コンピュータ アーキテクチャ Ⅰ　第４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

- 今回の授業は全体的に難しかったと思いました。
- RISC-Vのところがあまりわからなかったです。
- 今回の内容は少し難しかったです。

質問や感想など

- I-type,S-typeのfunct3で格納バイト数が異なる他のロード、ストアにしているとありましたが、なぜ変えるのでしょうか。

質問や感想など

- RISC-Ｖでは、演算命令以外の命令も全てR,I,S,Uのうちどのタイプでエンコーディングされるか決まっており、opcode部分からはタイプを判別した上で何の命令かをある程度まで絞ることができ、そのopcode部分を持つ命令が1つしかない場合を除いて、funct部分で何の命令かが確定するという理解で正しいでしょうか。

質問や感想など

- 命令セットはCPUを動かすための命令語を体系化したものだったかと思います。
  p.48に「オープン」という言葉は「自由に互換品を作れること」とありますが、命令セットの互換品というものがいまいちピンときません。命令セットアーキテクチャを自由に使用可能ということでしょうか？

  - 正確には，同じ命令セットの命令で書かれたプログラムを動かすことができる互換 CPU

質問や感想など

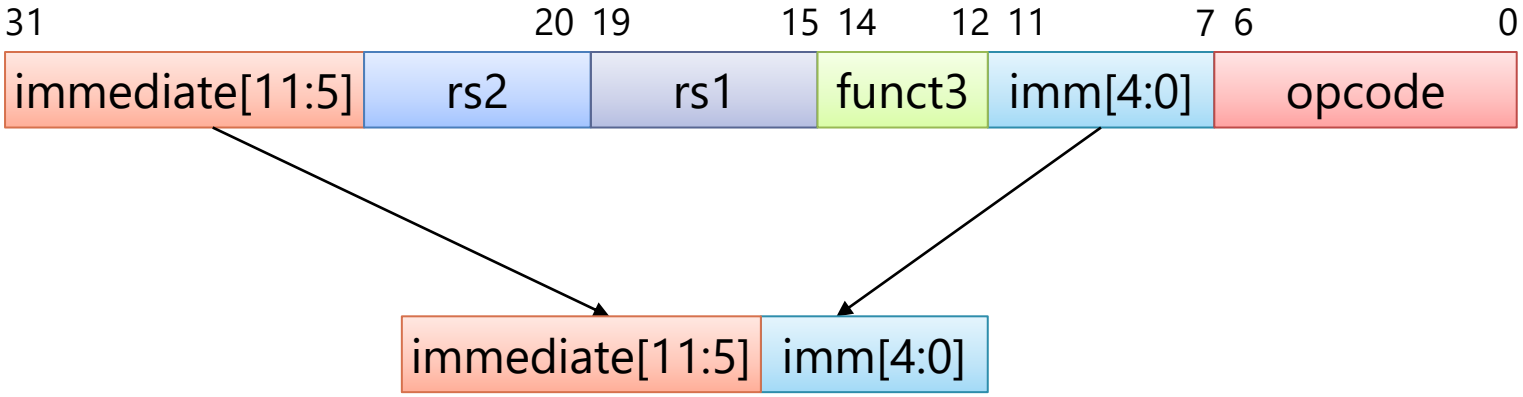
- RISC-V命令セットについてです。レジスタの幅には32、64、128bitがあるとのことでしたが、どのように使い分けるのでしょうか。32bitの方が好ましい例、128bit出ないといけない例など、使い分けを知りたいです。
  - ポインタ（アドレス）もレジスタに格納される
  - ポインタの最大の大きさはレジスタに格納できる幅で決まる
  - ポインタで指せる範囲 => 最大メモリ搭載量を決めている

質問や感想など

- パソコンのアプリで32bit版と64bit版があるのは、使用するレジスタの幅の違いなんですか？

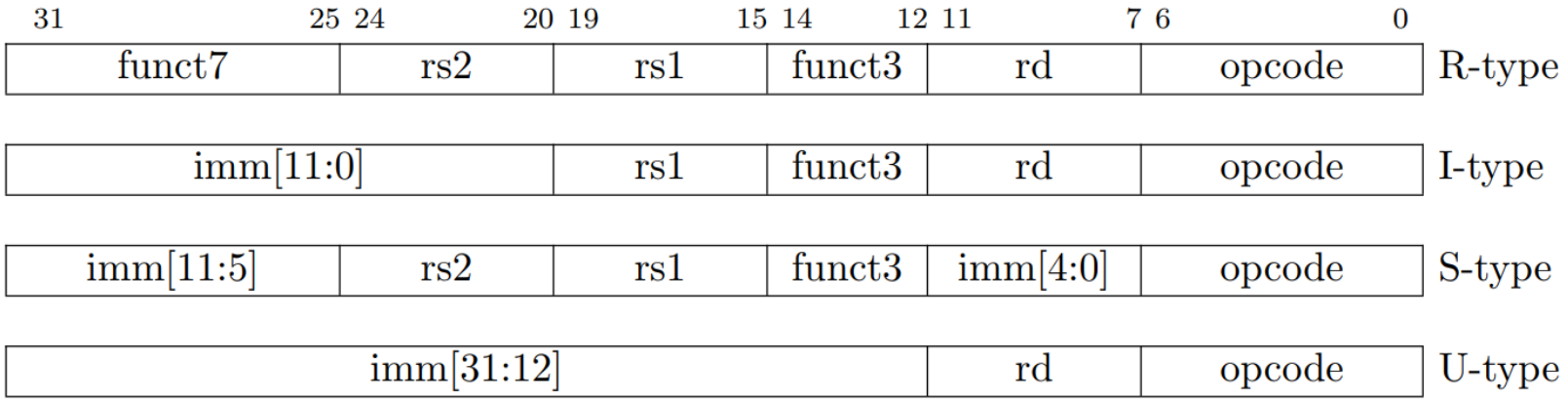
質問や感想など

- 資料の54ページのimmのところで、7-11bitと25-31bitを結合して11:0になるのがなぜ？となりました。



オペランドの格納方法

- 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
  - imm[11:5] は 5bit 目から 11bit がそこに格納されるということ
  - S-type では 元の 32bit のうち 7-11bit 目と 25-31bit 目をを取り出して結合し imm[11:0] (12bit) の値を作る



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

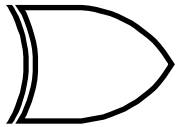
質問や感想など

- コンピュータ内では16進数であらわすということを決定してから1byteが8ビットということが決定したのか、1byteが8ビットであると決まっていたから16進数で表すようにしたのか気になりました。

質問や感想など

- XORとはどういう命令なのか分からなかったです。

a	b	z
0	0	0
0	1	1
1	0	1
1	1	0

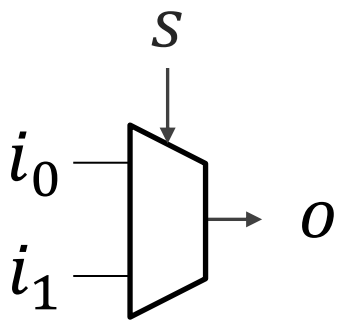


質問や感想など

- また、マルチプレクサも具体的にどういうふうに複数の入力からひとつを選んでいるのかが分からなかったです。

マルチプレクサ：複数入力から1つを選ぶ回路

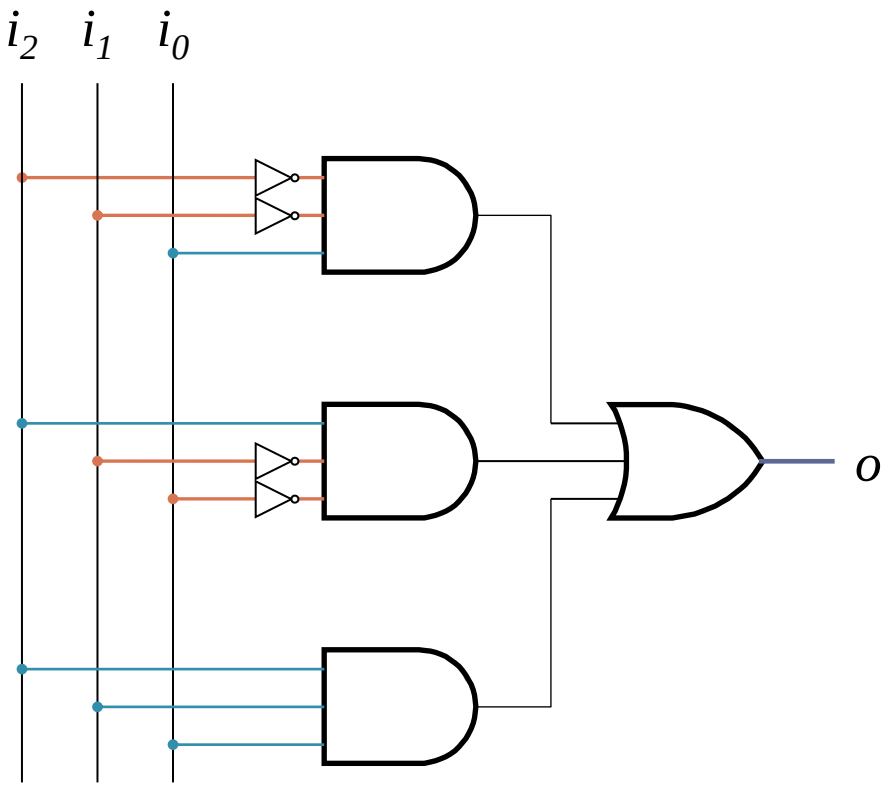
s	i_0	i_1	o
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



- ２：１マルチプレクサは真理値表でかける
  - s, i0, i1 の全ての取り得る値の組み合わせが入力に書かれている
  - s に着目すると,
    - s=0 のときは i0 が o に
    - s=1 のときは i1 が o に それぞれそのまま出ている=選択

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

質問や感想など

- XORはORとANDを組み合わせれば結果的に演算が可能ですが、XORの命令のコードがあるのには何か特別な理由があるのでしょうか。単純にまとめたほうが効率がいいからでしょうか

質問や感想など

- 16進数では、文字が数字として使われているのですが、文字列はコンピュータは直接理解することができないと認識しているのですが、この場合、別の進数に変換されてから計算を行っているのでしょうか。

質問や感想など

- 真空ジェシカというお笑い芸人のネタで二進数の指折りの話がでてきてて、その時はそんなのがあるんだ～程度にしか思いませんでしたが、二進数の指折りにちゃんと意味があるんだな～と思いました。

質問や感想など

- 論理回路は数理基礎論で学習済みでしたが、回路図を書けるようになって終わりでした。
- 今回の講義は今までの授業であった、数理基礎論やコンピュータシステム序論などと重なる部分が特に多く、全部意味があったんだなと思うと同時にこれだけ出でくるならもっとしっかり理解しないとと思いました。

質問や感想など

- C言語が機械語と近いとの事だったのですが、初心者にもわかりやすいようにしているプログラミングを作るのは機械語に近い言語より難しいのですか？

質問や感想など

- CPUの仕組みを実際に体験したいので、自作で作ってみようかと思い始めました。
  - 研究室の新入生向けの CPU 設計演習：
  - https://github.com/shioya-lab/cpu-exercise

前回の振り返り

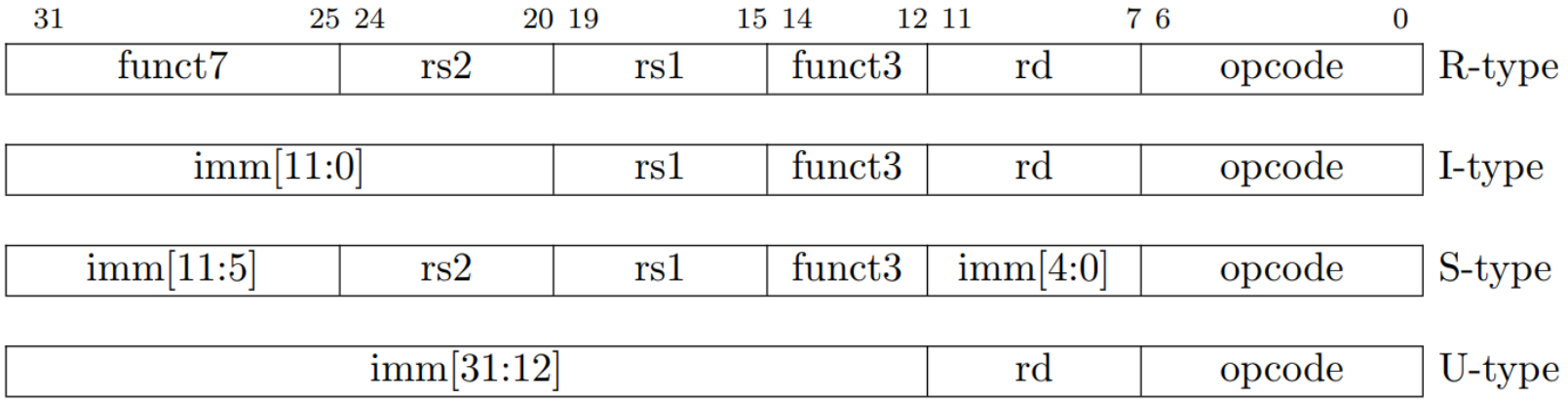
２進数や１６進数による数値表現

- 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」のような形で表現
  - ２進数だと回路を作るのが簡単
    - 中間の電圧とかを扱うのは大変
- ２進の利点と欠点
  - 利点：各桁に意味を持たす事が多いので色々と考えやすい
  - 欠点：桁数が大きく人間に把握が難しい
- １６進の利点
  - 利点：２進数とは相互変換が容易
  - 欠点：慣れていないとやや把握が難しい

実際の命令セットの例

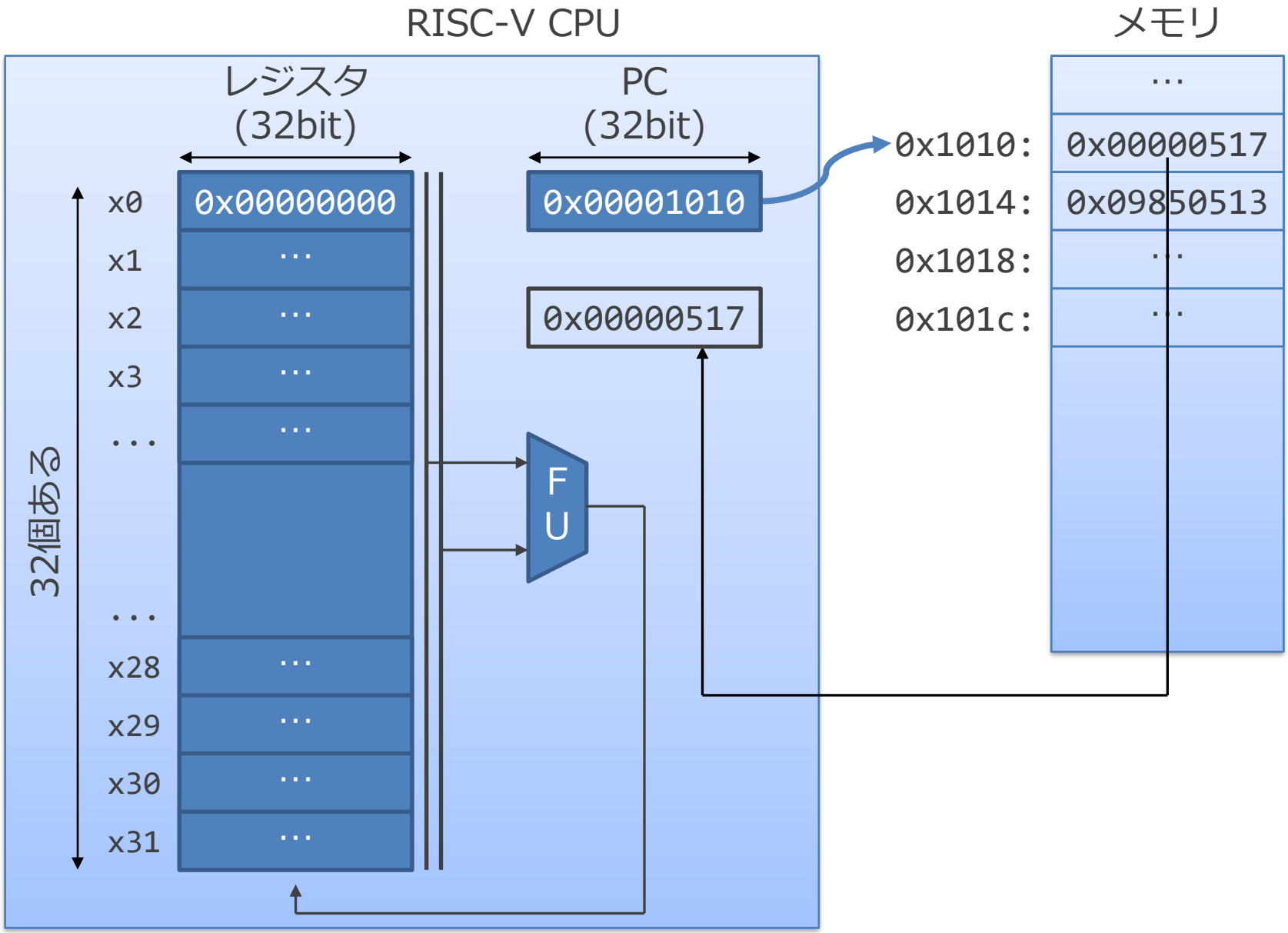
- RISC-V を例として説明
  - 加減算，論理演算，ロード・ストア，即値，分岐とジャンプなど
  - 各命令は基本的に 32bit 幅

- １回目の講義で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



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

RISC-V (32bit) のイメージ



組み合わせ回路/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

- 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

論理回路の実現方法

論理回路の作り方

- 前回の内容：
  - AND，OR，NOT があれば，どんな論理回路も作れる
    - 組み合わせ回路も順序回路も
  - CPU を構成する演算器や PC，レジスタ，メモリも作れる
    - 具体的な作り方の例をそれぞれ説明
- 今回の内容：
  - AND，OR，NOT のようなゲート回路はどのように作るのか？
  - それらのゲート回路の「遅延」や「消費電力」とは何なのか？
    - （なぜ物理的実体のない情報を扱っているのに時間や電力がかかるのか？

論理回路の作り方

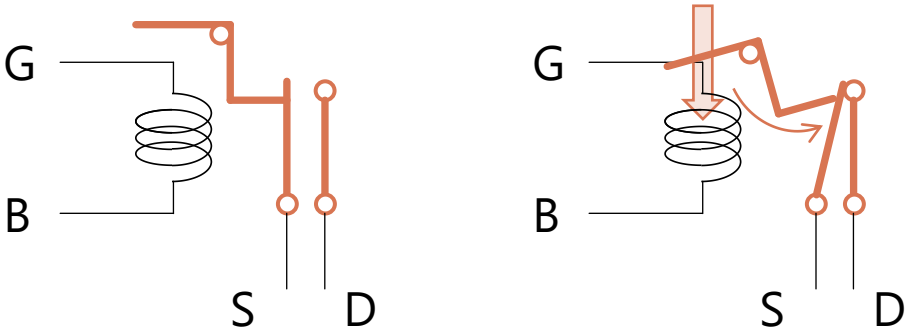
- 以下のそれぞれの仕組みを使った回路を説明
  1. リレー
  2. CMOS
- これらの論理的な動作は実はほぼ同じに作れる
  - リレーの方が直感的にわかりやすいのでこちらから説明

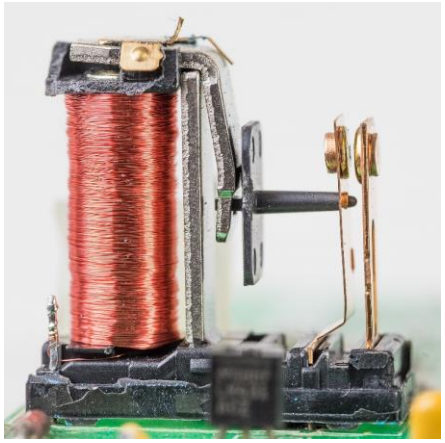
- 注意：
  - この資料で説明する，N型/P型 を使ったやり方は後の CMOS の説明をわかりすくするためのものです
  - 実際に用いられていたリレー計算機ではもう少し効率の良い違う方法が使われていたと思う

リレー（Relay）

- 電気的に動作するスイッチ
  - もともとは長距離の電信の中継のために作られた･･･らしい

- 以下の図のものの場合，
  1. 普段は電極部分がバネになっていて OFF になっている
  2. G～B 間に電圧をかけると，
  3. コイルが磁石となって，上の金属を引き寄せ，
  4. S に繋がっている端子が右に押し出されて，
  5. S と D が接続されて ON に

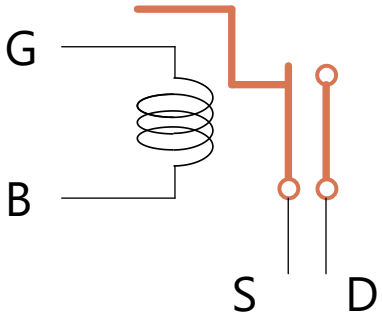


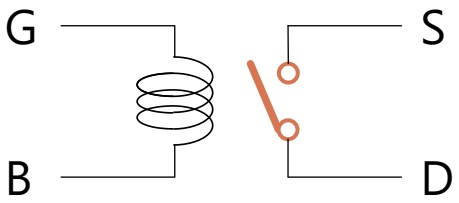


画像は https://en.wikipedia.org/wiki/Relay より

リレーの絵を回路記号に単純化

- わかりにくいので，以降では右のように単純化して表記
  - G～B 間に電圧をかけると，S～D が ON に

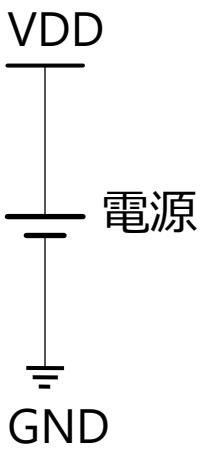




VDD と GND

- 以下の２端子を定義
  - VDD：高電位
  - GND（Ground=地面）：低電位

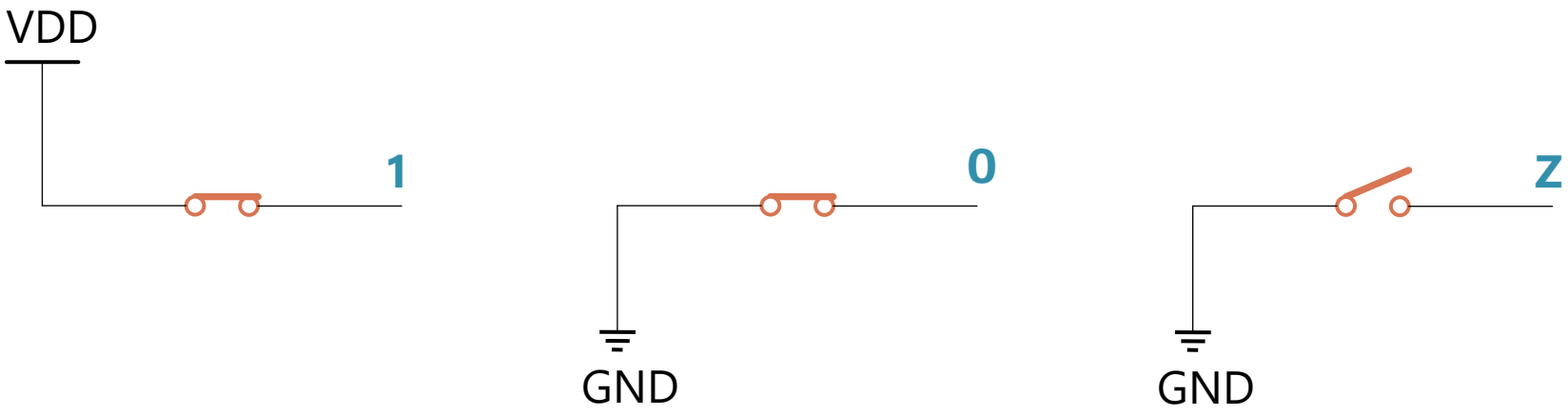
- 記号の描き方
  - 基本的に絵の上側に VDD，下側に GND を描く
  - VDD と GND があったら，図内に描かれていない世界のどこかで以下のように電源に繋がっていると考える



電位と数値表現

- 高電位を 1，低電位を 0，解放を Z と定義する

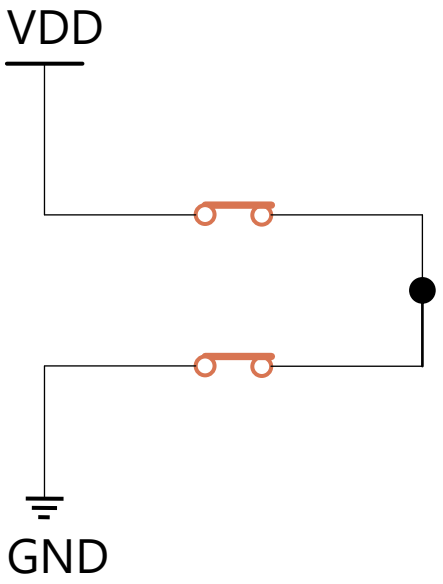
- 以下のように読み替えてもよい
  - VDD に直結している信号線の電位（=状態）は 1
  - GND に直結している信号線の電位（=状態）は 0
  - どこにも繋がっていない信号線の電位（=状態）は Z



短絡（ショート）は起きない想定

- 下記図のようにVDD と GND の双方に同時に直結することはない
  - 実際にやると VDD から GND に大電流が流れて回路が壊れる
  - なので，そう言う状態を絶対作らないように回路を組む

- もしやってしまった場合，以下の黒点は VDD と GND の中間ぐらいの電位になっていると思う

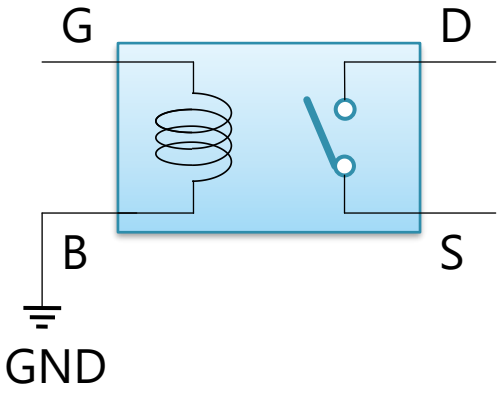


リレーを使って論理回路を作る

- 以下の２つの繋ぎ方をしたリレーを使って論理ゲートを作る
  - N 型
  - P 型

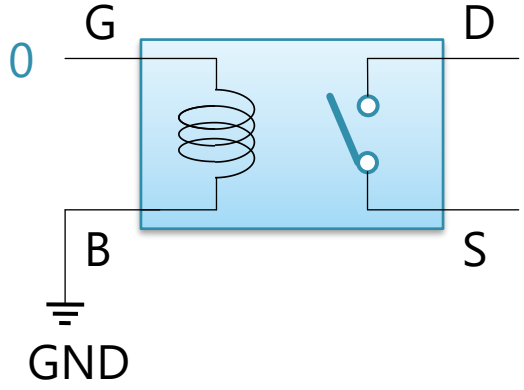
N 型リレー

- 左図のように B を GND（低電位）に接続する
  - N は Negative の略



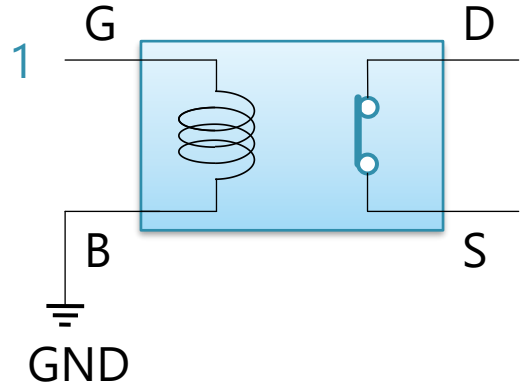
N 型の動作

- G が低電位（= 0） → スイッチ OFF
  - G~B 間は電位差がないのでコイルが働かない



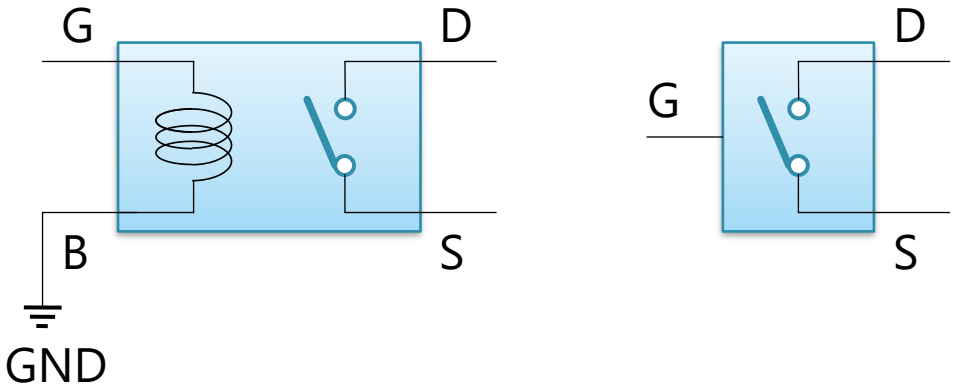
G	D~S
0	OFF
1	ON

- G が高電位（= 1） → スイッチ ON
  - G~B 間は電位差がありコイルが動作



N 型リレー

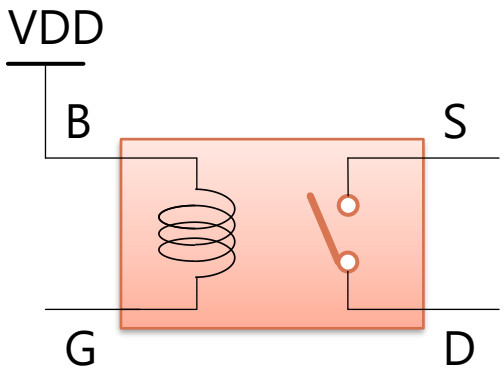
- 左の省略系として右のように描く
  - 毎回ぜんぶの場所にコイルや GND を描くとややこしい



G	D~S
0	OFF
1	ON

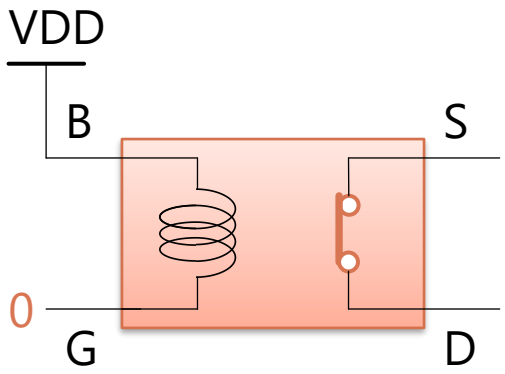
P 型リレー

- 以下のように B を VDD（高電位）に接続する
  - G が入力，D が出力
  - P は Positive の略
- 注意
  - N 型の時とは B と G，S と D が上下逆になっているのに注意
  - N 型の時は B が GND（低電位）に繋がっていた



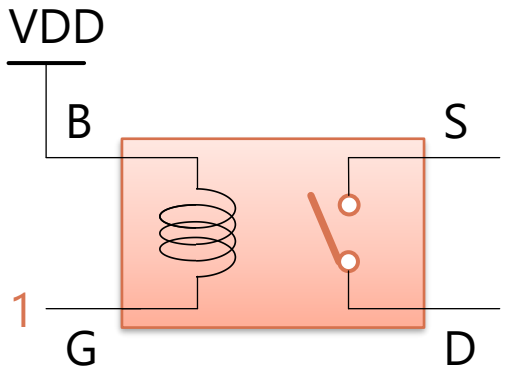
P 型の動作

- G が低電位（= 0）→ スイッチ ON
  - B~G 間に電位差があるのでコイルが動作



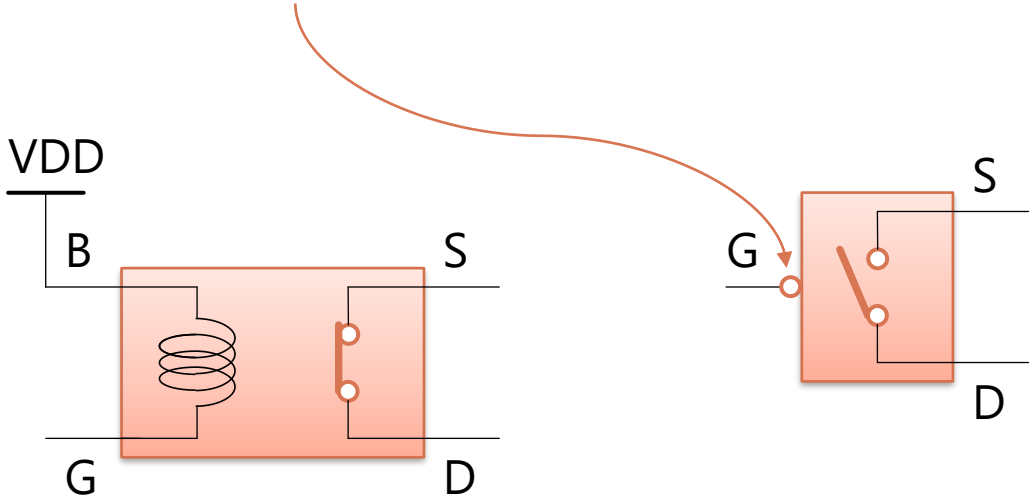
G	D~S
0	ON
1	OFF

- G が高電位（= 1）→ スイッチ OFF
  - B~G 間に電位差がないのでコイルが動作せず



P 型リレー

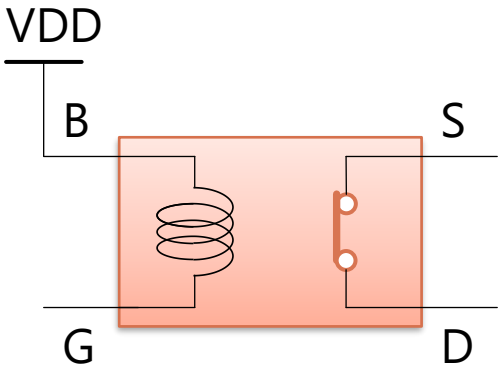
- 左の省略系として右のように描く
  - G のところに白丸があるのが N 型との描き方の違い

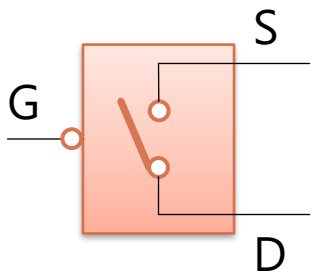


G	D~S
0	ON
1	OFF

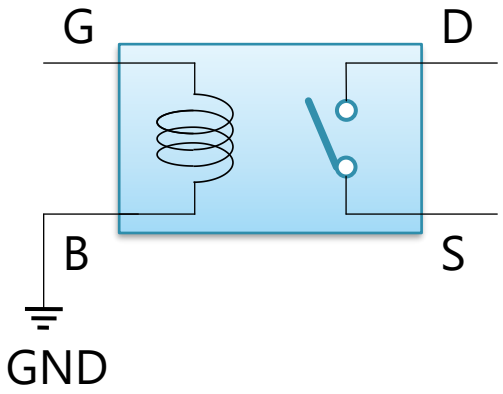
P 型と N 型のまとめ

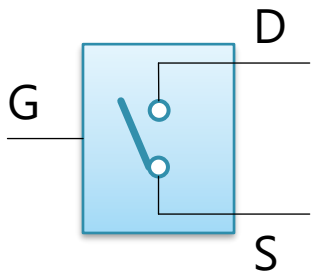
- P 型と N 型の違い：
  - P 型では G が 0 の時 ON，N 型は G が 1 のとき ON で逆
  - G に白丸があるのが P 型，ないのが N 型





G	D~S
0	ON
1	OFF



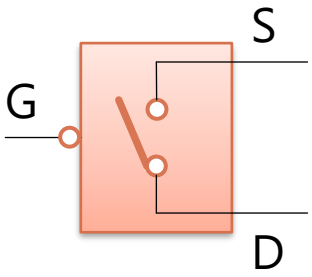


G	D~S
0	OFF
1	ON

P と N が，どっちかどっちだかわからない？

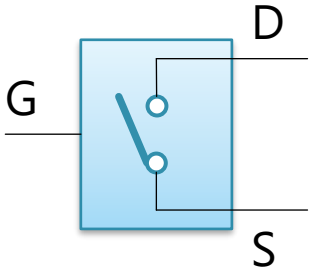
- 論理回路では白丸は反転を表す
  - 白丸なしの N型は 1 で ON ， 0 で OFF
  - 白丸があったらそこで反転するので 0 で ON
    ･･･と憶えると良いかもしれない

P 型



G	D~S
0	ON
1	OFF

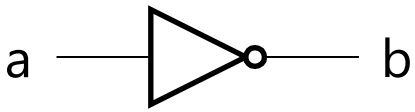
N 型



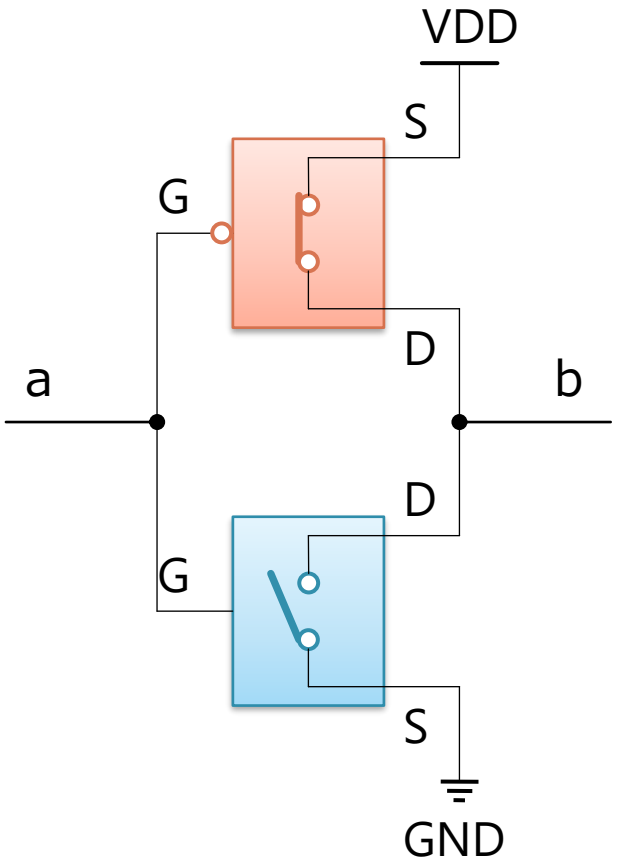
G	D~S
0	OFF
1	ON

NOT ゲート

- P 型と N 型を並べて右のように繋ぐと NOT ゲートができる



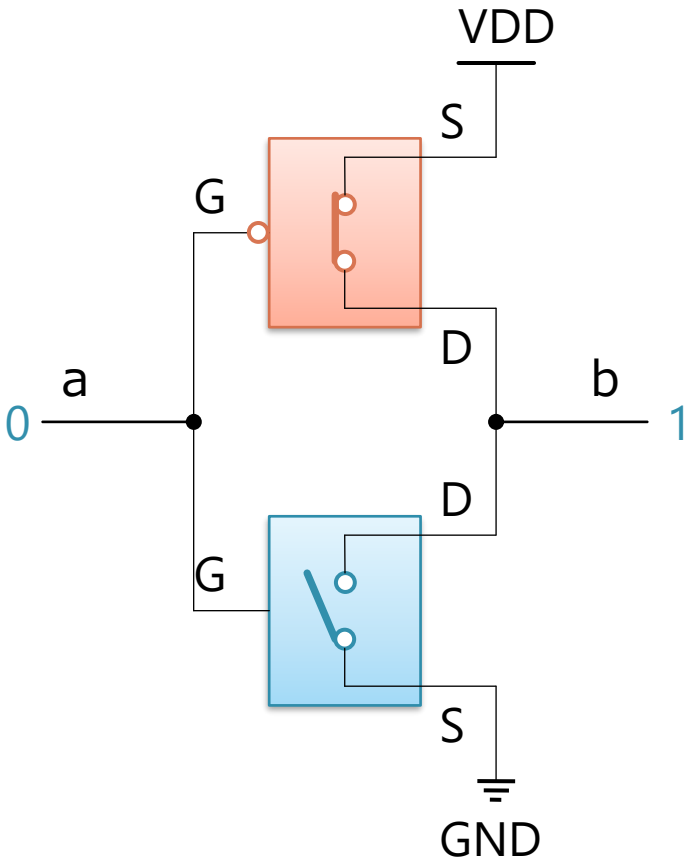
a	b
0	1
1	0

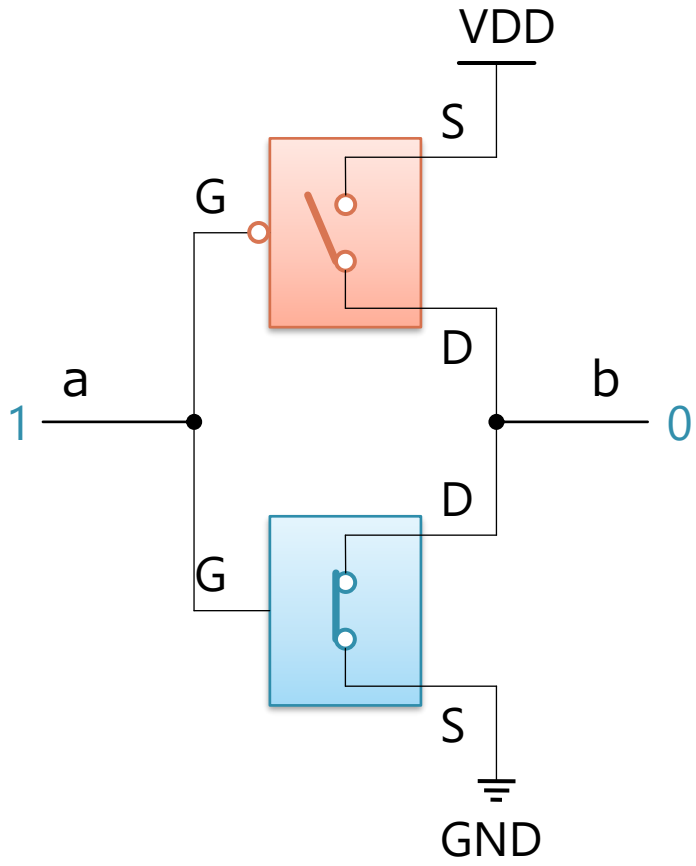


NOT ゲートの動作

- 双方のスイッチが必ず反対の動作になる
  - P 型は入力が 0 だと ON
  - N 型は入力が 1 だと ON

a	b
0	1
1	0

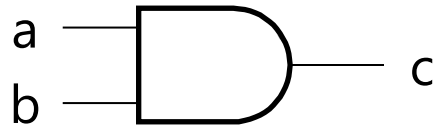




NAND ゲート

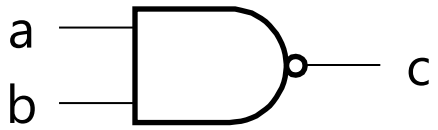
- NAND は AND の結果の NOT をとったもの
  - 左側が AND 右側が NAND
  - NAND は AND の後ろに白丸（反転）がある形で描く

AND



a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

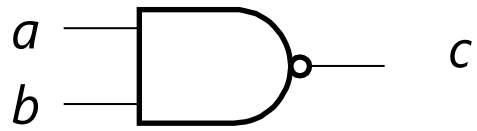
NAND



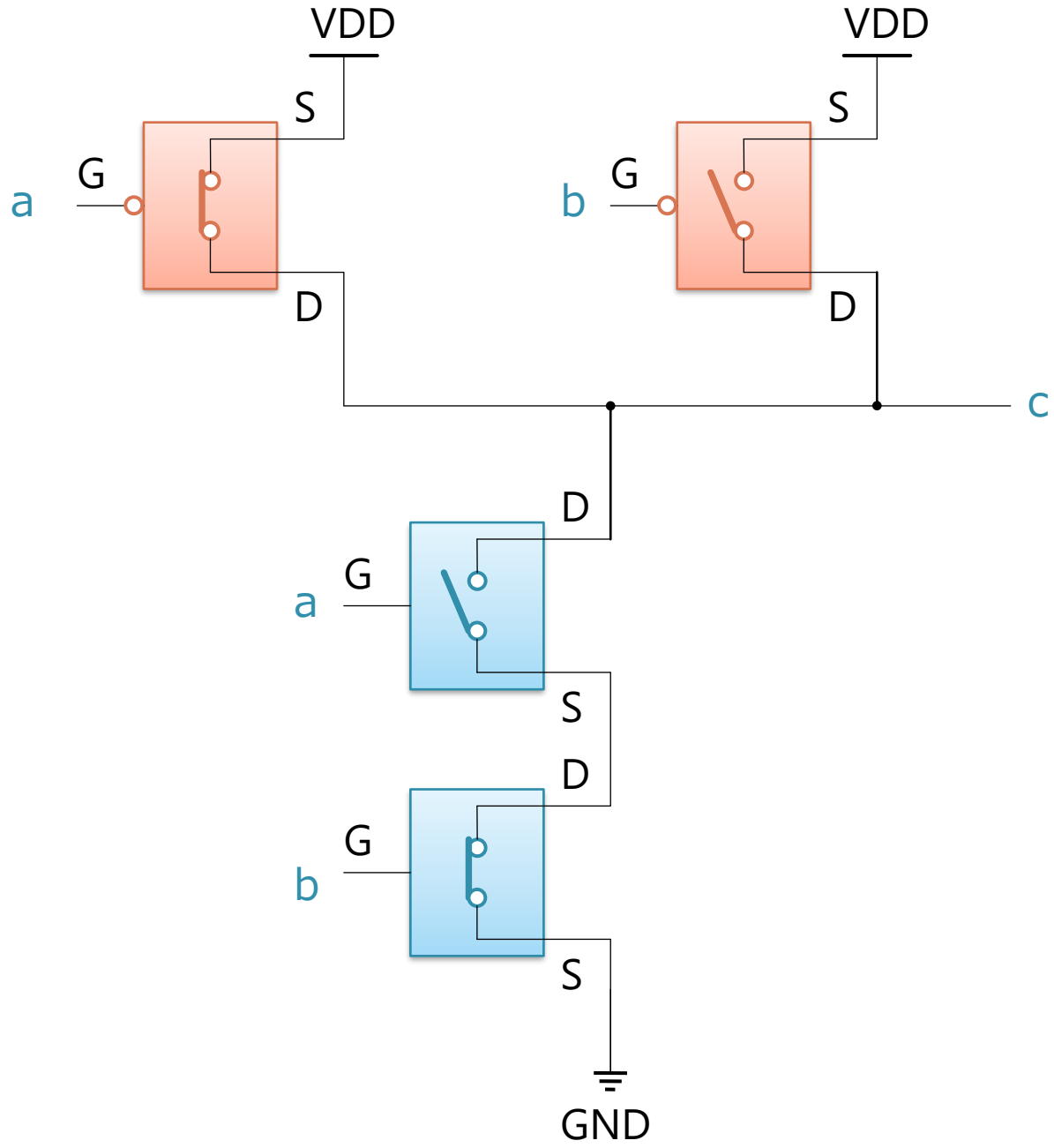
a	b	c
0	0	1
0	1	1
1	0	1
1	1	0

NAND ゲート

- 右のように接続すると NAND ゲートが作れる

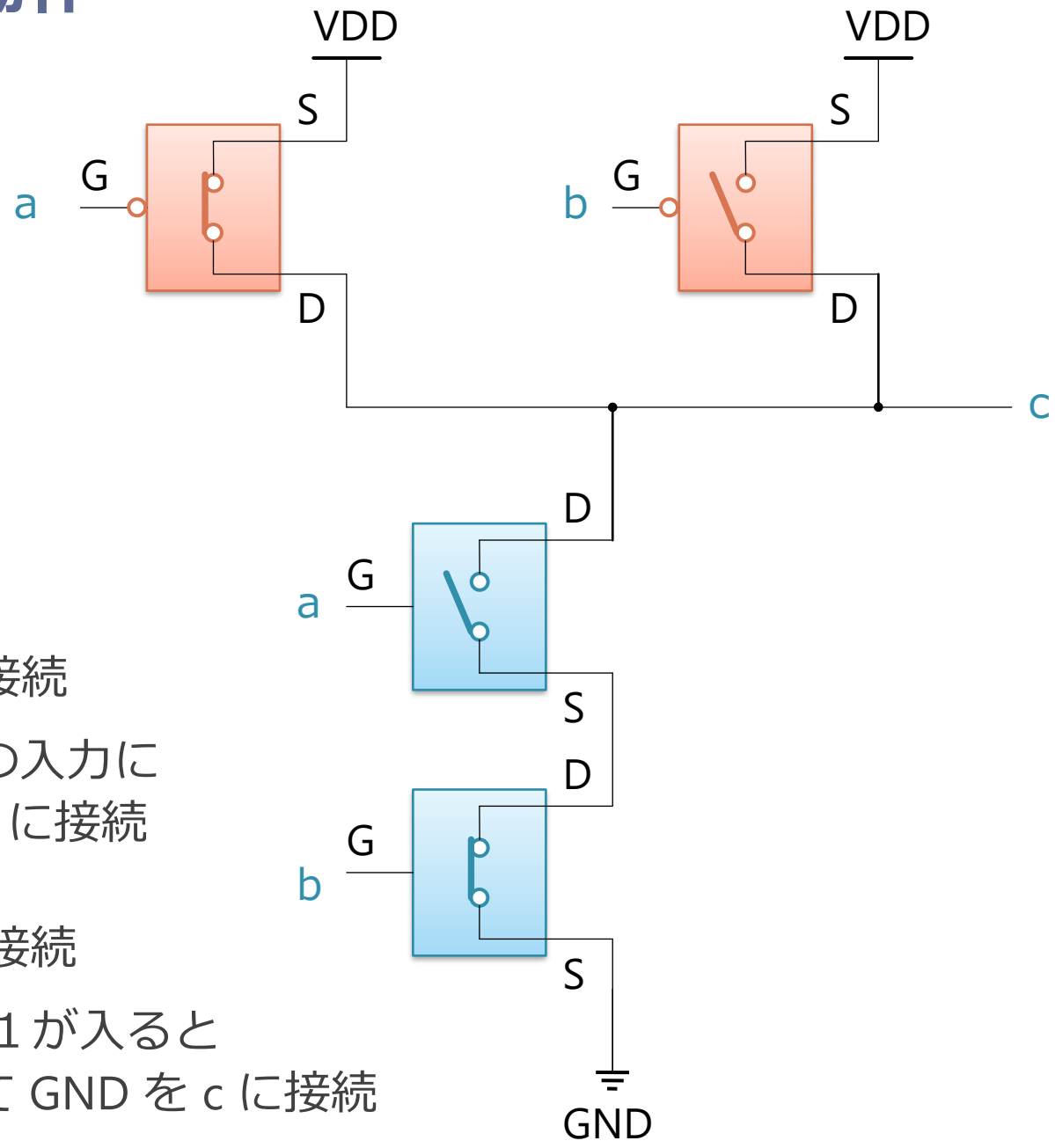


a	b	c
0	0	1
0	1	1
1	0	1
1	1	0



NAND ゲートの動作

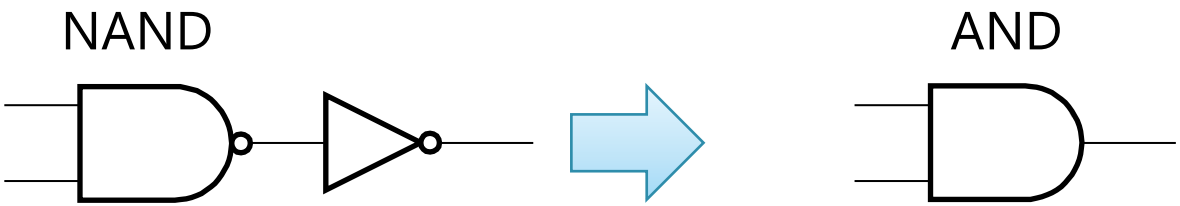
a	b	c
0	0	1
0	1	1
1	0	1
1	1	0



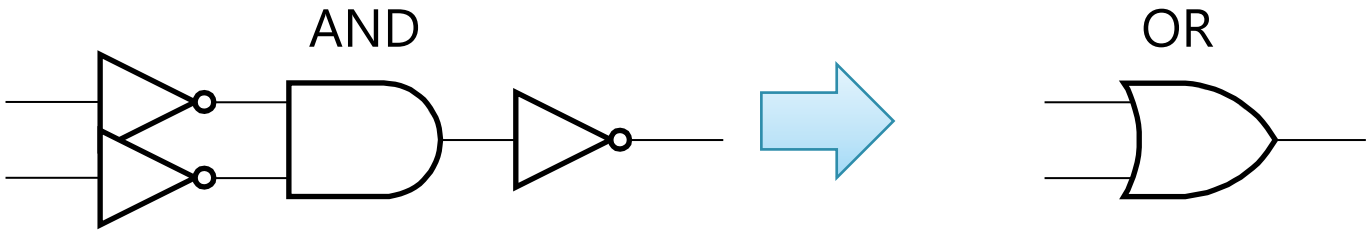
- 上側の P 型 2 つは並列接続
  - a と bどちらか片方の入力に 0 が入ると VDD を c に接続
- 下側の N 型 2 つは直列接続
  - a と b 双方の入力に 1 が入ると 2つとも ON になって GND を c に接続

NOT と NAND から AND と OR を作る

- NAND の出力に NOT をつけると AND が作れる

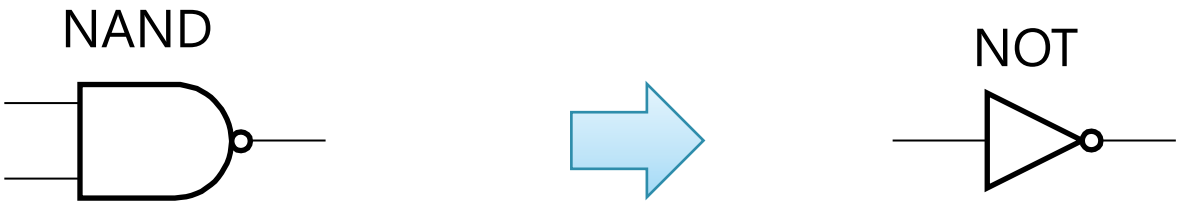


- AND の入力と出力に NOT をつけると OR が作れる
  - ド・モルガンの定理：　NOT(*a* OR *b*) = NOT(*a*) AND NOT(*b*)
  - 両辺に NOT を適用：　*a* OR *b* = NOT(NOT(*a*) AND NOT (*b*))



NAND が作れればそれだけでも良い

- NAND の 2 つの入力に同じものを入れれば NOT になる



リレーからコンピュータを作る

- リレーを使って NOT，AND，OR を作る
  1. リレーを使って P 型と N 型 のリレーを作る
  2. P 型と N 型を組み合わせて NOT と NAND を作る
  3. NOT と NAND を組み合わせて AND と OR を作る

リレーからコンピュータを作る

- リレーを使って NOT，AND，OR を作る手順
  1. リレーを使って P 型と N 型 のリレーを作る
  2. P 型と N 型を組み合わせて NOT と NAND を作る
  3. NOT と NAND を組み合わせて AND と OR を作る

- コンピュータを作る手順（前回の講義）
  1. NOT，AND，OR を使うと任意の論理関数と記憶回路が作れる
     - =任意の組み合わせ回路と順序回路が作れる
  2. 任意の組み合わせ回路と順序回路を使うと CPU の部品が作れる
     - PC，レジスタ，演算器，メモリ･･･
  3. それらを組み合わせるとコンピュータができる

リレーがあればプログラムが実行できる

- コンピュータがあれば，プログラムが実行できる
  1. コンピュータは機械語を解釈して実行できる
  2. 機械語はアセンブリ言語から変換できる
  3. アセンブリ言語は C 言語からコンパイルできる

- 途中に多数のステップがあるが，まとめると，
  - リレーがあれば，C 言語のプログラムを実行できる

一般化すると，

- **なんらかの入力に従って ON/OFF できるスイッチがあれば，それでプログラムが実行できるコンピュータが作れる**
  - 具体的な回路の組み方は，回路の種別ごとに違う事もある
  - NAND さえ出来ればこっちのもの

余談：いろいろなコンピュータ

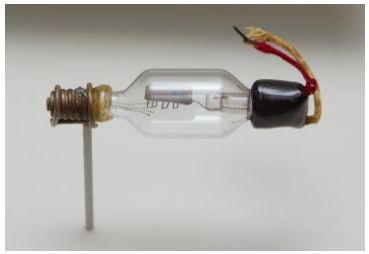
- なんらかの入力に従って ON/OFF できるスイッチさえあれば，プログラムが実行できるコンピュータが作れる
- （余談）いろいろなコンピュータ
  - リレー式
  - 真空管
  - レッドストーン回路

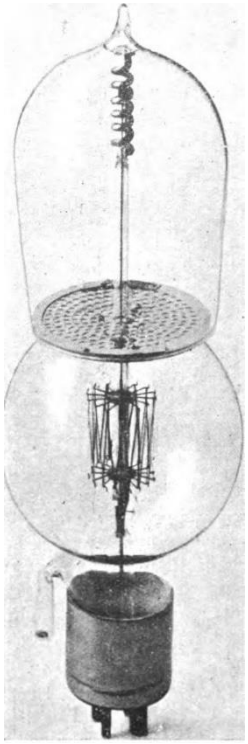
リレー式計算機 富士通 FACOM128B (1956)

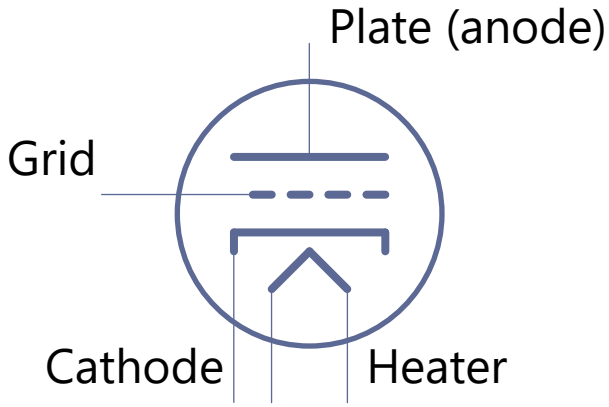
画像は https://pr.fujitsu.com/jp/news/2018/08/21.html より

真空管（三極管）

- 電球にカソードとグリッド，プレートなどの電極を追加したもの



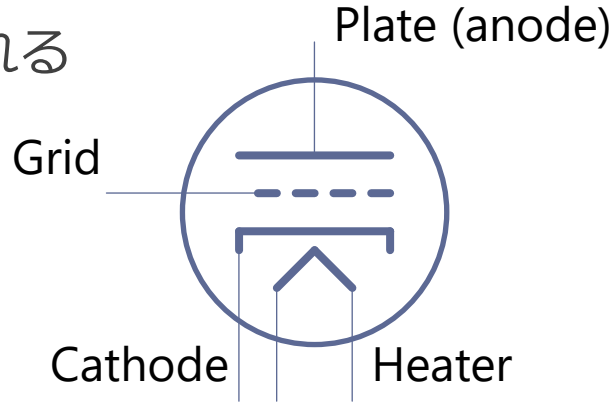




画像は https://en.wikipedia.org/wiki/Triode より
De Forest Audion tube と Lieben-Reisz tube
1900 年過ぎの発明

真空管の動作

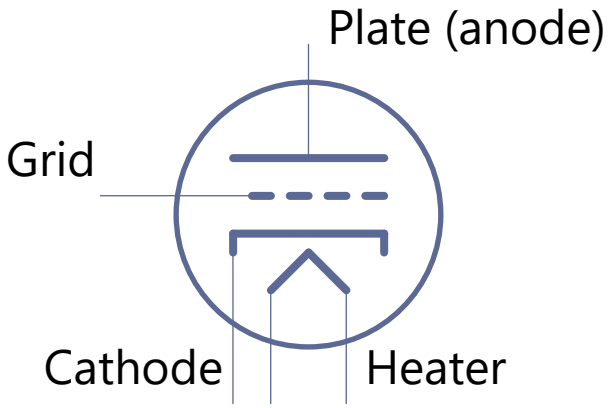
- 動作：
  1. Plate に正の電圧を，Cathode に負の電圧をかける
  2. Heater（電球のフィラメント）に電源を接続
  3. 暖められた Cathode が電子を放出
  4. Plate に電子が吸い寄せられる
     - 電子が移動する = それらの間に電流が流れる



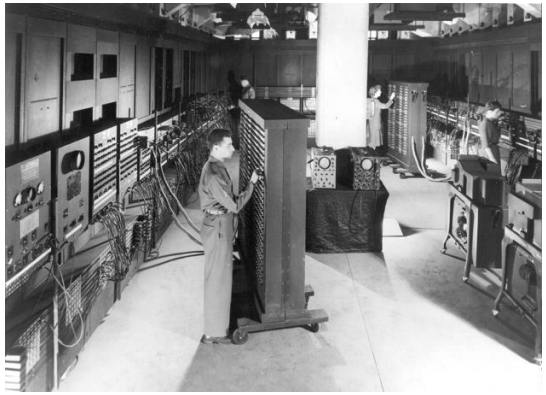
- スイッチング
  - なにもしなければ Grid は網なので電子はすき間を通過
  - Grid に負の電圧をかける
  - 負同士ではじきあうので，電子が上にいけなくなり電流が OFF

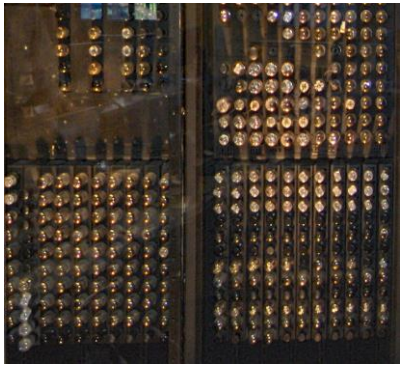
真空管

- Grid にかける電圧で，Cathode と Plate を ON/OFF できる
  - つまり，これでもコンピュータが作れる



- ENIAC (1945)
  - 世界最初のプログラムが実行できるコンピュータ
  - 18,000 個の真空管で出来ている





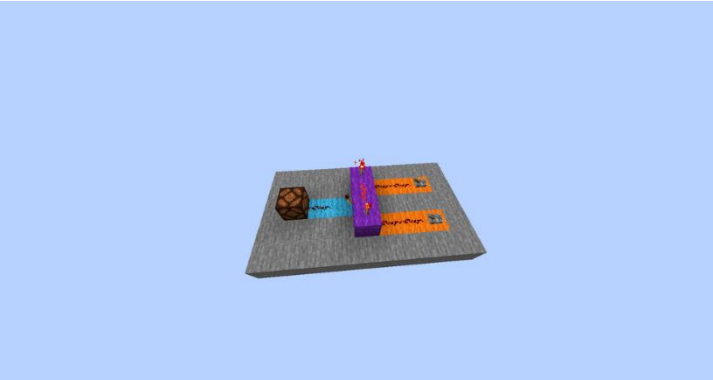
レッドストーン回路

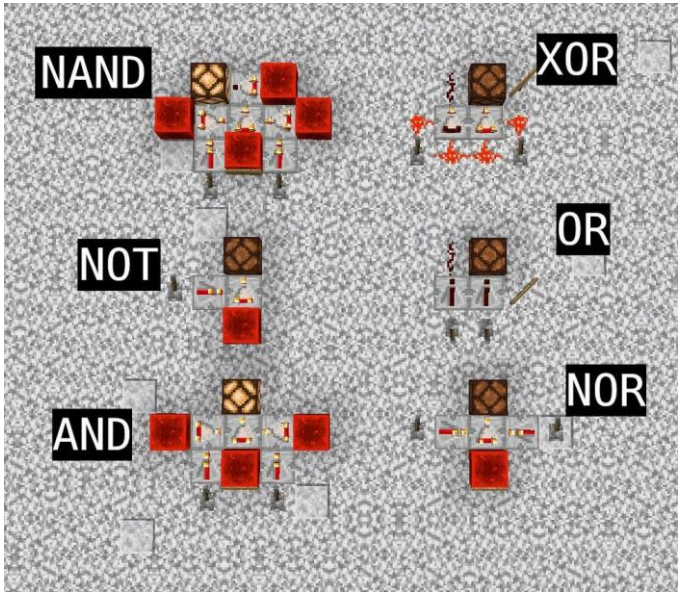
- Minecraft というゲームの中で作られている回路
  - https://www.minecraft.net/ja-jp
  - もともとは箱庭でブロックをおいて色々つくって遊ぶゲーム

レッドストーン回路

- 「レッドストーン」と言うゲーム内の要素を利用
  - 有効と無効の状態を持つ
  - スイッチと扉を繋げて，遠くから開けたり閉じたりとかする
- これも入力で出力を制御出来ていることに誰かが気づいた

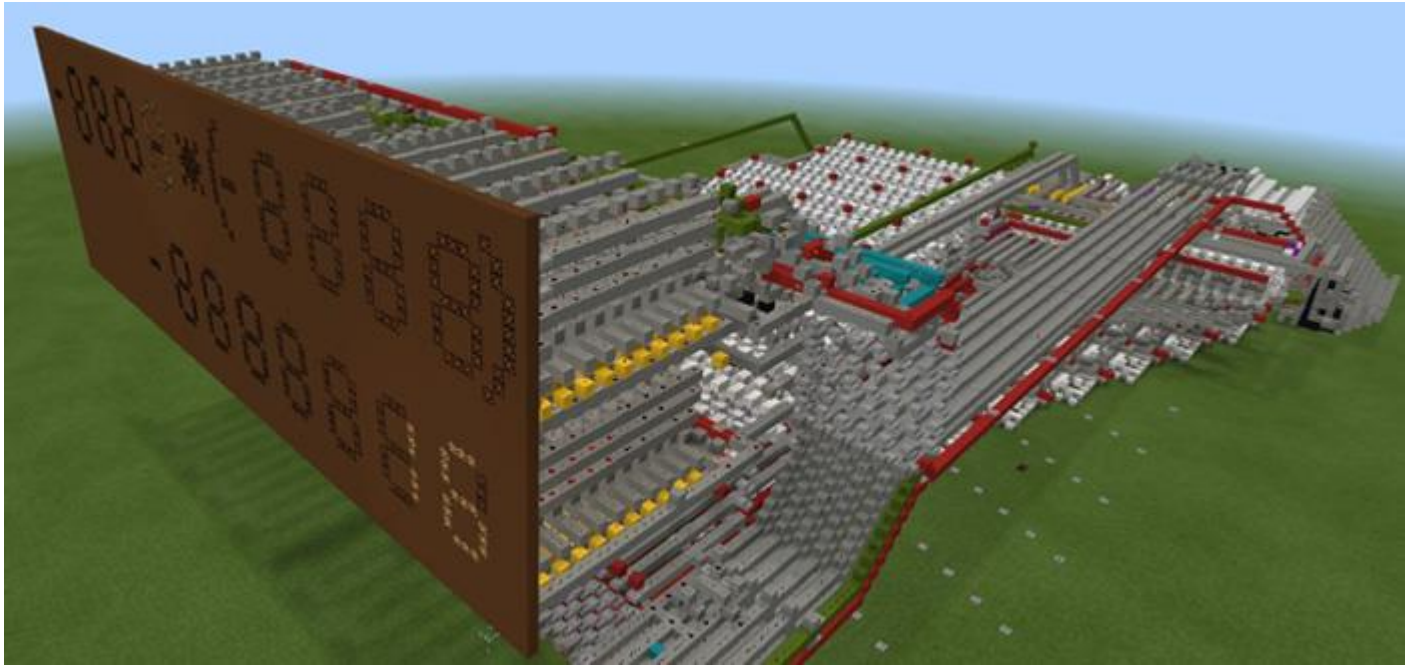
AND ゲート
右のスイッチを２つとも入れると
左側が赤くなる





画像は https://minecraft.fandom.com/wiki/Redstone_circuits/Logic より

入力で出力を制御出来ればコンピュータが作れる



画像は https://mcpedl.com/raffis-calculator-2-map/ より
四則演算ができる計算機･･･らしい

CMOS

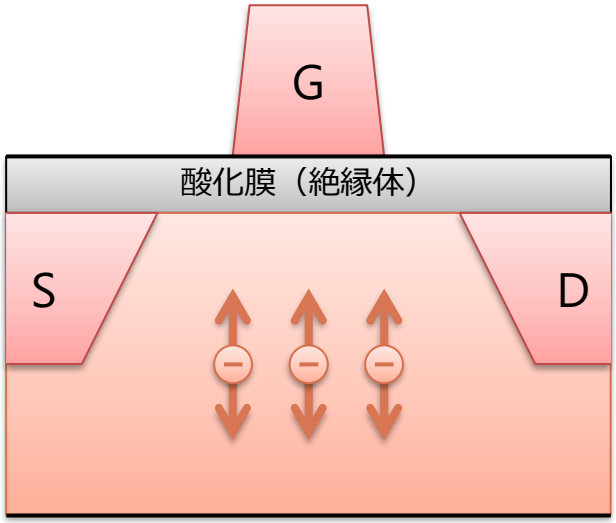
論理回路の作り方

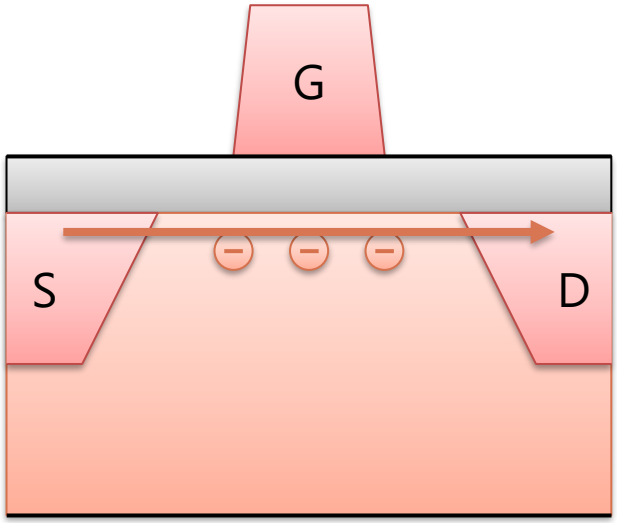
- 以下のそれぞれの仕組みを使った回路を説明
  1. リレー
  2. CMOS

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

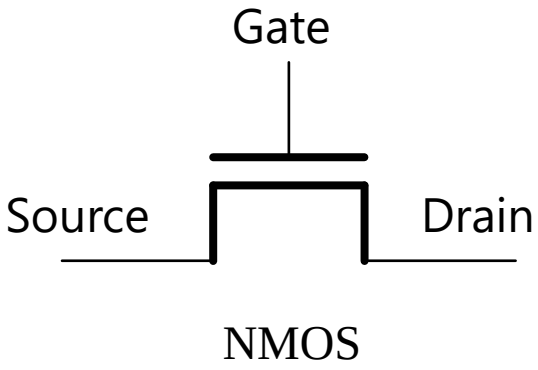
- 電界効果トランジスタ (FET: Field-Effect Transistor)
  - 電界で電子を動かしてスイッチングする
    1. G に電圧をかけてコンデンサに充電
    2. 電子の層（チャネル）酸化膜下にできて，S と D が繋がり ON に



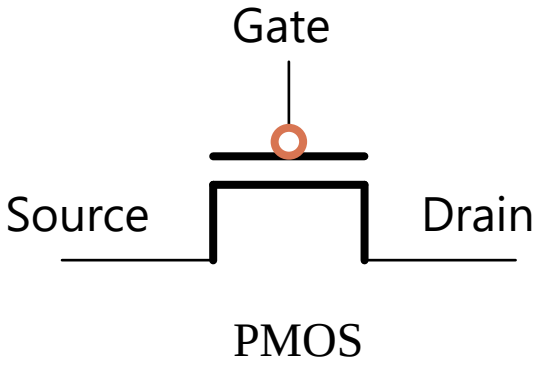


MOS には NMOS と PMOS の２つがある

- Negative (N)
  - 電子がキャリア
  - 低電位しか通せない
  - 接地側に配置

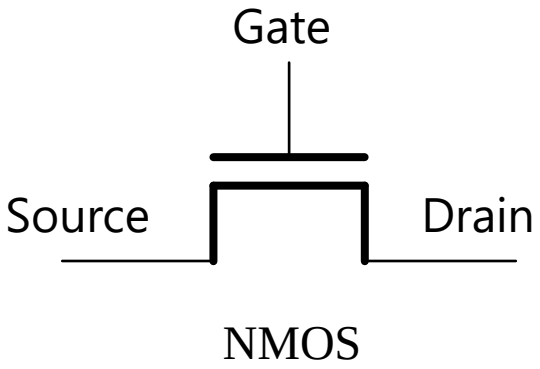


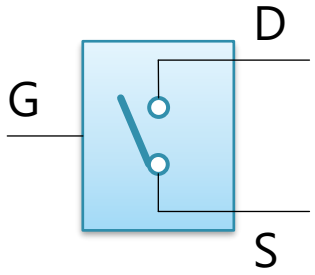
- Positive (P)
  - 正孔 (hole) がキャリア
  - 高電位しか通せない
  - 電源側に配置
  - Gate に○がついている



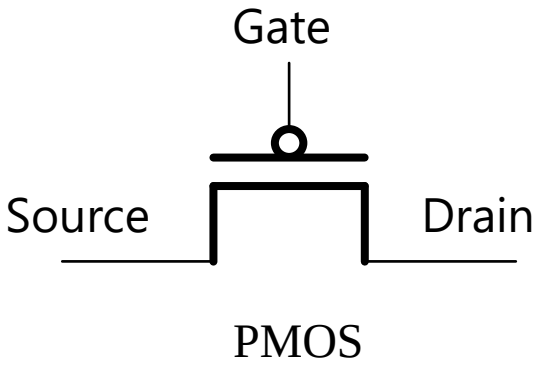
N 型/P 型リレーとほぼ同じ動作

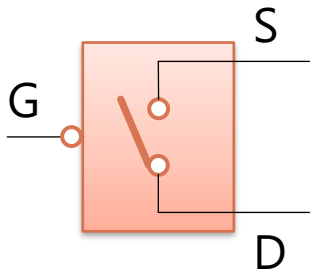
- Negative
  - Gate を 1 にすると ON になる





- Positive
  - Gate を 0 にすると ON になる

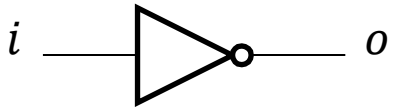


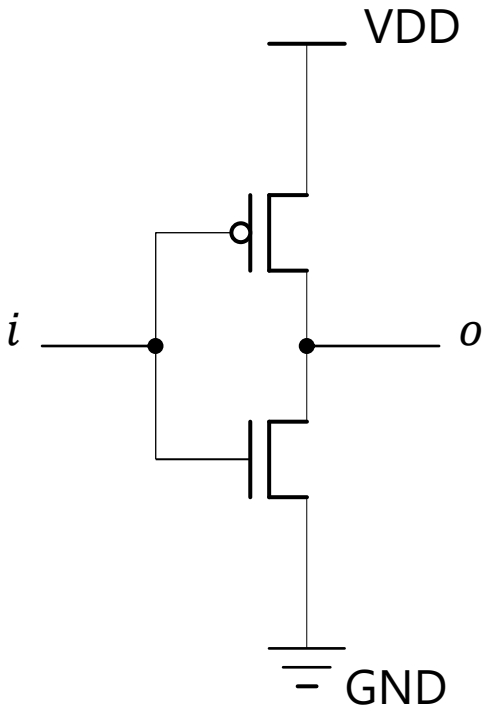


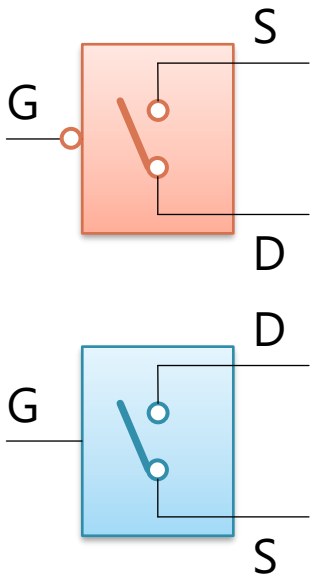
CMOS: Complementary Metal–Oxide–Semiconductor

- CMOS は NMOS と PMOS の 2 種類のトランジスタから成る
  - 電界で電荷（電子/正孔）を動かして ON/OFF する
- 基本的に N 型/P 型リレーとほぼ同じ動作
  - つまり全く同じように論理回路が組める
- 現代のコンピュータはこの CMOS を使って作られている

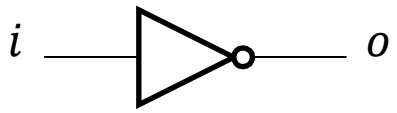
NOT ゲートの例

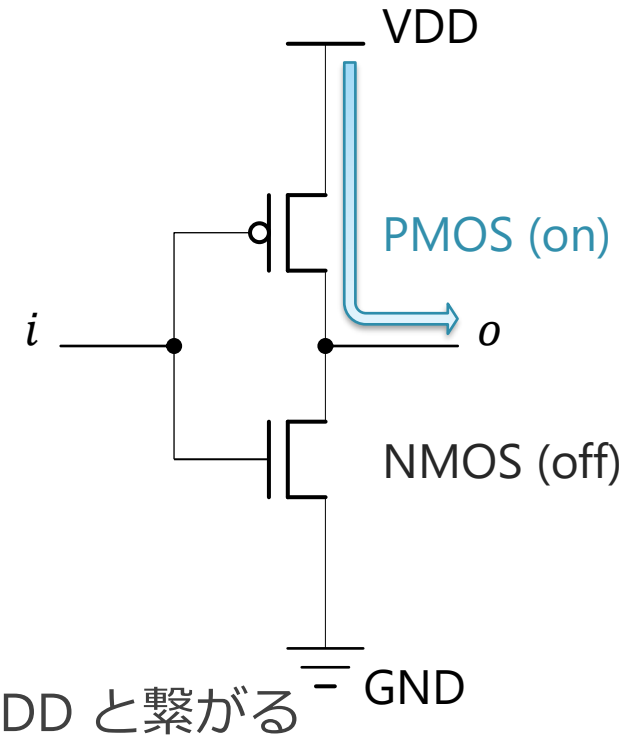






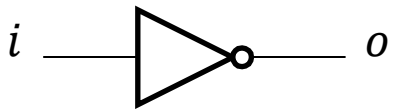
NOT ゲートの例

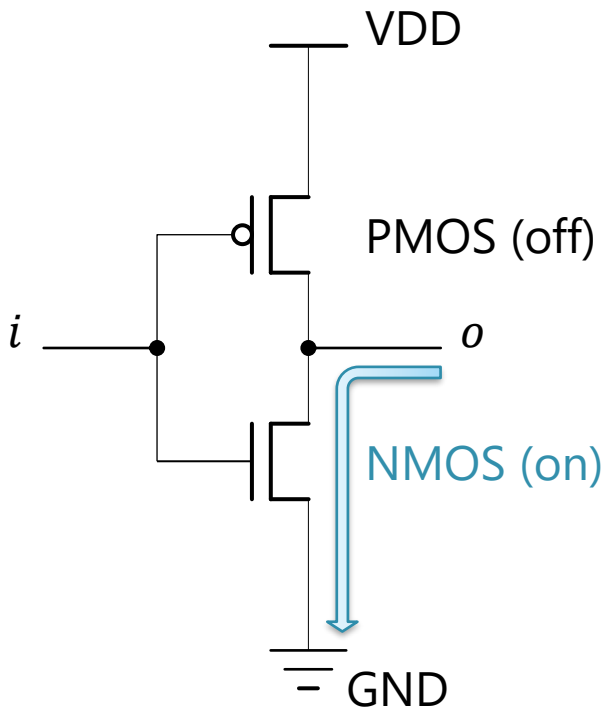




- i が 0 （低電位）の場合
  - PMOS（上側）は on = VDD と繋がる
  - NMOS（下側）は off
- 出力が 1（高電位に）

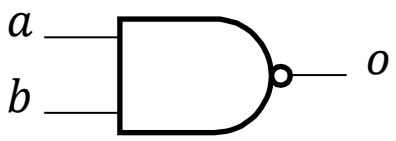
NOT ゲートの例





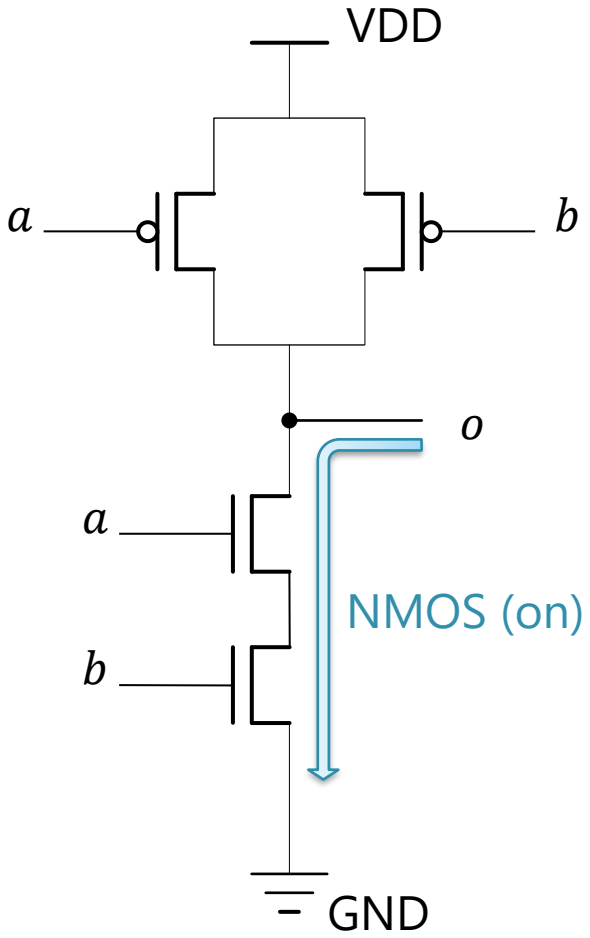
- i が 1 （高電位）の場合
  - PMOS（上側）は，off
  - NMOS（下側）は，on = GND と繋がる
- 出力が 0（低電位に）

NAND の例



$o = (a \cdot b)'$

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

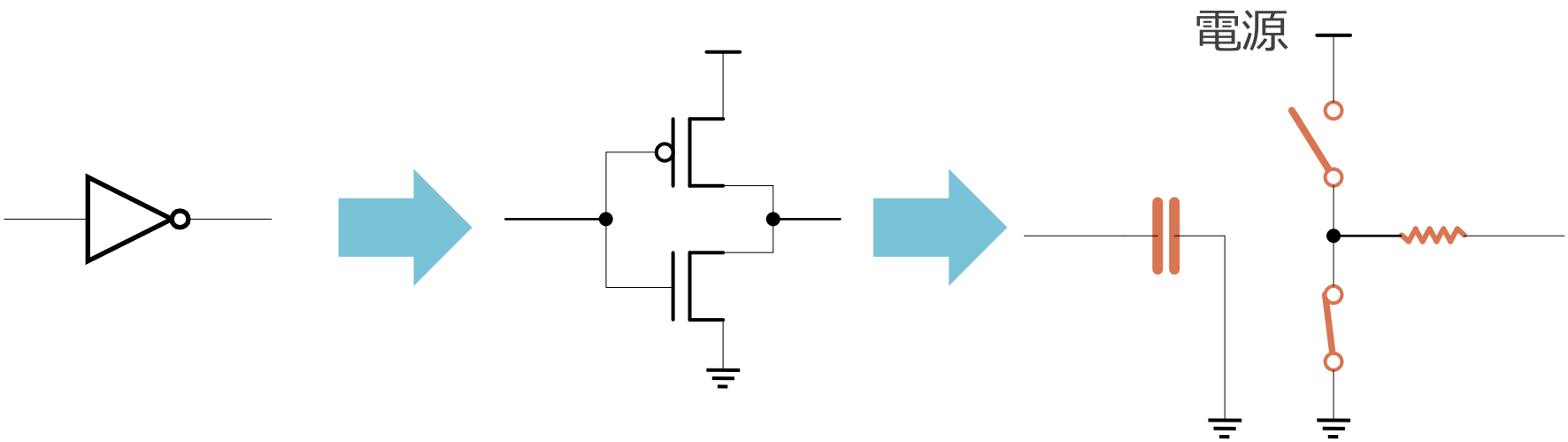


- a と b 共に 1 （高電位）の場合
  - PMOS（上側）は，off
  - NMOS（下側）は，双方 on = GND と繋がる
- 出力が 0（低電位に）

CMOS による論理回路の実現のまとめ

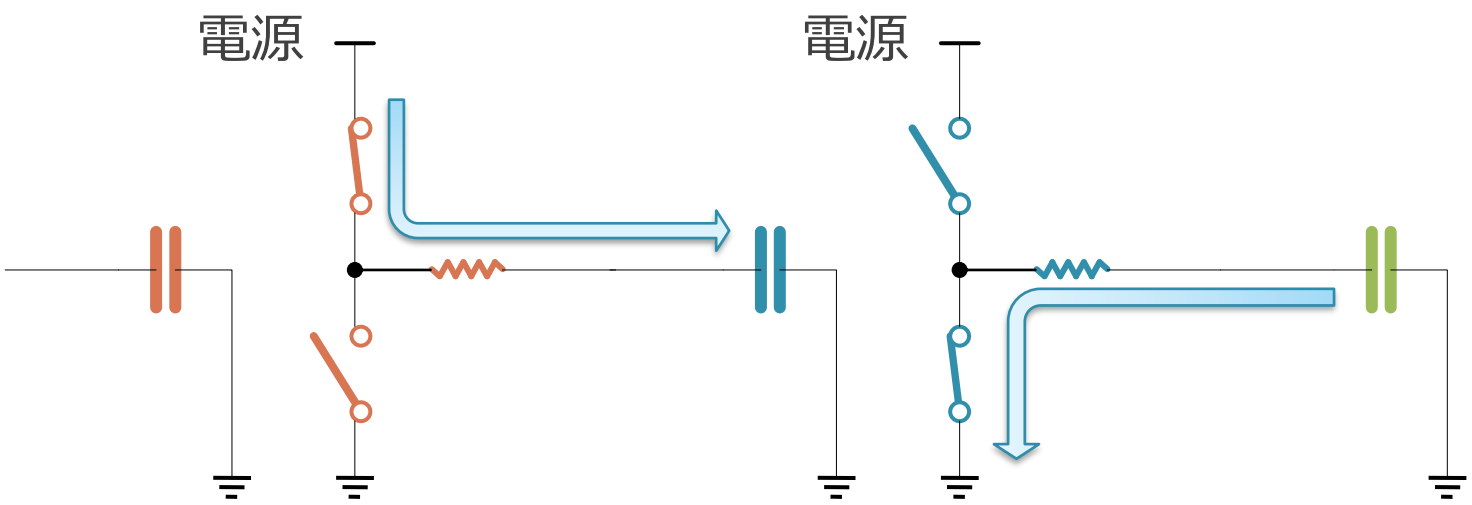
- 全ての論理ゲートは,
  NMOS/PMOS の組み合わせによって構成されている
- リレーの時と同じ

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
- コンデンサ：
  - 電荷を溜めることが出来るもの
  - 電子を引き寄せてチャネルを作る = 充電している

CMOS ゲートの遅延の実体

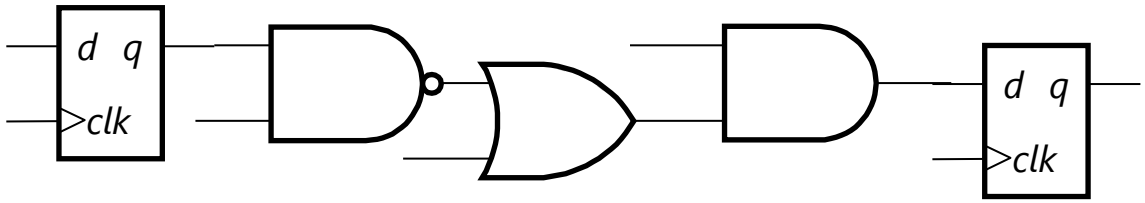


- 遅延：コンデンサの充放電にかかる時間
  1. あるゲートのスイッチが切り替わる
  2. 次の段のゲートへの充放電が開始
  3. 次の段のスイッチが切り替わる
  4. ・・・

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

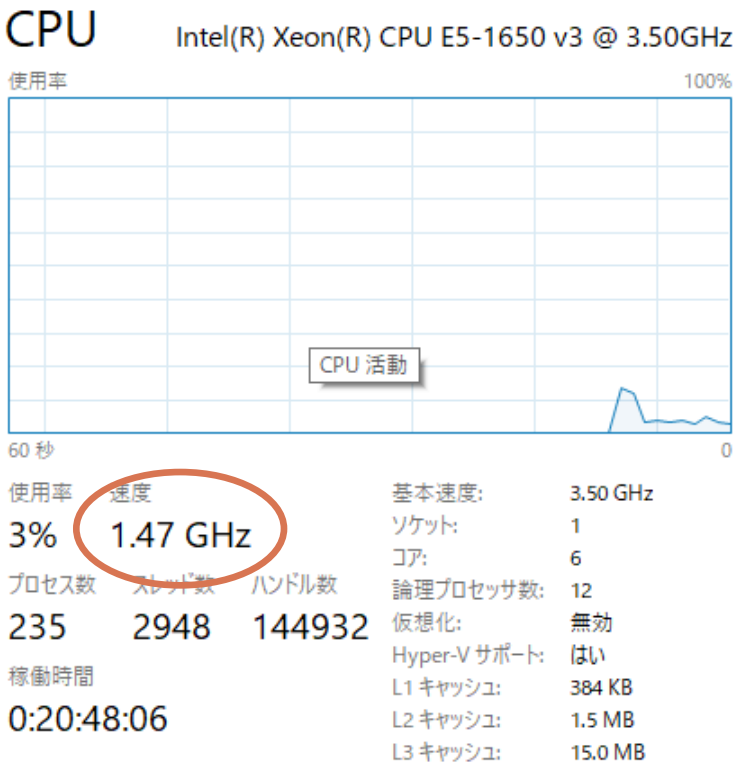
遅延の違い

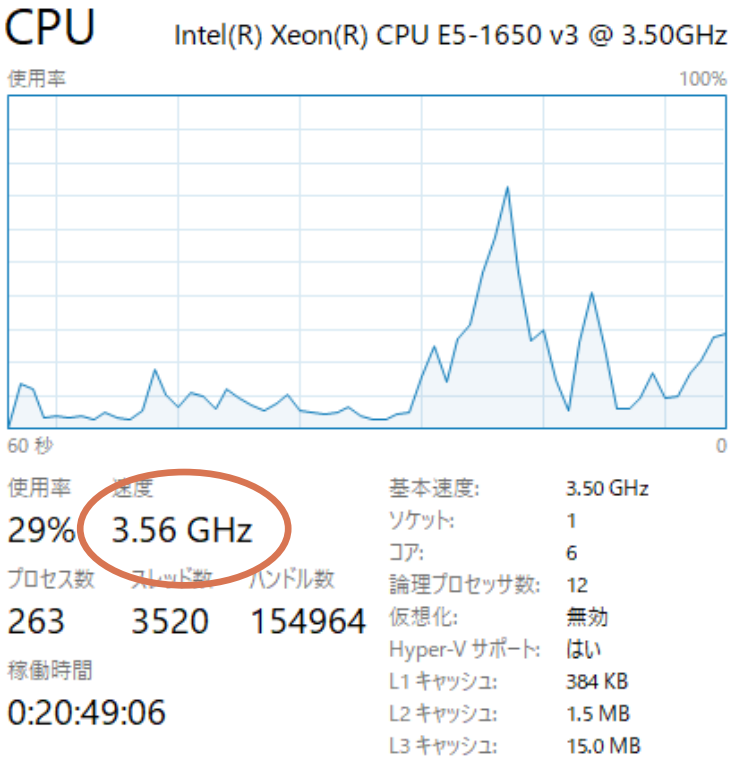


- 左側の D-FF から何個もゲートを経て右の D-FF に接続（遅い）
  - q が切り替わると，
  - 各ゲートのスイッチ（充放電）が順々に行われて，
  - 右側の d に伝わる

- 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

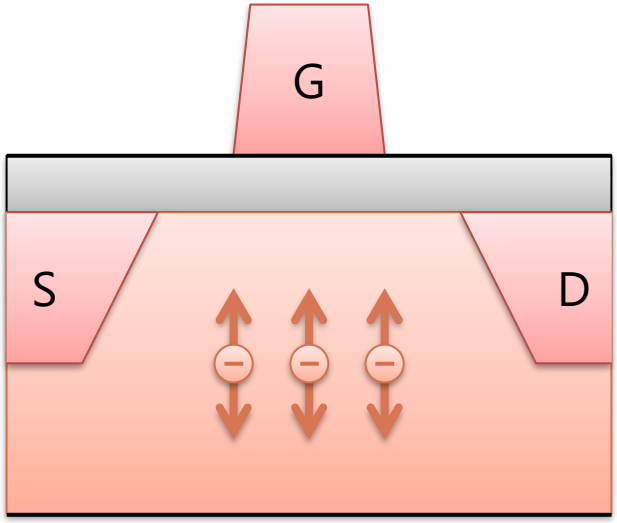
余談：DVFS (Dynamic Voltage Frequency Scaling)





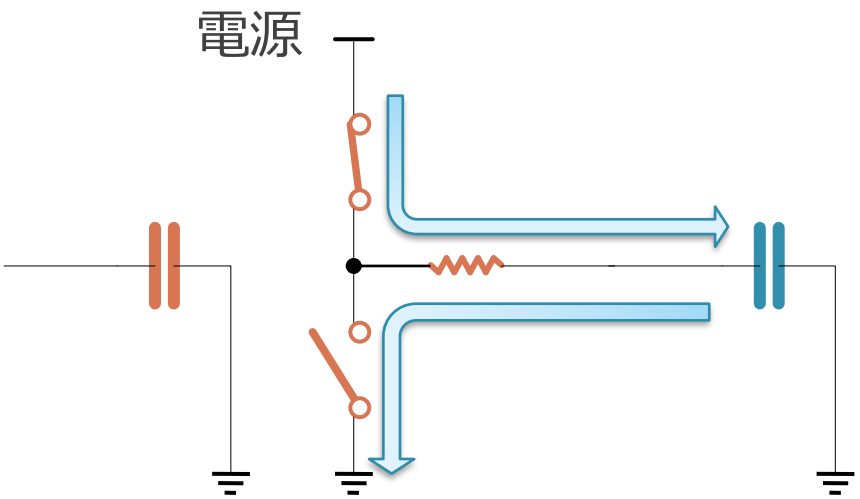
- CPU は通常，負荷に応じて動作周波数を変えている
  - 消費電力削減のため
  - この時，電圧も同時に操作している

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：一番下から積み上げて一番上までいけた

一番下がすげ変わっても，上は同様に作れる

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　…

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

CMOS/リレー/真空管/レッドストーン etc...

リレーや真空管と比較した CMOS の利点

- 超小さい：CMOS はナノメータ単位ぐらいの大きさで作れる
  - リレーや真空管は頑張っても数 cm ぐらい
- 超速い：CMOS はナノ秒以下で ON/OFF できる
  - リレーは物理的に動くので，ミリ秒ぐらいはかかる
  - リレーよりマシだが電子の移動距離が長いので遅い
- 超省電力
  - リレーは物理的に動くので結構電気を食う
  - 真空管は常に暖め続けるのでめっちゃ電気を食う
- 壊れない：
  - リレーは物理的に動くので，かなり壊れる
  - 真空管はフィラメントがそのうち焼き切れるし，ガラスが割れる

まとめ

- 論理回路の実現方法
  - リレー
  - CMOS
- 論理回路の遅延と消費エネルギー
  - CMOS の場合，充放電にかかる時間とエネルギー

課題 4

- 第１回の講義資料を参考に，１０から０までを下りながら数える以下のループをアセンブリ言語で書け
  - 使用する命令セットは第２回の講義資料のものに準じる
- ヒント：
  - 減算には add の代わりに sub を使う
  - 初期値は li 命令で設定
  - 講義中の for 文の例のようにメモリ上に i を置くとややこしいので，レジスタの上で全て終わらした方が楽

```
1: for (i = 10; i >= 0; i--) {
2: }
```

要はこういう感じのものを出してほしい

```
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
```

提出方法

- 以下の２つを提出：
  - 課題の提出は Moodle の「課題４」のところからお願いします
  - 感想や質問を「感想や質問」のところに投稿してください
    - わからない場所がある場合，具体的に書いてもらえると良いです
- 提出締め切り
  - 5/21 日曜日の 23:59 まで
- 注意：
  - 課題の出来は，ある程度努力したあとがあれば良しです
  - 必ずしも正解していなくても良いです

付録

実際の回路生成

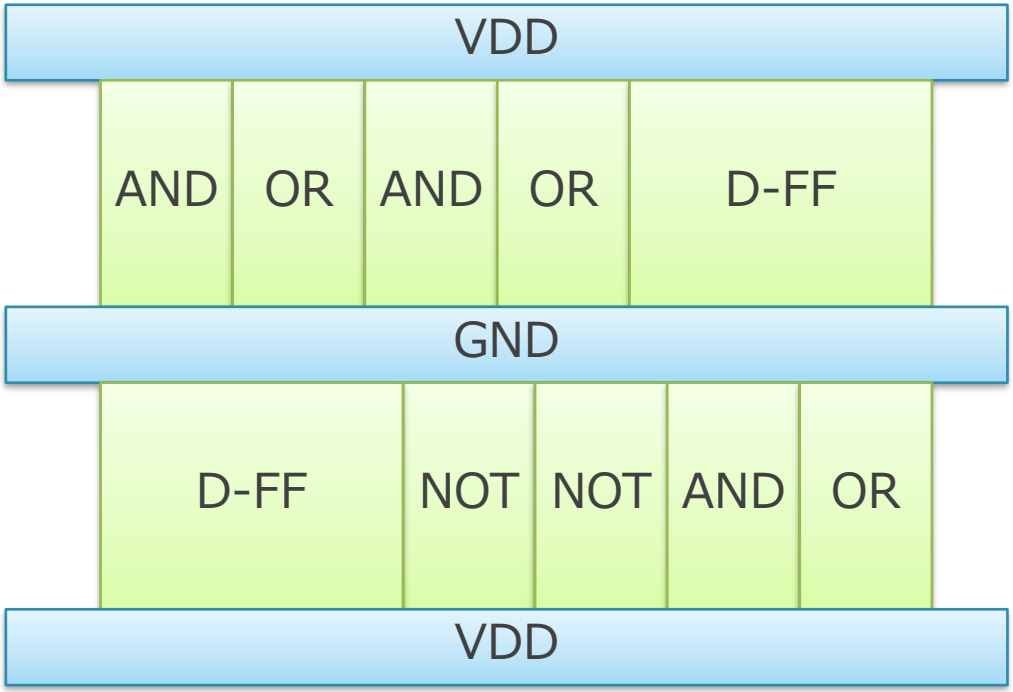
```
if code == 100:
    out = rs1 xor rs2
elif code == 110:
    out = rs1 or  rs2
else:
    out = rs1 and rs2
```

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

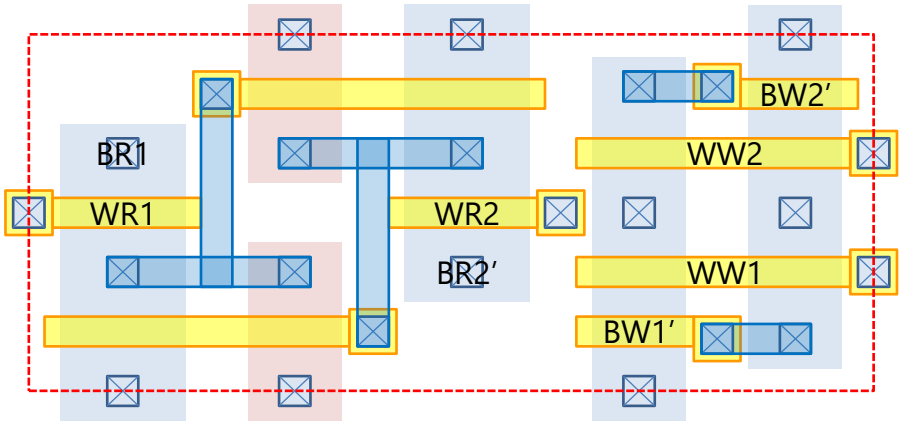
1. 論理合成：
    1. ハードウェア記述言語から，論理関数を生成
    2. 生成された論理関数を最適化
        - カルノー図：４入力ぐらいが限界
        - クワイン・マクラスキー法：入力数が増えると計算量が爆発
        - さまざまな最適化アルゴリズムが提案・使用されている
    3. 回路のプリミティブと接続関係を生成
        - AND, OR, NOT, D-FF を始めとして，よく現れる回路の部品が用意される
2. 配置・配線：
    1. 回路のプリミティブを配置・配線

スタンダード・セル



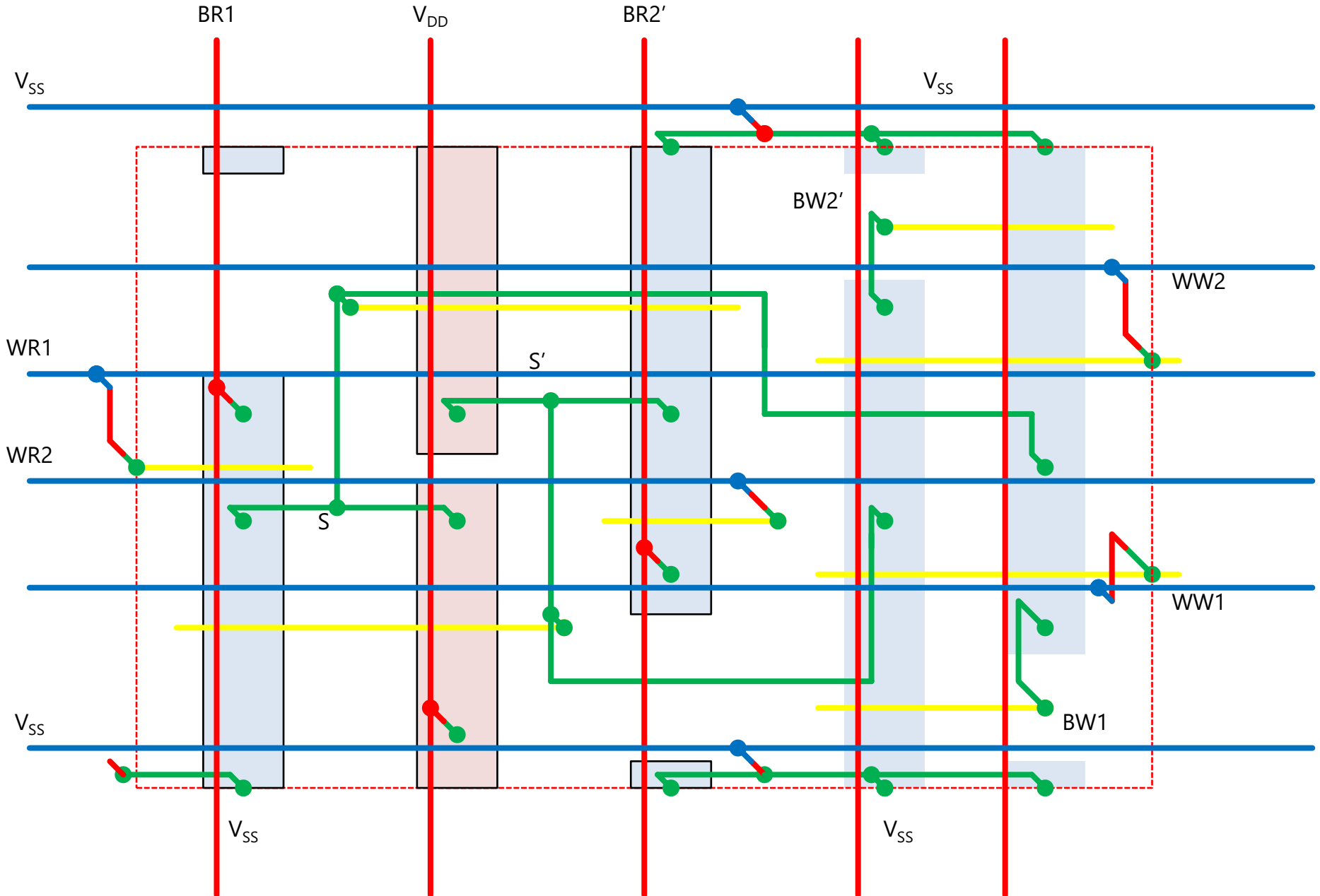
- スタンダード・セル：
  - AND や OR のようなプリミティブとなる回路「セル」を用意
    - 高さを幅を揃えておいて，簡単に並べられるように作ってある
  - これらを配置して，配線を接続することにより回路を実現
    - 配置配線という

フルカスタム・レイアウト

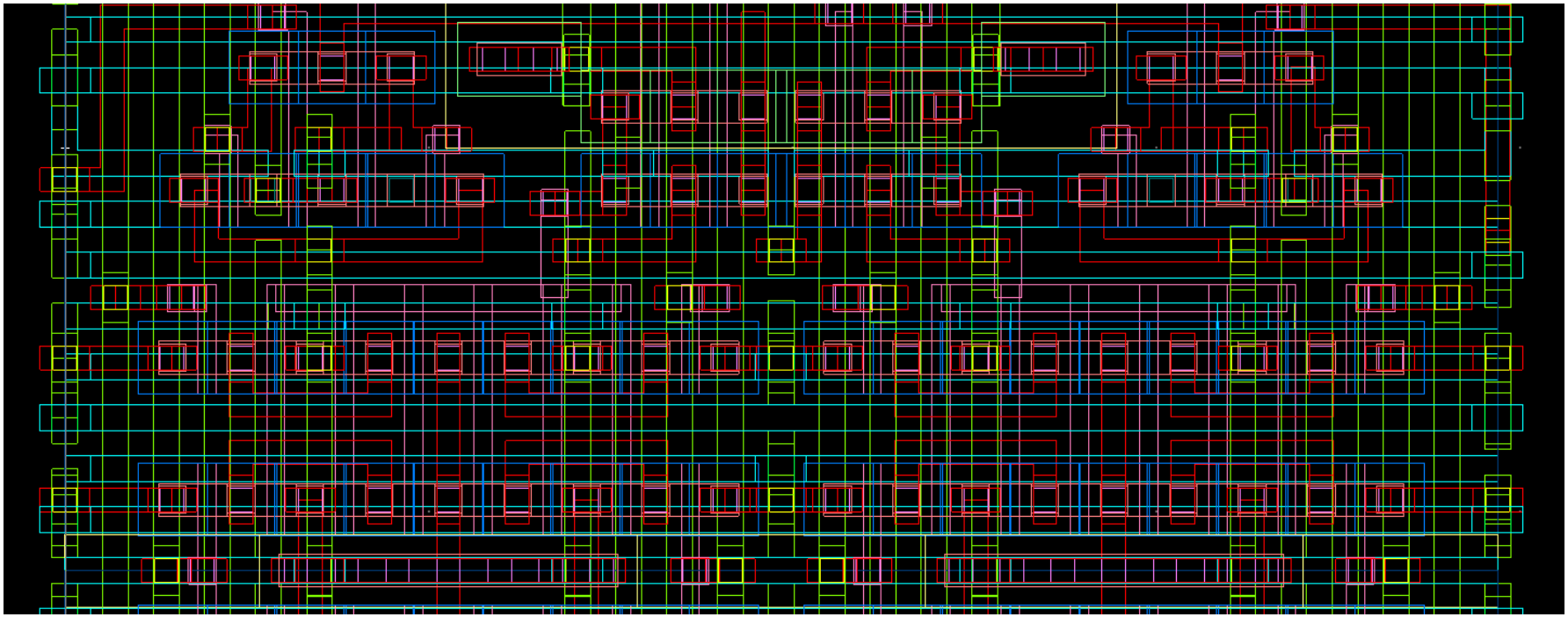


- 各スタンダード・セル（ゲート）は複数のトランジスタにより構成
- フルカスタム・レイアウト：
  - 半導体チップ上に１つ１つトランジスタをお絵かきして設計
    - 形状が特定のパターンを満たすと，トランジスタとして機能
  - 尋常じゃない手間がかかる
- 今はスタセルの中身と，本当に性能が出したい部分だけこれで作る
  - ごく一部だけアセンブリ言語でプログラムするようなもの

3次元的な構造を考えながら，設計

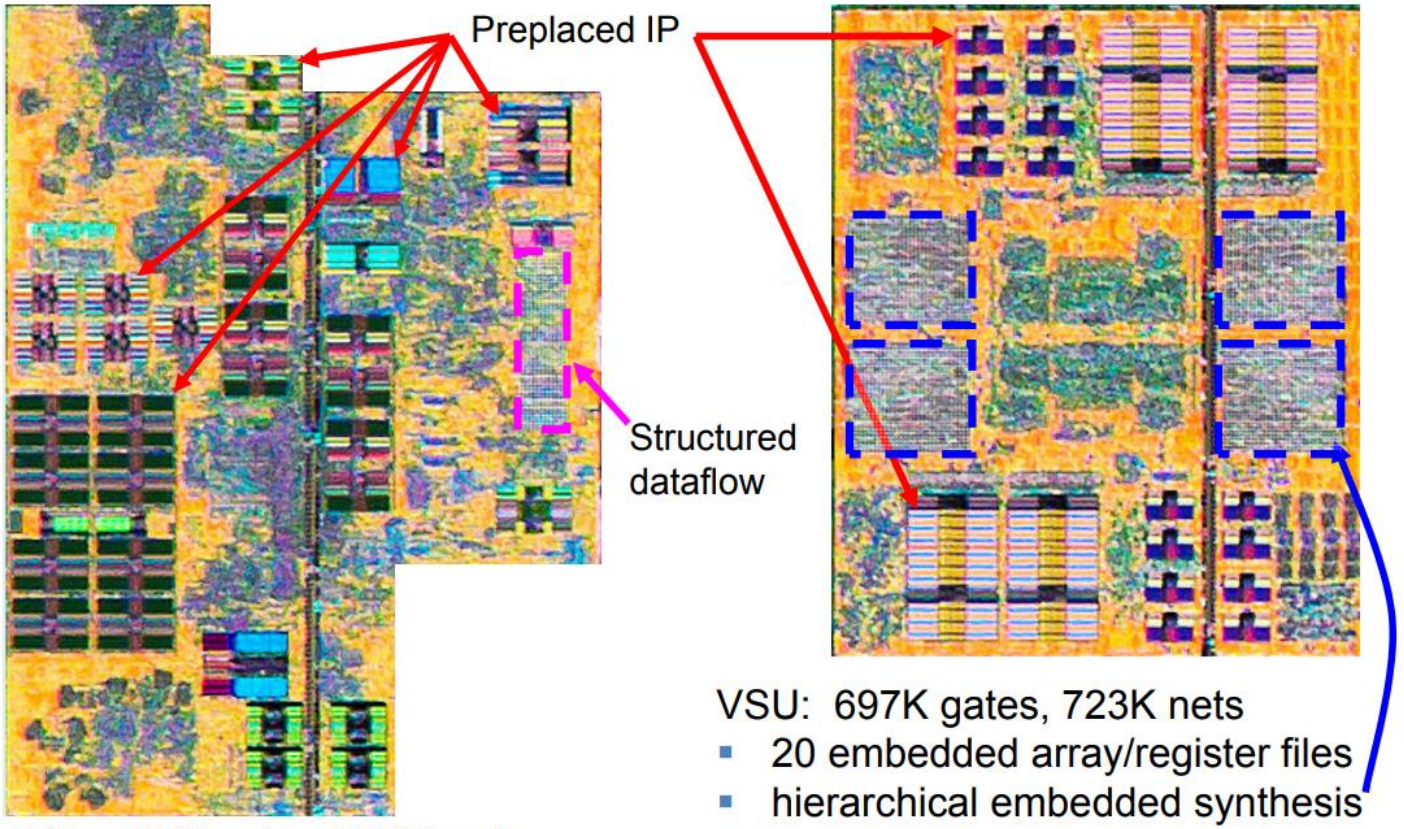


実際の設計データ



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