がうれしいのか？

- 塩谷サイド：
  - 研究して作った CPU が公開できた
    - https://github.com/rsd-devel/rsd
    - 最初は ARM だったため公開しようとして怒られが発生したが，RISC-V に移植して公開
  - CPU の研究に，また注目が集まってありがたい
    - 以前：「インテルとか ARM のやつでいいんでないの？」
    - 現在：「『RISC-V』というものが熱いらしいやん？」
    - 実際，企業で新しく作るための共同研究の話もいくつか来たりして，ありがたい

余談：RISC-V の何がうれしいのか？

- 塩谷サイド：
  - ◇ 研究して作った CPU が公開できた
    - https://github.com/rsd-devel/rsd
    - 最初は ARM だったため公開しようとして怒られが発生したが，RISC-V に移植して公開



余談：RISC-V の何がうれしいのか？

- 塩谷サイド：
  - CPU の研究に，また注目が集まってありがたい
    - 以前：「インテルとか ARM のやつでいいんでないの？」
    - 現在：「『RISC-V』というものが熱いらしいやん？」
    - 実際，企業で新しく作るための共同研究の話もいくつか来たりして，ありがたい

先端計算機構成論 02

東京大学大学院 情報理工学系研究科 創造情報学専攻
塩谷 亮太
shioya@ci.i.u-tokyo.ac.jp

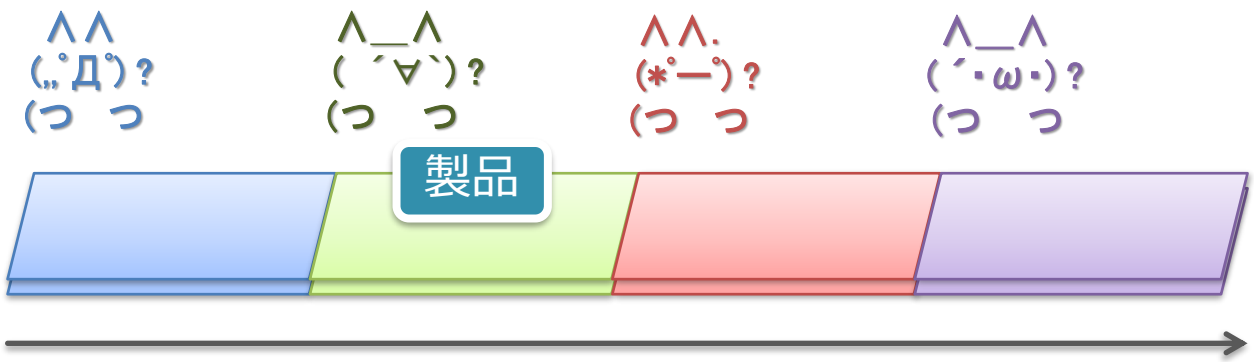
導入：工場のラインを考える

- ベルトコンベアのラインの上を製品が流れていく
  - ◇ ４人の人が，それぞれの工程の作業をおこなって完成
- 上のように1つしか製品をながさないと，
  - ◇ 各人は他の人が作業している間はヒマ

導入：工場のラインを考える

- 実際の工場：複数の製品を同時に流す
  - ◇ 各工程を並列して処理することによりスループットを向上
  - ◇ さっきの4倍の速度で製品ができあがっていく
- これが 命令パイプライン
  - ◇ ・・・ の話を今日はしようと思っていたのですが，

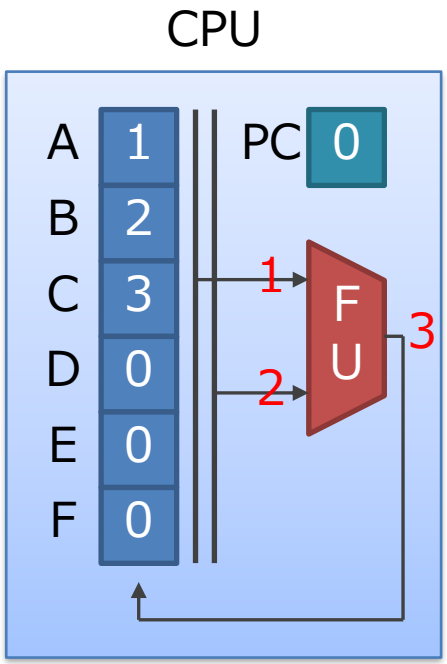
遅延に対する疑問



- 工場のラインの絵は，回路中を信号が伝搬されていくことのたとえ
  - 「製品」は，処理対象の信号（データ）
- そもそも「回路中を信号が伝搬される」のに「かかる時間」とは？
  - そりゃ，伝搬するのになんか時間はかかるのだろう
  - しかしそもそも回路が速いとか遅いとは，どういうことなのか？
  - 遅延は，「論理回路」の授業では理想化されていることが多い

回路に対する疑問

- 前回の講義：CPU の動作をふわっと説明
  - 「ここでもし命令が add だったら，加算する」
  - 「ここでもし命令が store だったら，メモリに書く」
- このような制御は，どのような回路で実現されるのか？



今日の内容：

- 命令パイプライン
  - ◇ ・・・の話をすると予告したが，あれは嘘だ
- 回路と遅延の話をします

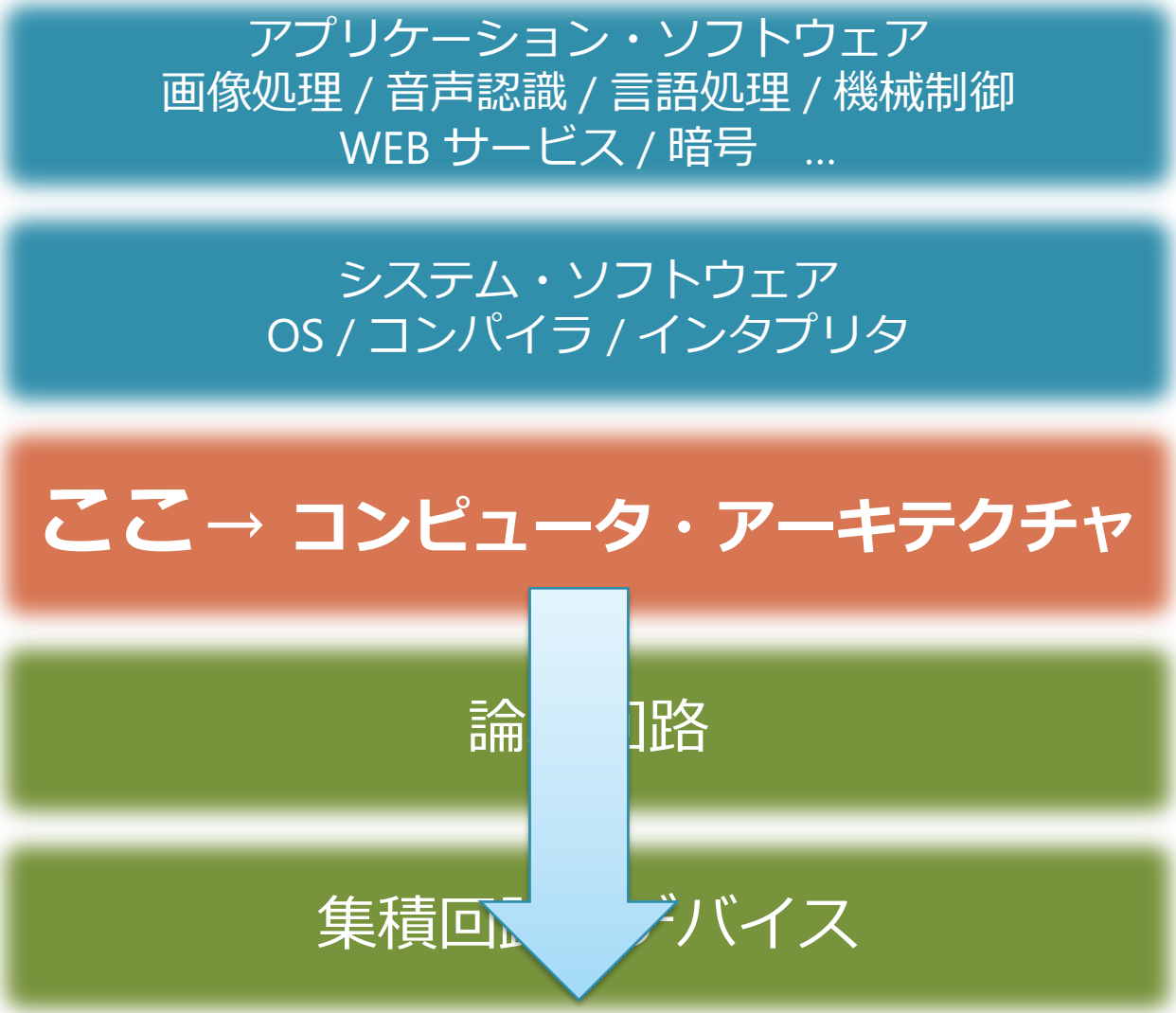
回路と遅延

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

情報の各分野の階層

- 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きました

情報の各分野の階層



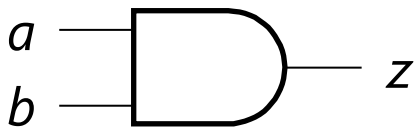
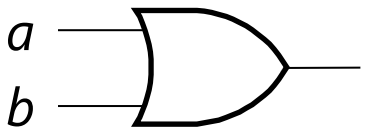
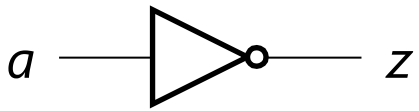
- 今回は，「コンピュータのハードを作るためには」という視点で，さらに下がっていきます

論理回路

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

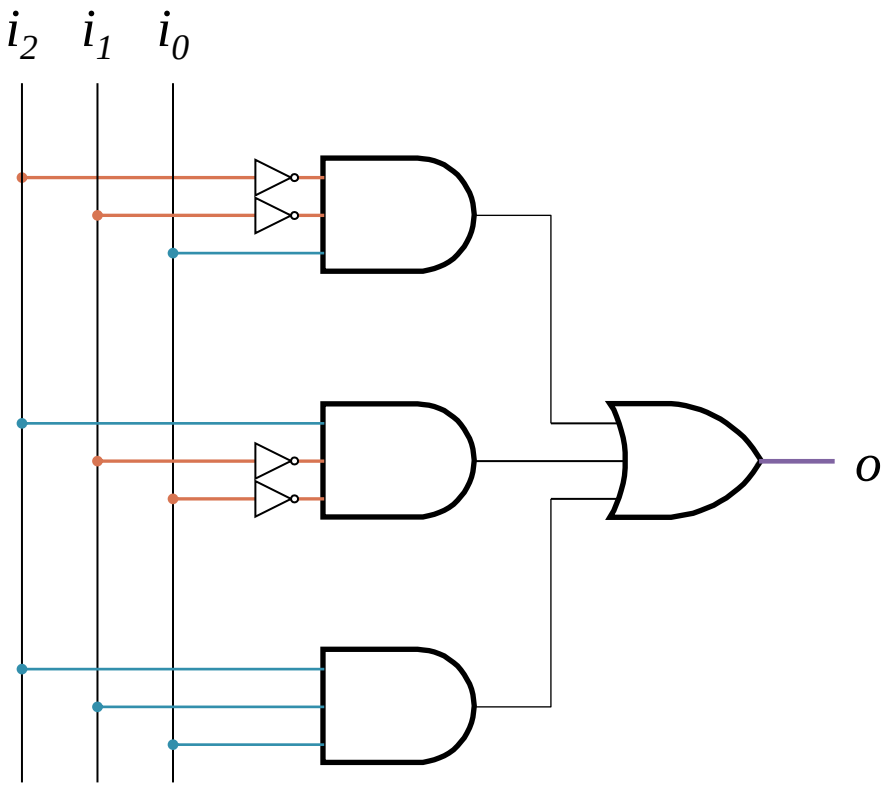
	AND （論理積）	OR （論理和）	NOT （論理否定）
MIL記号 MIL symbol			
論理式 logic expression	$z = a \cdot b$	$z = a + b$	$z = a'$ $z = \overline{a}$ $z = \neg a$
真理値表 truth table	a b z 0 0 0 0 1 0 1 0 0 1 1 1	a b z 0 0 0 0 1 1 1 0 1 1 1 1	a z 0 1 1 0

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
  - （証明は，数学的帰納法を使えばそんなに難しくない）

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]

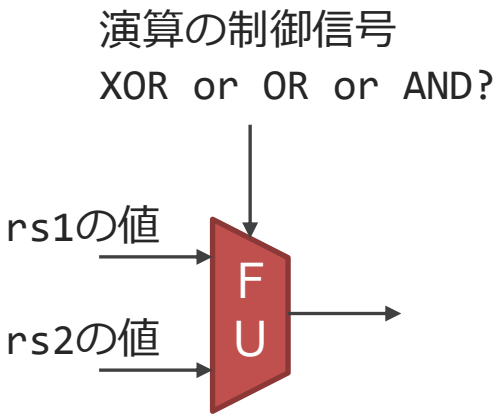
OR: x[rd] ← x[rs1] - x[rs2]

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 3ビットの違いで識別する

制御の例：RISC-V の AND/OR/XOR 命令

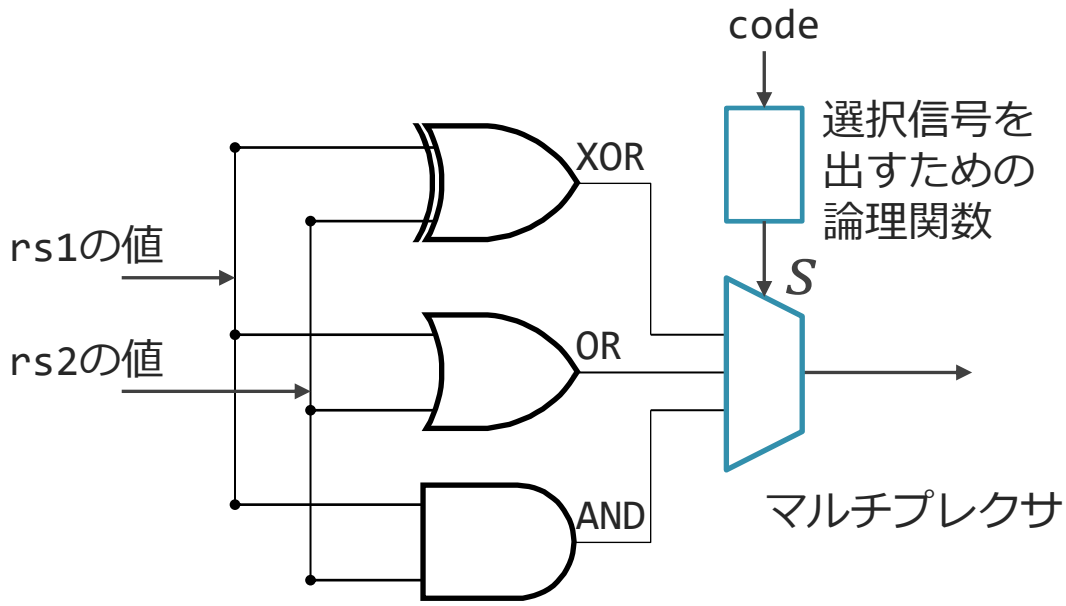
```
if code == 0b100:
    rs1 xor rs2
elif code == 0b110:
    rs1 or  rs2
else:
    rs1 and rs2
```



- 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

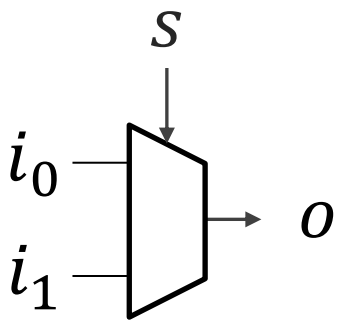
```
if code == 0b100:
    rs1 xor rs2
elif code == 0b110:
    rs1 or  rs2
else:
    rs1 and rs2
```

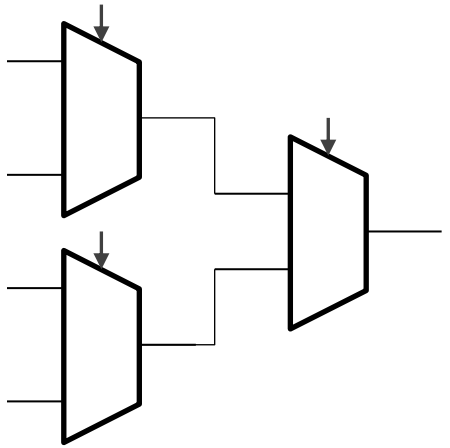


- 典型的には,
  - ◇ 各場合ごとの回路を用意して並列に配置
    - □ XOR, OR, AND ゲートを並べる
  - ◇ 制御に従ってマルチプレクサで出力を選択

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

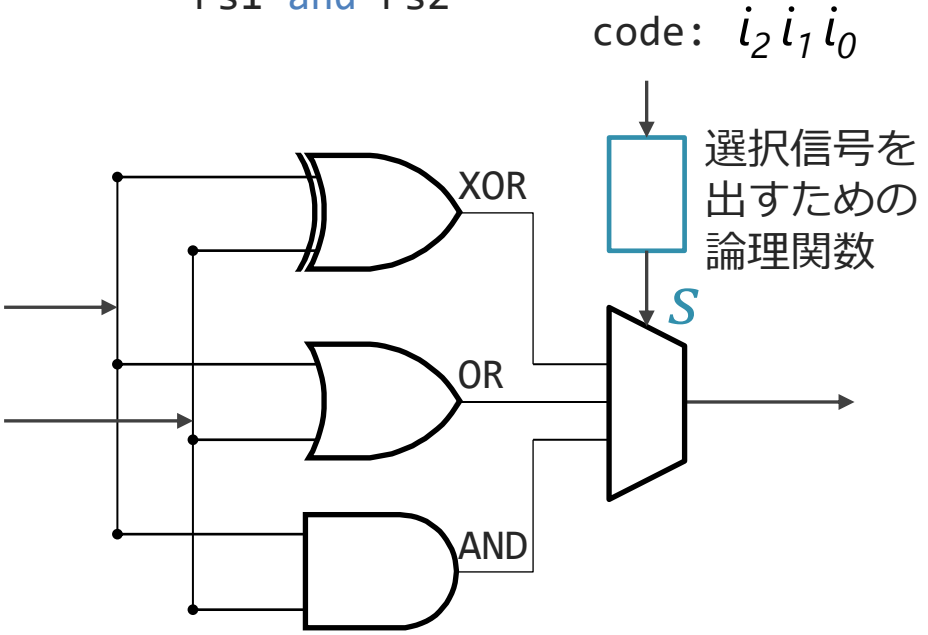




- 以下により，回路が生成できる
  - ◇ ２：１マルチプレクサは真理値表でかける = 回路が作れる
  - ◇ 多入力マルチプレクサは，カスケードすれば良い

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
```

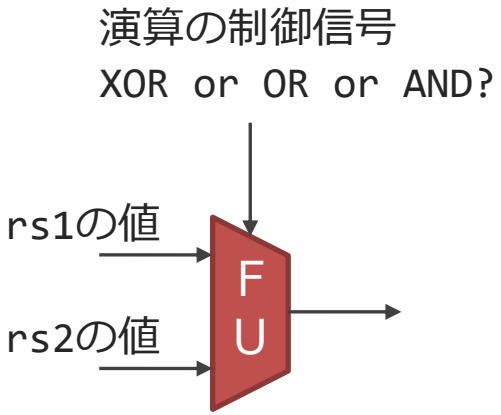


	i_2	i_1	i_0	s_1	s_0
	0	0	0	x	x
	0	0	1	x	x
	0	1	0	x	x
	0	1	1	x	x
XOR	1	0	0	0	0
	1	0	1	x	x
OR	1	1	0	0	1
AND	1	1	1	1	0

- 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

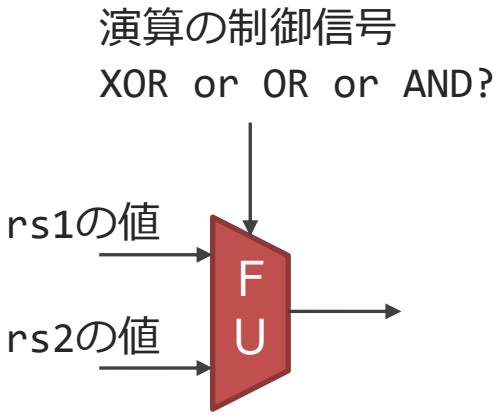
```
if code == 0b100:
    rs1 xor rs2
elif code == 0b110:
    rs1 or  rs2
else:
    rs1 and rs2
```



- 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，
  - ◇ 各場合ごとの回路を用意して並列に配置
  - ◇ 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
```

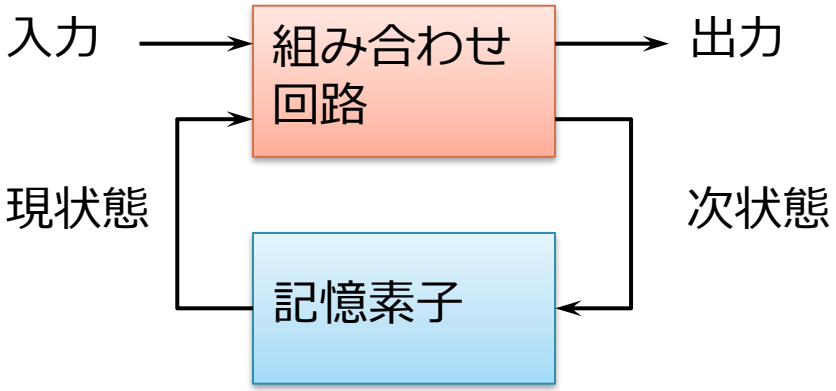


- code, rs1, rs2 を全て含む真理値表を作れば，そこから直接回路に落とし込むことも原理的にはできる
  - ◇ 表が大きくなりすぎて (2 の 3+32+32乗)，現実的には無理
  - ◇ もっと効率の良い表現方法で表現（後述）

組み合わせ回路と順序回路

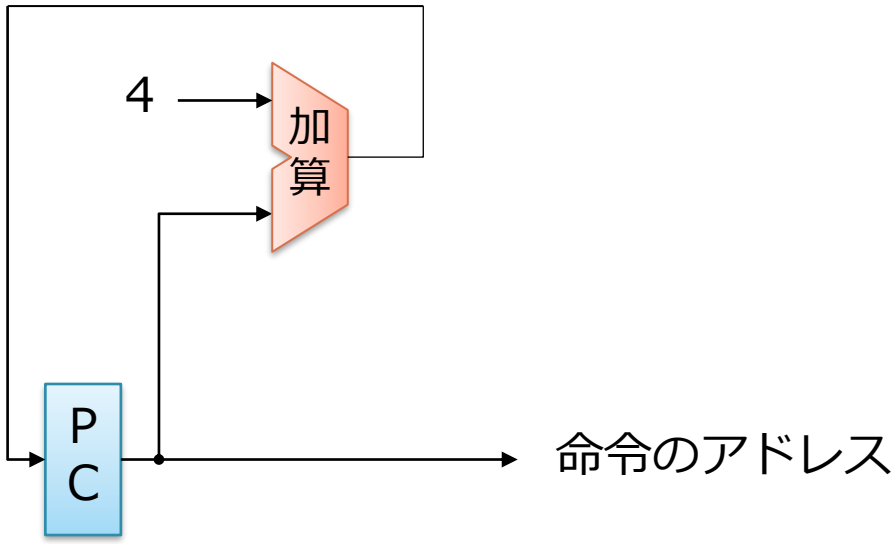
1. 組み合わせ回路
   - ◇ 出力が，現在の入力のみにより決定される論理回路
2. 順序回路
   - ◇ 出力が，過去の入力（の履歴）にも依存する論理回路

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

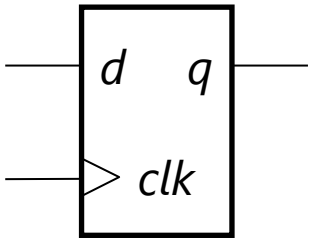
CPU の PC 部分



- 毎サイクル PC が加算される部分は，典型的な順序回路

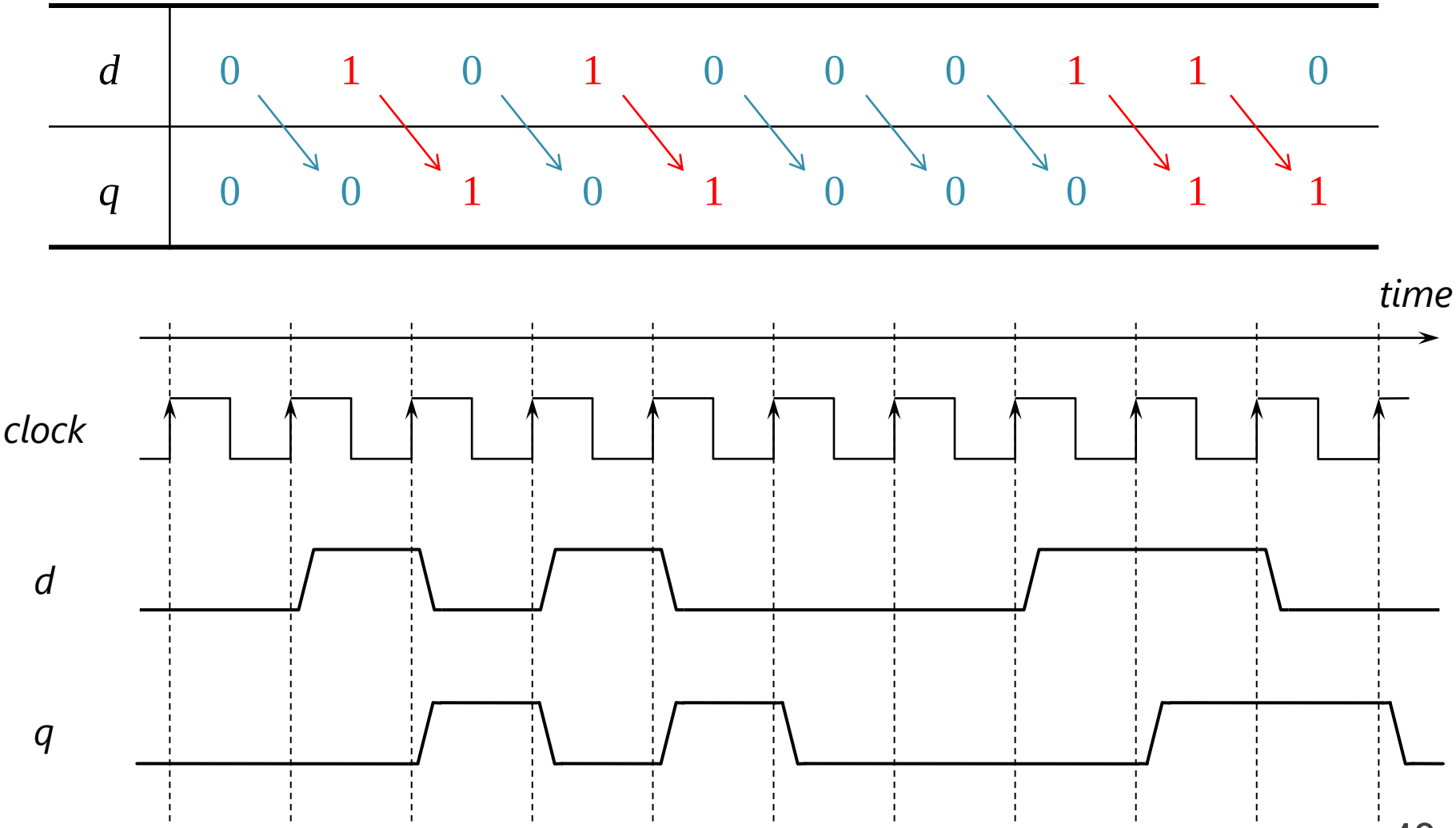
記憶素子の例：D-FF（Flip Flop）

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

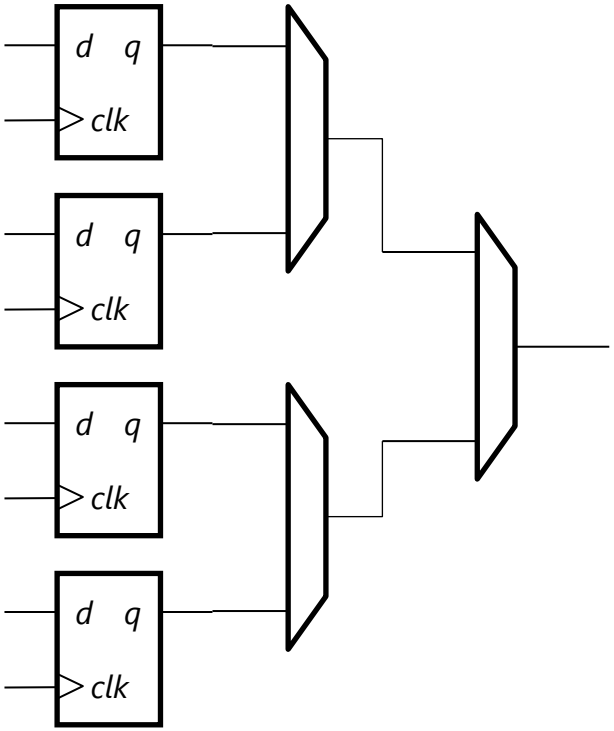


- 働き：
  - ◇ クロックの立ち上がりのたびに，*d* の値がサンプリングされる
  - ◇ その値が次のサイクルの間 *q* から出力される

D-FF の動作



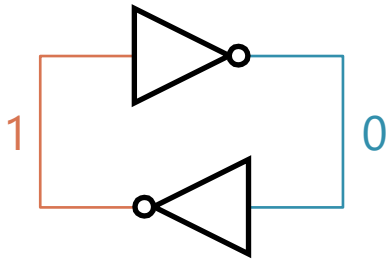
メモリやレジスタ・ファイル

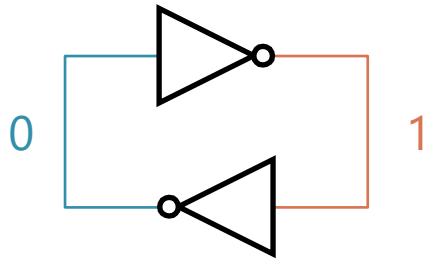


- D-FF を必要なだけ並べて，マルチプレクサで選択することで実現できる
  - ◇ 実際には，もっと最適化された回路（SRAM）が使用される事が多い
  - ◇ （後の講義で詳しく説明する予定

記憶素子の原理

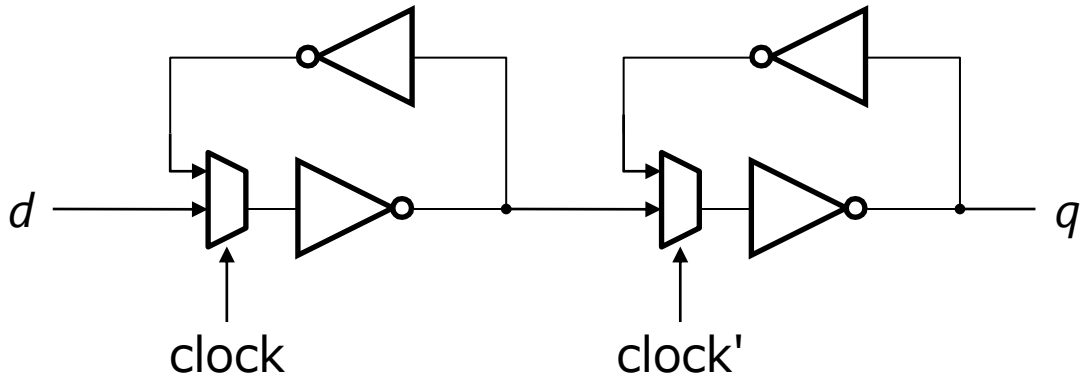
- 記憶
  - ◇ 2 つの NOT ゲートをループさせた回路により記憶
  - ◇ 2 通りの安定状態がある：1 bit を記憶





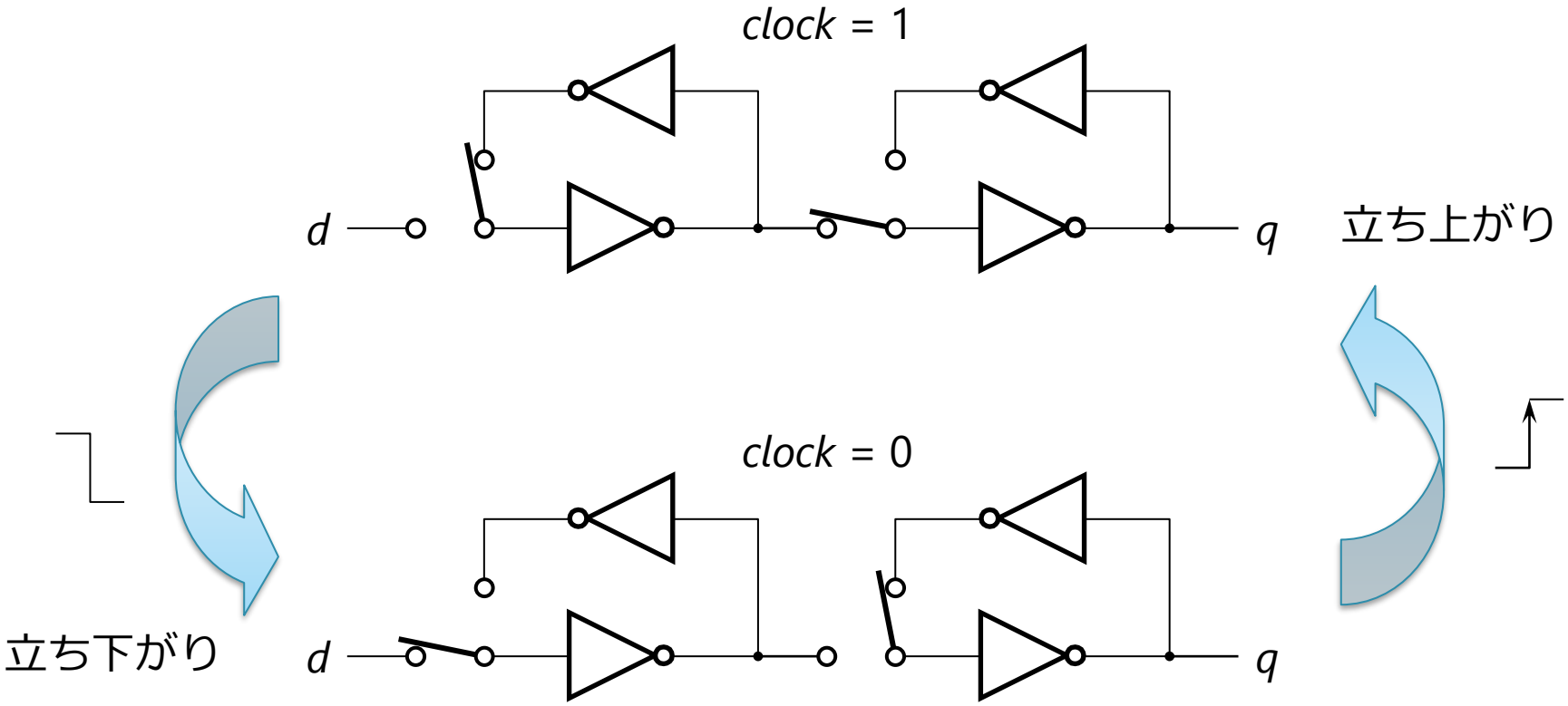
D-FF の実装

- 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



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

実際の回路生成

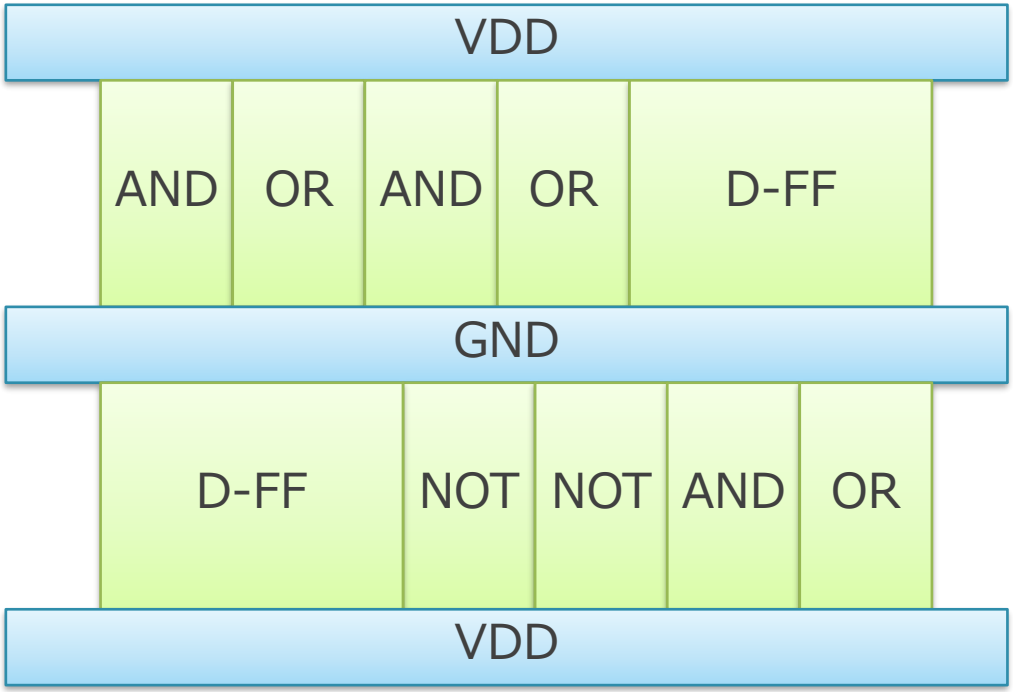
```
if code == 100:
    out = rs1 xor rs2
elif code == 110:
    out = rs1 or  rs2
else:
    out = rs1 and rs2
```

1. ハードウェア記述言語から，論理関数を生成：
   - ◇ ナイーブには，入力の全パターンに対して出力を記録した真理値表
     - □ 実際には表のサイズが爆発して，真理値表ではすぐに破綻
     - □ 表のサイズ = 2 の入力ビット数乗
   - ◇ さまざまな効率的な保持方法が提案されており，使用されている
     - □ Binary Decision Diagram (BDD)

実際の回路生成

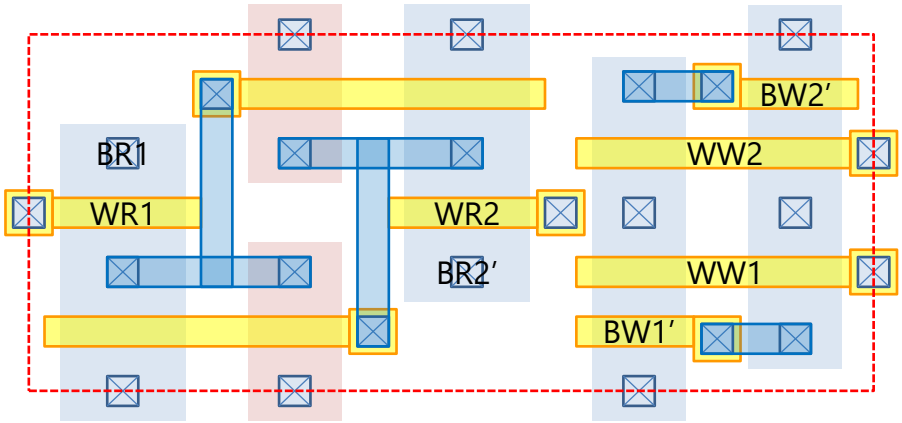
1. 論理合成：
    1. ハードウェア記述言語から，論理関数を生成
    2. 生成された論理関数を最適化
        - カルノー図：４入力ぐらいが限界
        - クワイン・マクラスキー法：入力数が増えると計算量が爆発
        - さまざまな最適化アルゴリズムが提案・使用されている
    3. 回路のプリミティブと接続関係を生成
        - AND, OR, NOT, D-FF を始めとして，よく現れる回路の部品が用意される
2. 配置・配線：
    1. 回路のプリミティブを配置・配線

スタンダード・セル



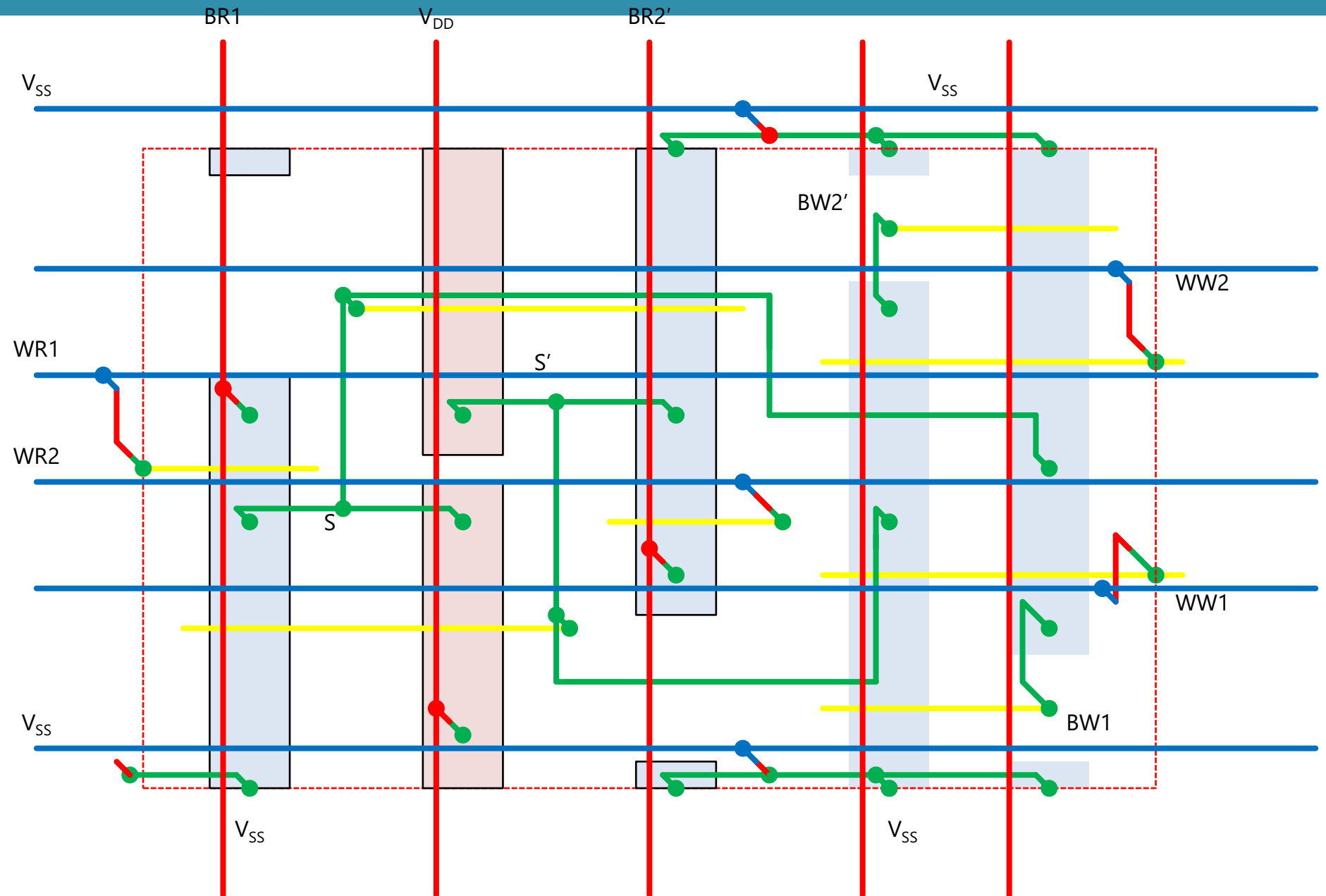
- スタンダード・セル：
  - ◇ AND や OR のようなプリミティブとなる回路「セル」を用意
    - □ 高さを幅を揃えておいて，簡単に並べられるように作ってある
  - ◇ これらを配置して，配線を接続することにより回路を実現
    - □ 配置配線という

フルカスタム・レイアウト

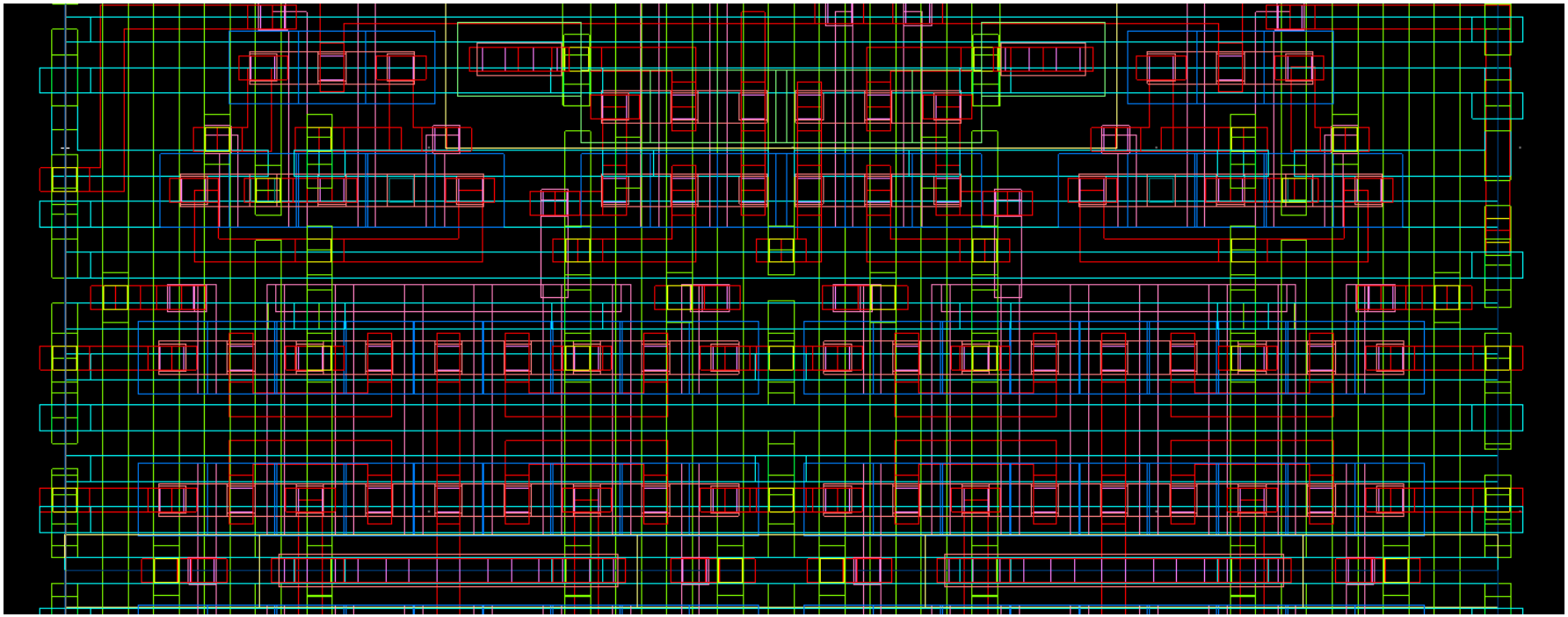


- 各スタンダード・セル（ゲート）は複数のトランジスタにより構成
- フルカスタム・レイアウト：
  - ◇ 半導体チップ上に１つ１つトランジスタをお絵かきして設計
    - □ 形状が特定のパターンを満たすと，トランジスタとして機能
  - ◇ 尋常じゃない手間がかかる
- 今はスタセルの中身と，本当に性能が出したい部分だけこれで作る
  - ◇ ごく一部だけアセンブリ言語でプログラムするようなもの

3次元的な構造を考えながら，設計

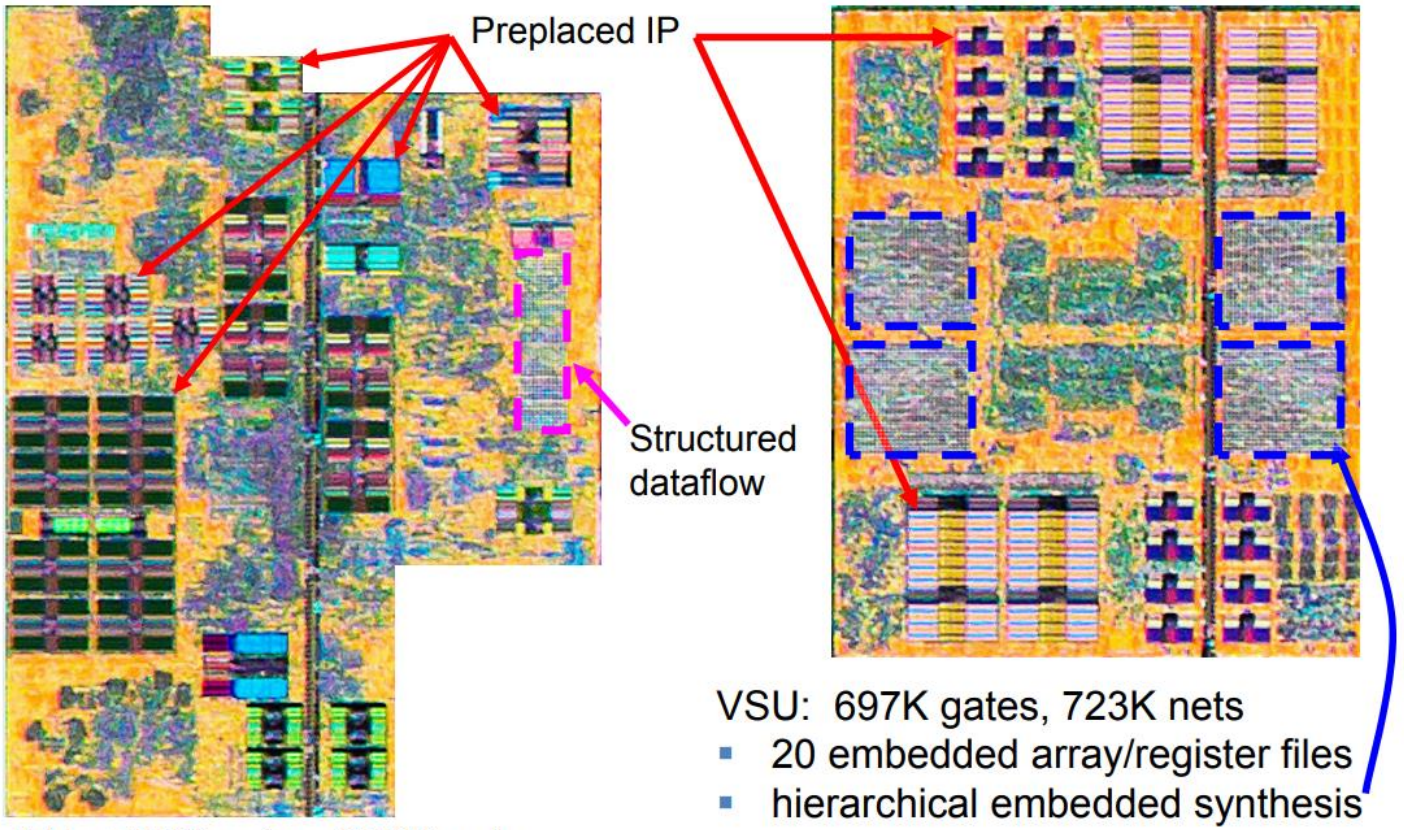


実際の設計データ



実際の例：IBM POWER8 のチップ写真

画像は 2014 IEEE International Solid-State Circuits Conference 5.1: POWER8TM: A 12-Core Server-Class Processor in 22nm SOI with 7.6Tb/s Off-Chip Bandwidth より



- ◇ もわっとした模様の部分がスタンダード・セルを自動で配置した部分
- ◇ 四角いところは基本的にはメモリ
  - □ 同じものが規則正しく並んでいる

回路の設計のまとめ

- 論理関数や記憶素子は,
  すべて完全集合の要素のゲートに分解して実装出来る

- CPU の設計と製造の流れ
  1. 命令セットの仕様を考える
  2. ブロック図のレベルで設計する
  3. ハードウェア記述言語で回路を記述する
  4. 論理合成する
  5. 配置配線する
  6. 設計データを工場で焼いてもらう

論理回路の実現方法

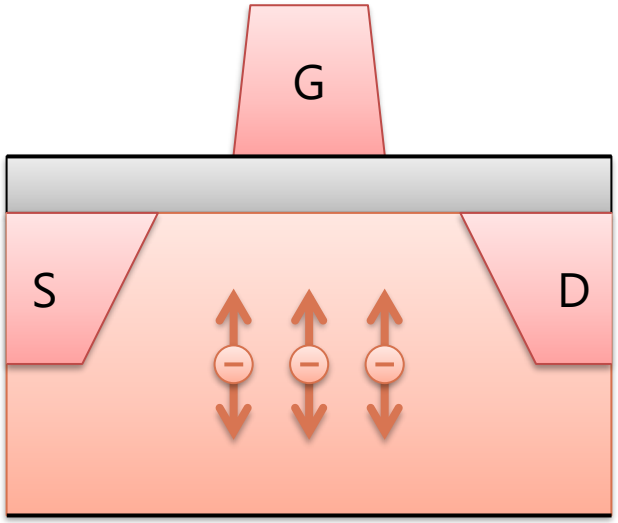
論理ゲートの構成

- CMOS と呼ばれる方式により，各種ゲートは実現されている
  - ◇ スタンダード・セルの中身は CMOS
  - ◇ フルカスタム・レイアウトは，CMOS を描いている

トランジスタ (MOS FET)

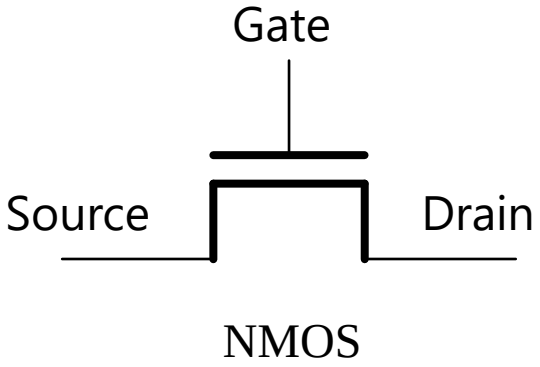
- MOS: metal-oxide-semiconductor
  - 上から，金属，酸化膜，半導体のサンドイッチ構造
  - 酸化膜を挟んで平行板コンデンサが構成されている

- 電界効果トランジスタ (FET: Field-Effect Transistor)
  - 電界で電子を動かしてスイッチングする
    1. G に電圧をかけてコンデンサに充電
    2. 電子の層（チャネル）酸化膜下にできて，S と D が繋がり ON に

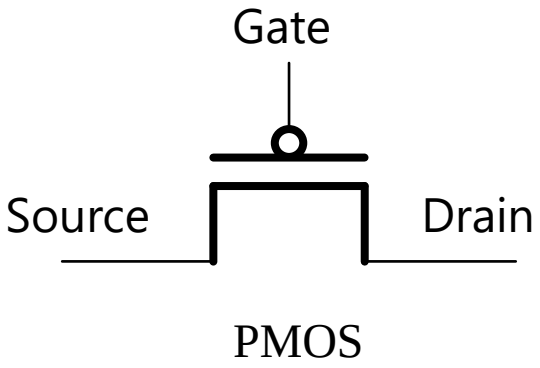


NMOS と PMOS の２つがある

- Negative
  - ◇ 電子がキャリア
  - ◇ 高電位を伝えられない
  - ◇ 接地側に配置

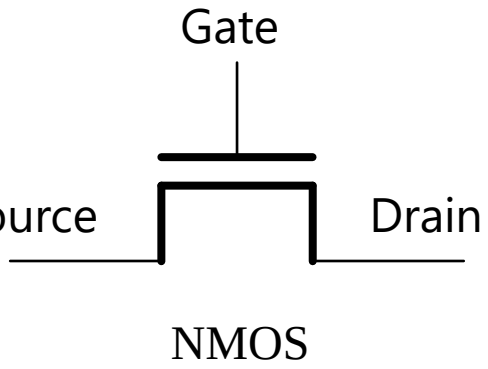


- Positive
  - ◇ 正孔 (holl) がキャリア
  - ◇ 低電位を伝えられない
  - ◇ 電源側に配置

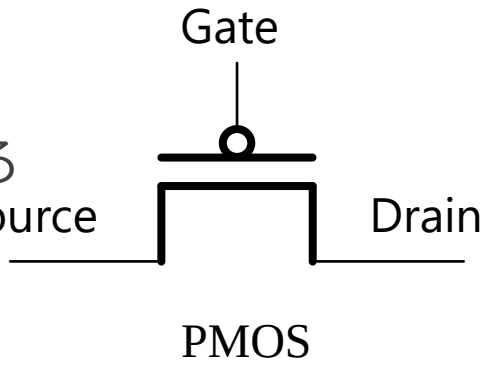


NMOS と PMOS の２つがある

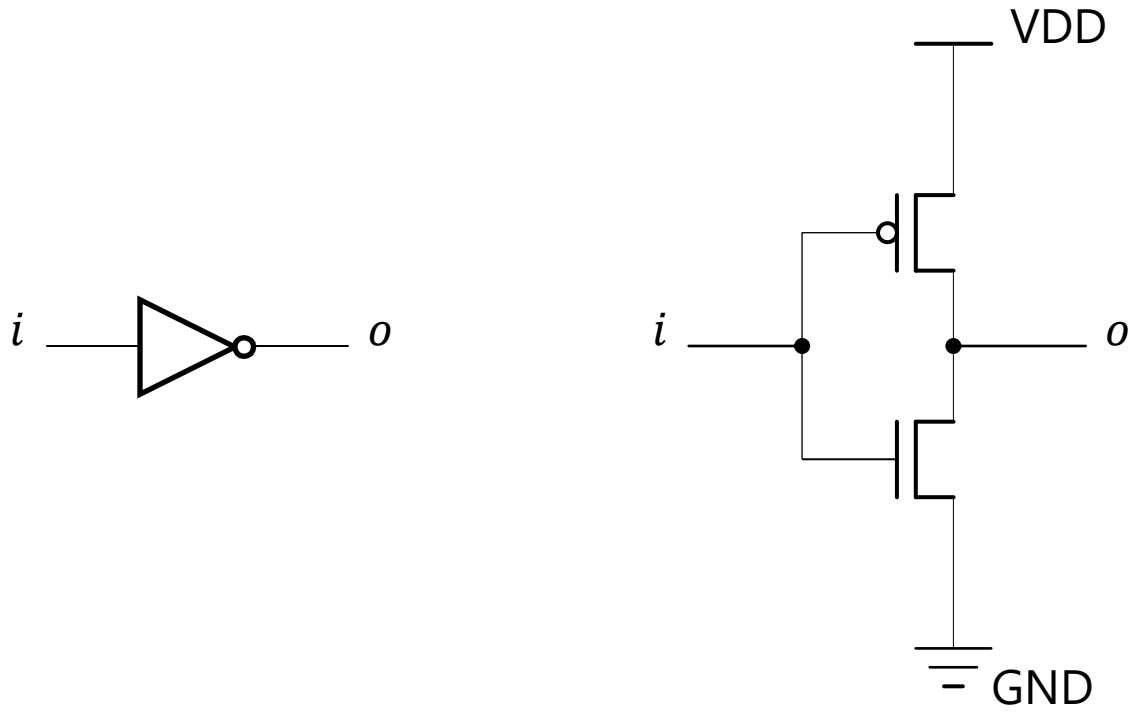
- Negative
  - ◇ Gate を 1 にすると低電位のみ ON になる



- Positive
  - ◇ Gate を 0 にすると高電位のみ ON になる

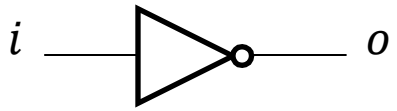


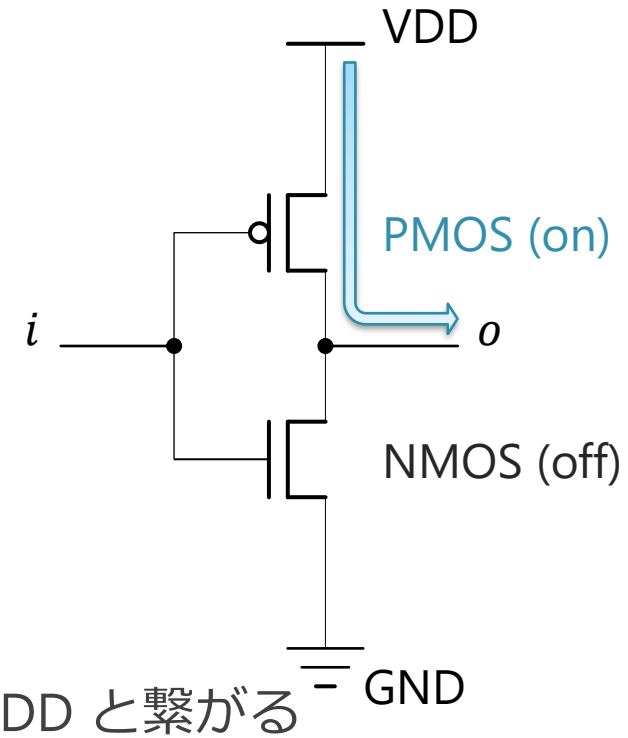
NOT ゲートの例



- PMOS（上側）は高電位しかうまく伝えられないので，VDD に
- NMOS（下側）は低電位しかうまく伝えられないので，GND に

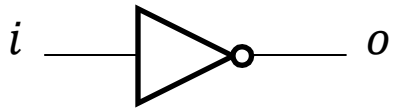
NOT ゲートの例

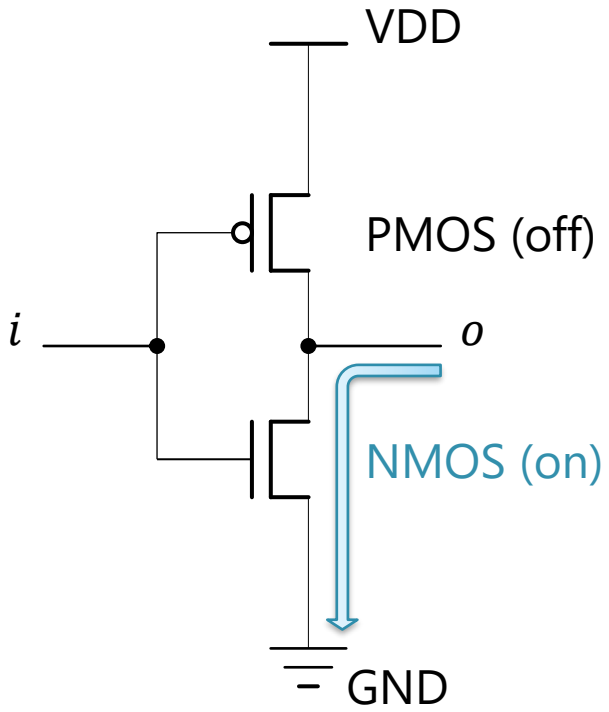




- i が 0 （低電位）の場合
  - ◇ PMOS（上側）は on = VDD と繋がる
  - ◇ NMOS（下側）は off
- 出力が 1（高電位に）

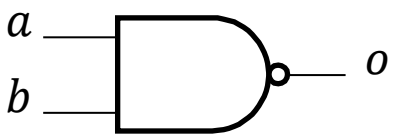
NOT ゲートの例





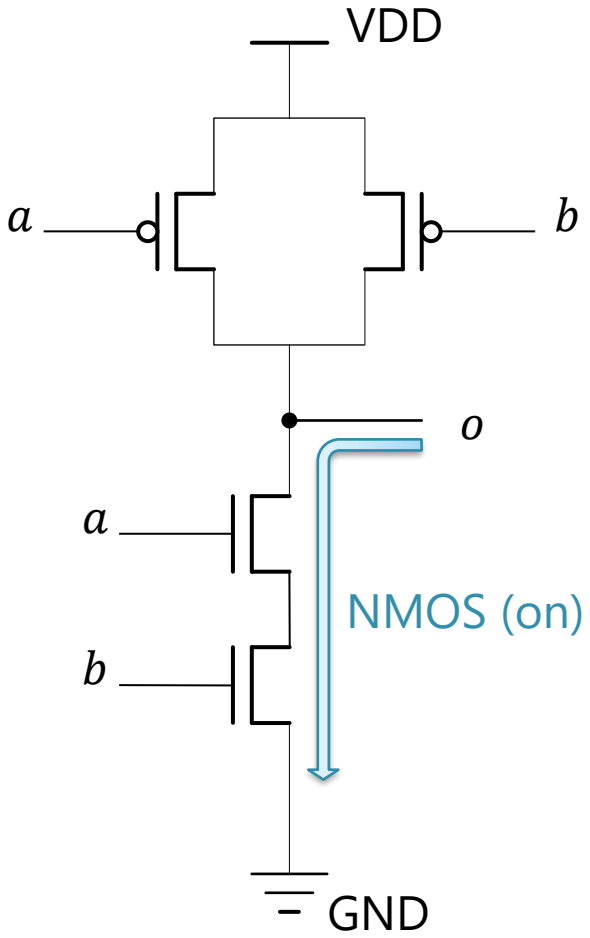
- i が 1 （高電位）の場合
  - ◇ PMOS（上側）は，off
  - ◇ NMOS（下側）は，on = GND と繋がる
- 出力が 0（低電位に）

NAND の例



$o = (a \cdot b)'$

a	b	o
0	0	1
0	1	1
1	0	1
1	1	0



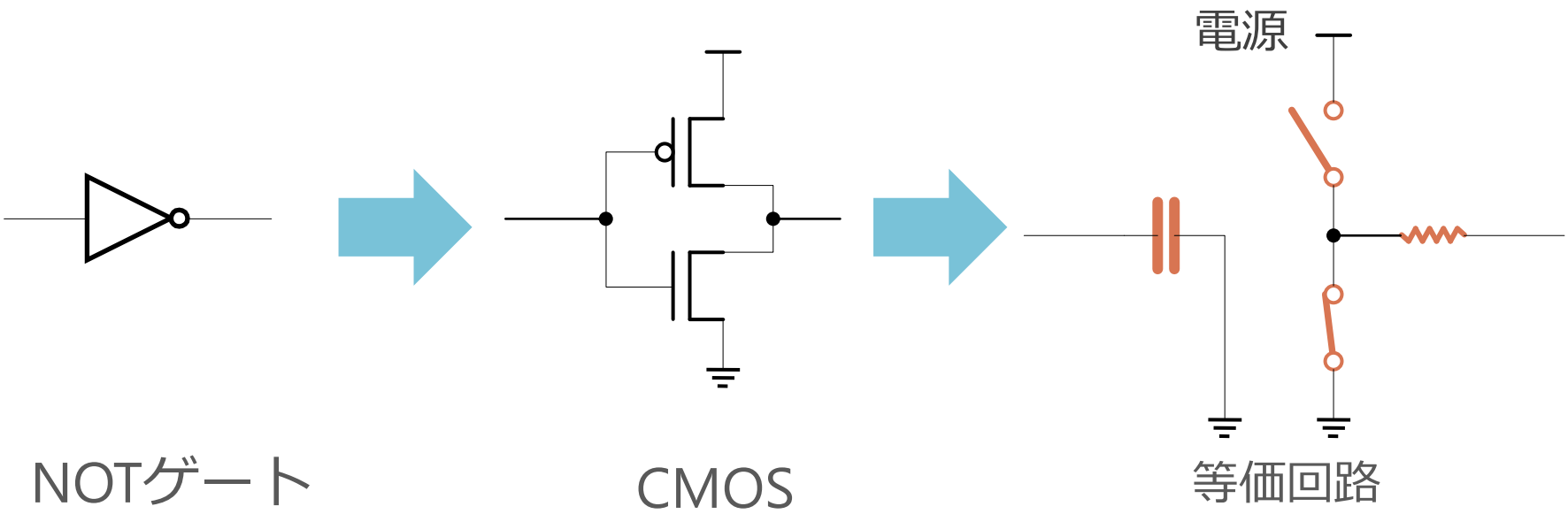
- a と b 共に 1 （高電位）の場合
  - ◇ PMOS（上側）は，off
  - ◇ NMOS（下側）は，双方 on = GND と繋がる
- 出力が 0（低電位に）

CMOS による論理回路の実現のまとめ

- 全ての論理ゲートは,
  NMOS/PMOS の組み合わせによって構成されている

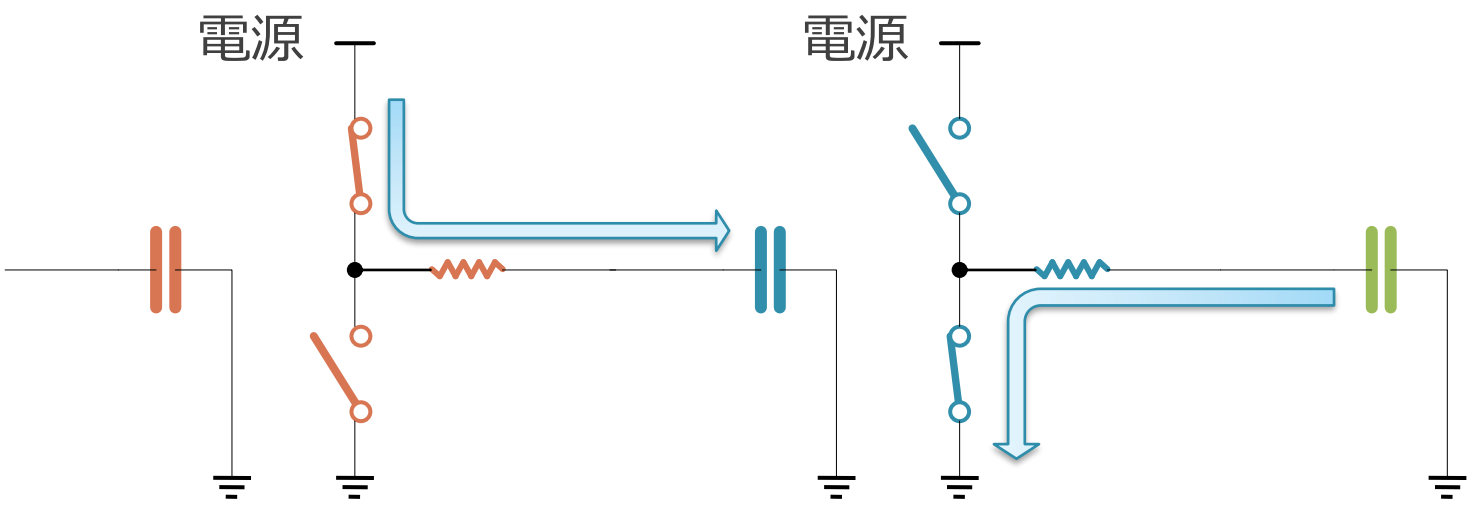
CMOS 回路の遅延

CMOS ゲートの等価回路



- 抵抗 & コンデンサと，連動したスイッチによって表せる
  - ◇ コンデンサに充電：下のスイッチがON
  - ◇ コンデンサを放電：上のスイッチがON

CMOS ゲートの遅延の実体

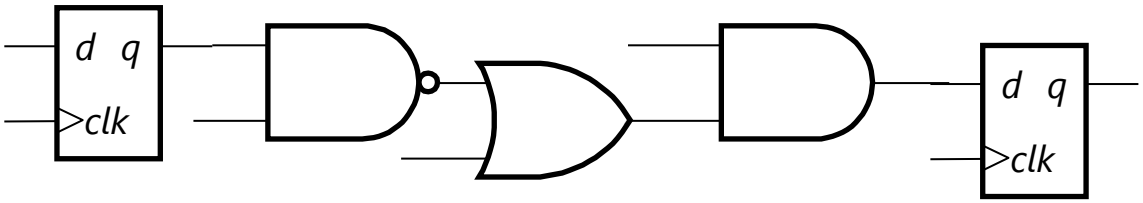


- 遅延：コンデンサの充放電にかかる時間
  1. あるゲートのスイッチが切り替わる
  2. 次の段のゲートへの充放電が開始
  3. 次の段のスイッチが切り替わる
  4. ・・・

実際には

- 寄生容量：
  - ◇ トランジスタのゲート部分以外にも，あらゆる場所にコンデンサができてしまう
  - ◇ なにか導体が不導体をはさんで並べばコンデンサになる
- 寄生容量への充放電にも時間がとられる
  - ◇ 通常はトランジスタのゲートがメイン
  - ◇ 長い配線では，配線間にできてしまうコンデンサの影響も大きい

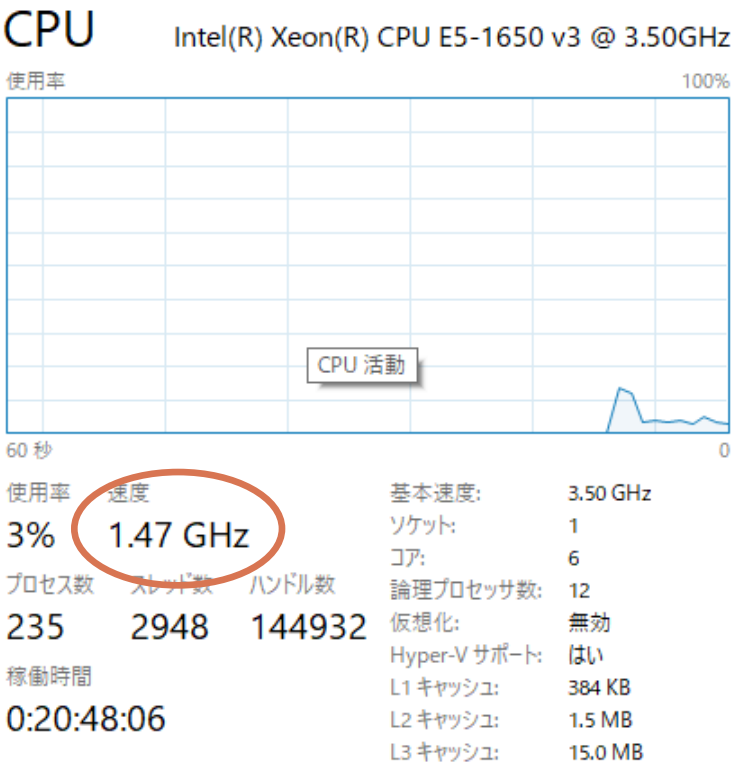
遅延の違い

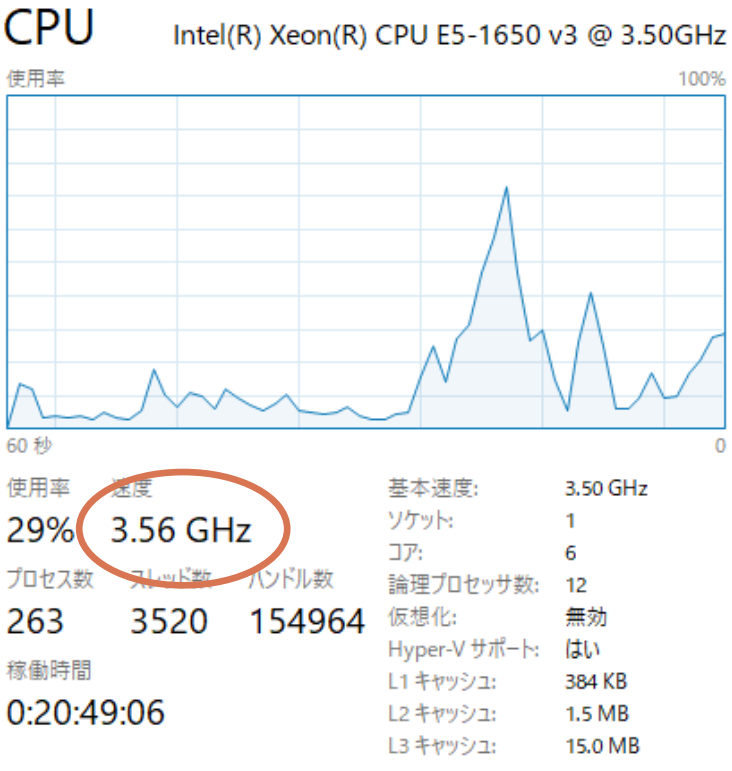


- 左側の D-FF から何個もゲートを経て右の D-FF に接続（遅い）
  - ◇ q が切り替わると，
  - ◇ 各ゲートのスイッチ（充放電）が順々に行われて，
  - ◇ 右側の d に伝わる

- 左側の D-FF から１つのゲートだけを経て右の D-FF に接続（速い）
  - ◇ q が切り替わると，１回だけスイッチ（充放電）が行われて，
  - ◇ 右側の d に伝わる

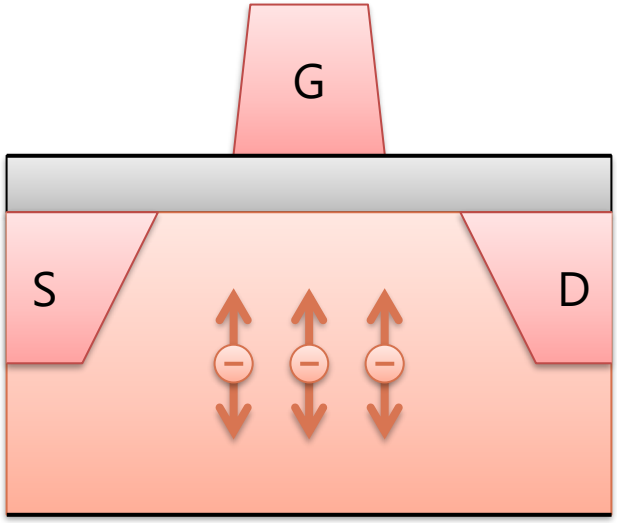
余談：DVFS (Dynamic Voltage Frequency Scaling)





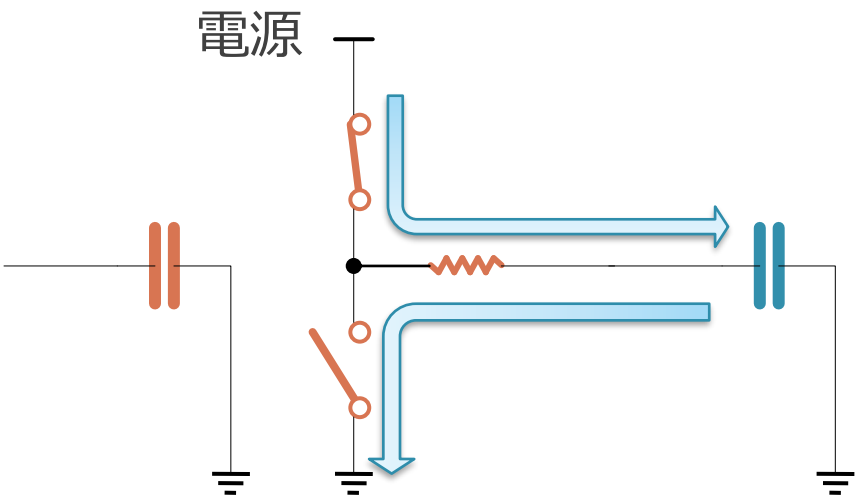
- CPU は通常，負荷に応じて動作周波数を変えている
  - ◇ 消費電力削減のため
  - ◇ この時，電圧も同時に操作している

電圧と遅延



- Gate にかける電圧を上げると，より高速に動作する
  - ◇ よりたくさん電子が上に集まる
  - ◇ チャネルが厚くなり，電流が流れやすくなる
  - ◇ 次段のコンデンサの充放電が速くなる

消費エネルギー



- 消費エネルギーは，主にコンデンサへの充放電で消費される
  - ◇ 消費エネルギーは電圧の二乗に比例：$E = CV^2$
  - ◇ 電荷 $Q = CV$ が，電圧 V の分だけ電源から GND へ移動するから
    - □ 忘れた人は高校の物理の教科書を読もう
- 他にリーク電流と呼ばれるものによる消費もある
  - ◇ スイッチが一部ガバガバなので，常時多少漏れてる

DVFS: Dynamic Voltage Frequency Scaling

- 電圧と周波数の組を用意しておき，これを切り替える
  - ◇ 高速：高周波数（低遅延）で動かすために，電圧を上げる
  - ◇ 低速：低周波数（高遅延）でよいので，電圧を下げる
- 電圧を下げると，すごく消費電力が減る
  - ◇ 消費エネルギーは電圧の２乗に比例：$E = CV^2$
- 低周波数にすると充放電の回数自体（=周波数）も減る
  - ◇ 結果として，3乗のオーダーで電力が削減できる

まとめ

- 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，それを実現する物理的な回路の繋がり
  - それら論理回路の遅延
- 論理回路の復習から始めて，遅延が何できまるのかまでを説明
  - 論理回路と，その設計
  - CMOS による実現
  - 遅延
- 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

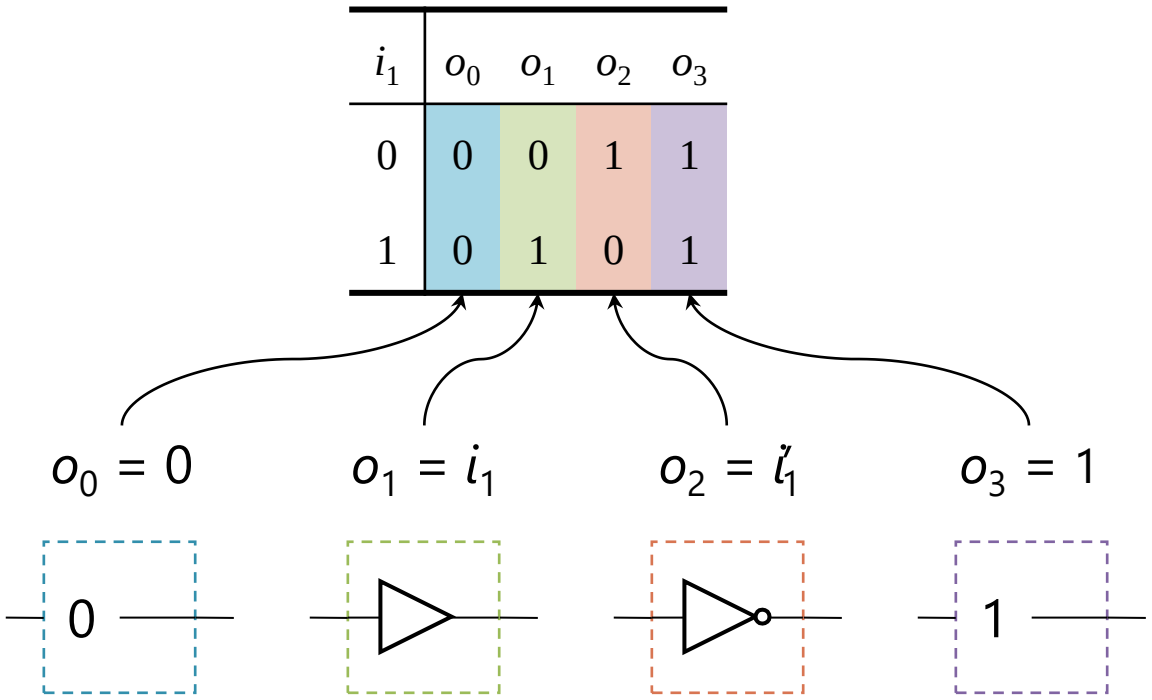
出欠と感想

- 本日の講義でよくわかったところ，わからなかったところ，質問，感想などを書いてください（なんか一言書いてね）
  - LMS の出席を設定するので，そこにお願いします
  - パスワード：logic
- 意見や内容へのリクエストもあったら書いてください

ボツ資料

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



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	

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