

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

質問や感想など

- 授業内容とは関係ないですが、授業の資料（スライド等）はどこに載せてますか？復習時に使いたいのので載せてくださるとありがたいです
 - シラバスに書いています
 - <https://github.com/shioyadan/otya-computer-architecture-i>

質問や感想など

- 中間考査は実施されますか？また、昨年度の考査と大きく変わらない問題・形式にする予定でしょうか？
 - まだ未定です
 - ただ，形式は結構かわると思います

質問や感想など

- 今回の講義で主に扱われていた命令などは、ハードではなくソフトの話のように思えるのですが、ハードの話なのですか？
 - 命令に関わる話はソフトとハードの境界面にあります
 - 強いて言えば,
 - 命令 = ソフト
 - 命令の定義 = ハード

質問や感想など

- 数理基礎論で「ゲーデル数」という論理記号と自然数を1対1対応させる考えが登場していたのですが、これとアーキテクチャの命令を数字の列に変換する操作が対応しているのか気になりました。

質問や感想など

- 辞書やインターネットで「アセンブリ言語」と調べたところ、「機械語を記号化した言語」や「プログラミング言語の一種」と説明されていることが多かったのですが、バイナリだけを機械語と呼び、アセンブリ言語もプログラミング言語に含めることもあるのでしょうか。

質問や感想など

- CPUの動作を料理にたとえたら、流れがわかりました。読み出して実行して書き出しての単純な流れで計算機の主な部分が動いていると思うとコンピュータはそこまでブラックボックスではないのではと思えました。

質問や感想など

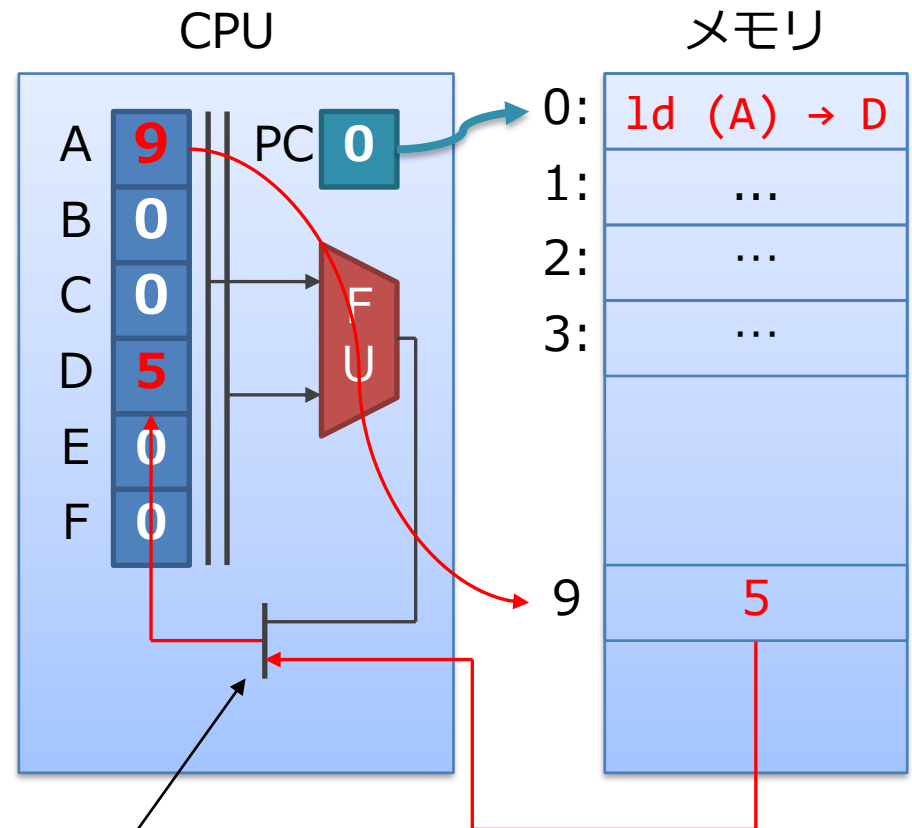
- p57、p58でロード命令ではメモリからレジスタに矢印が直接向いてなかったのに(何か別の場所を経由していた)ストア命令ではレジスタからメモリに直接矢印が向かっているのはなぜですか。

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



レジスタの書き込み口は1つなので, 演算器の結果と選択してから書き込む

- 単に「メモリ」という場合、メモリ一般に関することを言っているのでしょうか。それとも主記憶装置だけですか。

質問や感想など

- バイナリは数字の列ということでしたが、一対一に対応する命令が10を超えた場合、どうやって数字の列だけで表すのかが気になります。

- $A+B-C$ を計算する場合前の2つから先に計算するとのことでしたが、 $A*(B-C)$ などの計算順序が決まっている計算は()もなにかしらのバイナリに変換されて判断されるのでしょうか？

- C言語の制御構文が基本的にif~gotoに書き換え可能とのことでしたが、具体的にどのように書き換えられるのか気になりました。

質問や感想など

- returnはif～gotoで書き換えられない理由として、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だからというのがよく分かりませんでした。liなどで保存できるように感じてしまいます。
 - 同じ関数を、異なる場所から呼ぶことがあるため
 - A から呼ばれたのか、B から呼ばれたのかで戻り先が異なる
 - どこから飛んできたのか = 飛び元の命令アドレス（PC）の保存が必要

- 私は少しC言語が苦手で、全然理解できていない状態で授業を受けているのですが、大丈夫でしょうか。

- LABELやgoto LABELはプログラムでは省略できるということですか？

質問や感想など

- c言語におけるアドレスやポインタについてがあまりはっきり理解できていなかったけど、ロード命令やストア命令でアドレスの考え方をすると知って別方面からc言語を理解するきっかけになった気がしました。
- レジスタなしでも演算はできると知って、c言語でもアドレスなしでプログラムを書けるのかなと思いました。

- C言語を実行する為にはCPUにこれだけの命令があれば良いというお話がありましたが、新しいプログラミング言語を使う際、PCにその言語をインストールするのは、その命令を入れているということなのですか？
- 現代のほとんどの言語（の処理系）はC言語を使って作られています

- インタープリター言語でも似たような動作をしてるんですか？してるんですか？

- goto文というものを今回の授業で初めて知りましたが、調べてみると禁じられた条件文と書かれていて興味深かったです。

もくじ

1. 前回の振り返り
2. 2進数や16進数による数値表現
3. 実際の命令セットの例

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

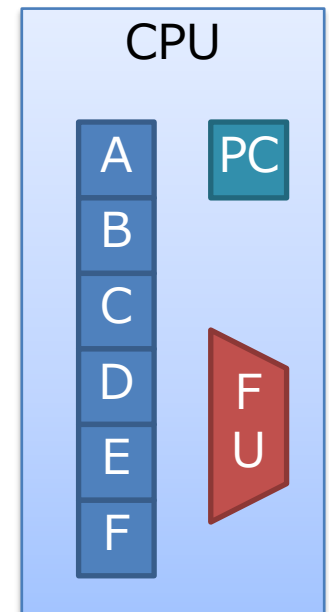
CPU

■ 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

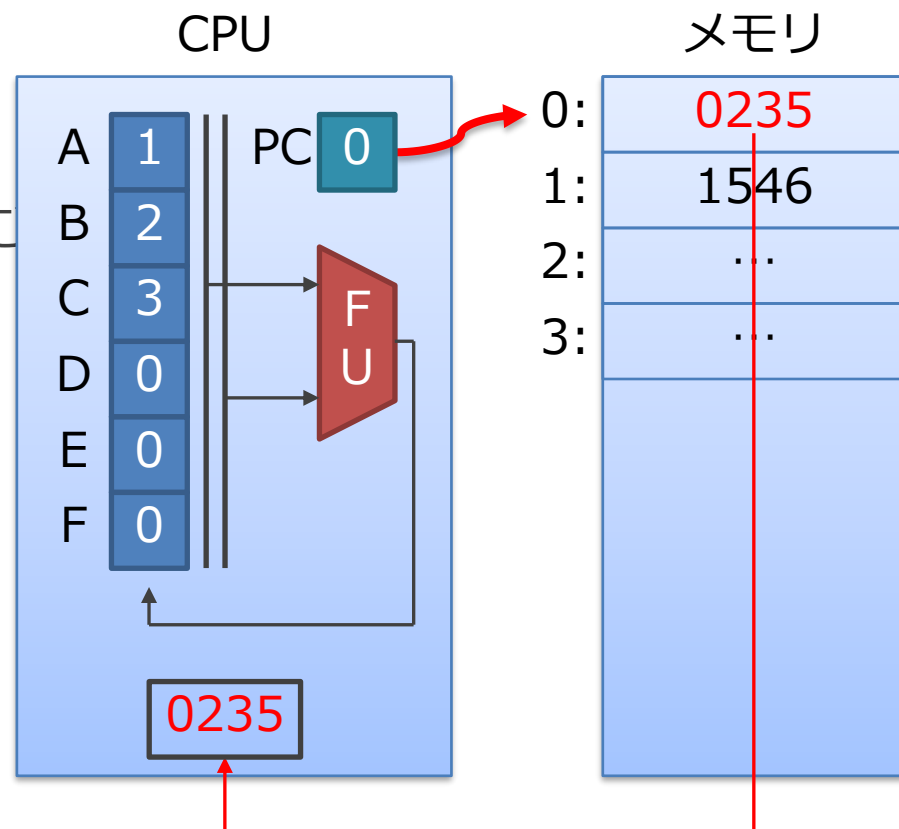


CPU の動作

- 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



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

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1 or 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. 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進数の相互変換

- 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 3 2 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

■ 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なので誤差が出る
- 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

実際の命令セットの例

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

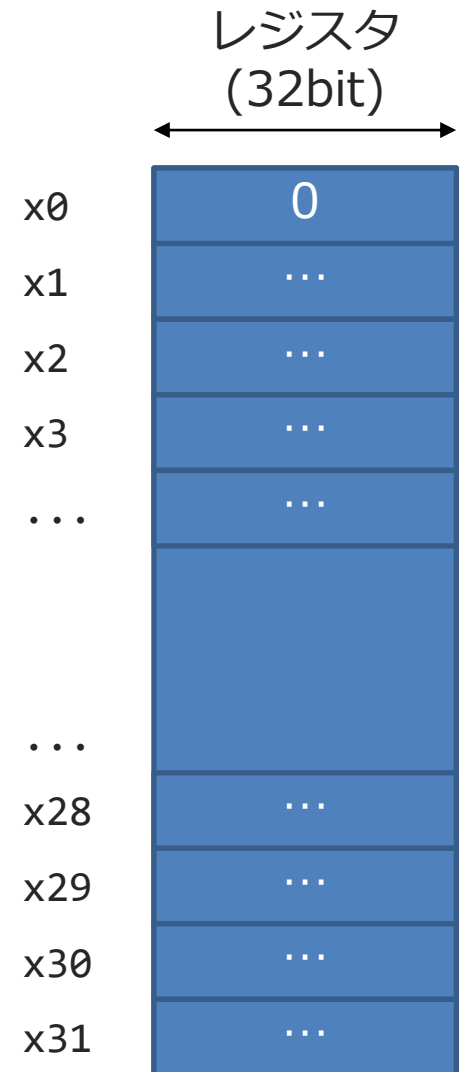
RISC-V 命令セットの基本

■ 基本的な特徴：

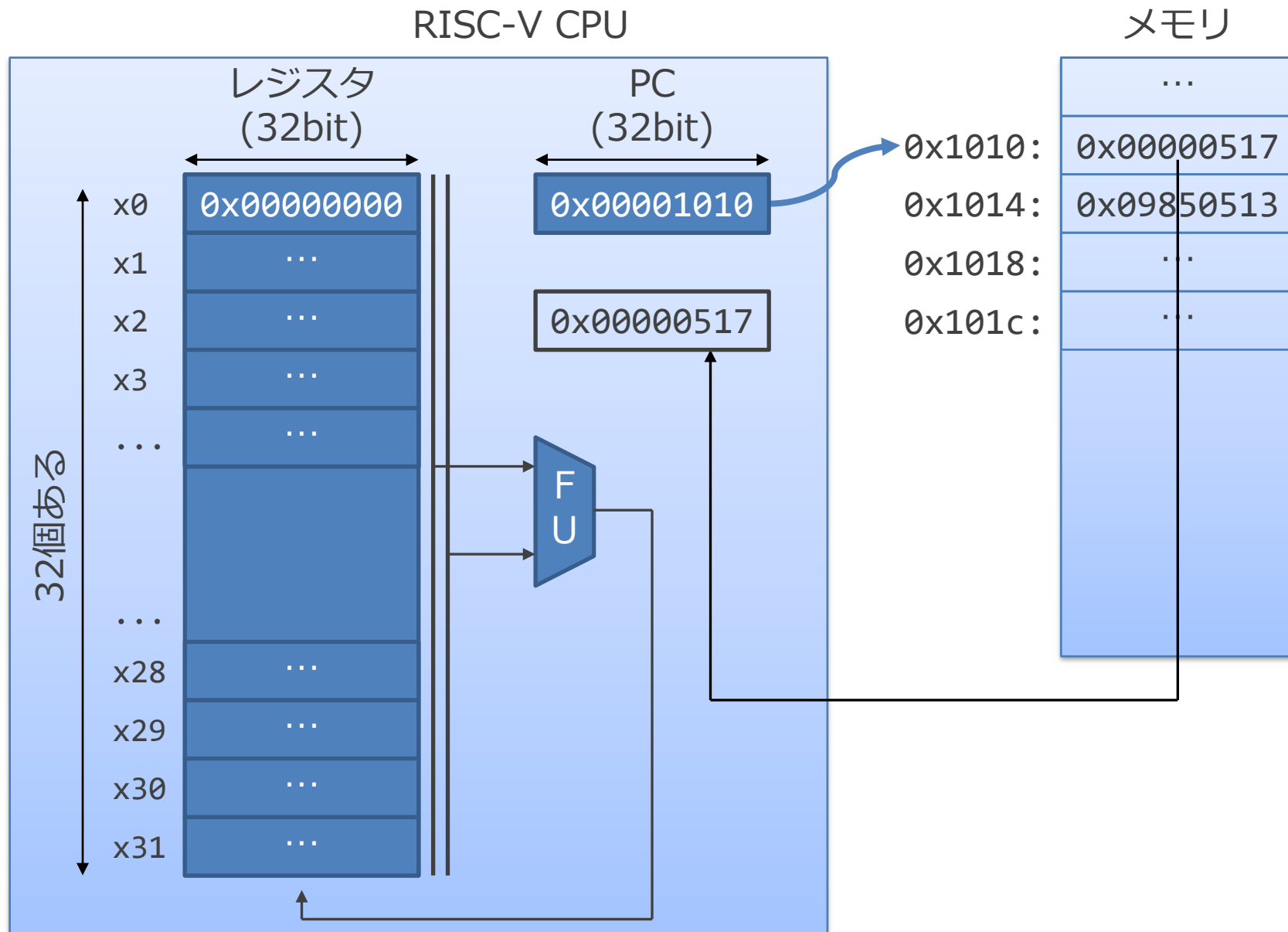
- 各命令は 4 バイトのデータで構成
- レジスタは 32本 ある
 - A,B,C ... ではなく, x0, x1, x3... と表記
 - うち 1 つはゼロレジスタ
(常にゼロが入っている)
- 各レジスタの幅は 32/64/128 bit が規定されている
 - ここでは 32bit のものを取りあげる

■ 基本的には、これまでに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

- これまでは、なんとなく適当に10進数で話していた



RISC-V (32bit) のイメージ



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
- 32bit 中をどう区切って解釈するかが4タイプあるということ

■ opcode 部分によって, 32bit 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 と思えばよい

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

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

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



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



■ ADD や SUB は R-Type となる

- まず opcode = 0110011 は R-Type の整数演算を表す
- 次に funct7 の部分で, さらに ADD や SUB を判別

■ 大分類が整数演算, 小分類が ADD や SUB になるような感じ

I-Type の演算命令



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



- opcode が 0010011 は ADDI
 - レジスタを読んだ値ではなく, immediate の部分をそのまま加算する
 - 大分類を見ただけで ADDI であることが確定する

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

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, メモリ読み出しの前に加算処理が入っている
 - 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 : $(x[rs1] + \text{immediate}) \leftarrow x[rs2]$



■ SW : Store Word 命令

- opcode: 0100011 はロード命令で S-Type
- funct3 部分がかわると、格納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

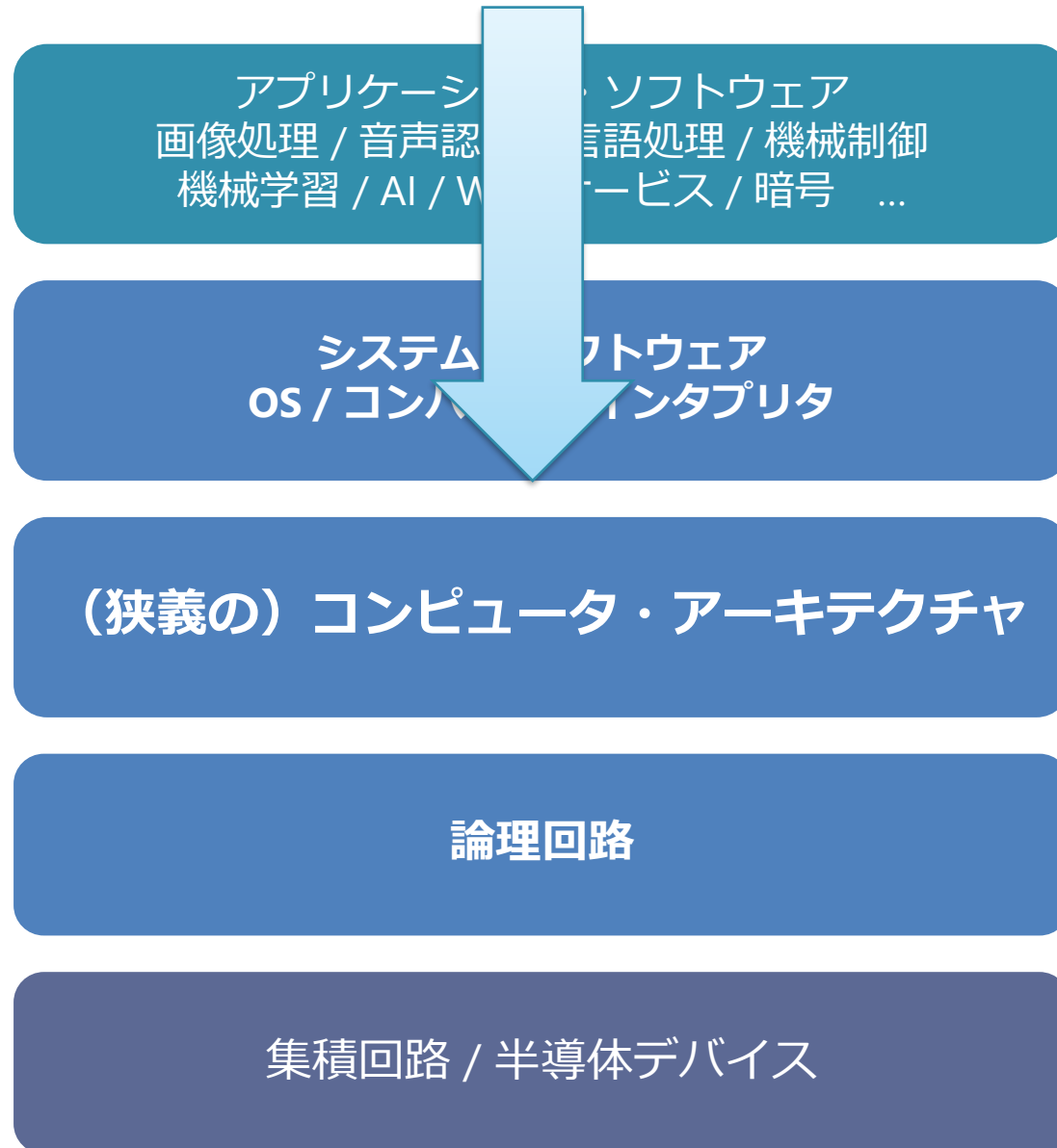
- 残りの命令は, 大体これのバリエーション

論理回路と半導体デバイスによる実装

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
 - 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

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

集積回路 / 半導体デバイス

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

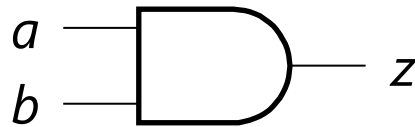
2. 順序回路

- 出力が、過去の入力（の履歴）にも依存する論理回路

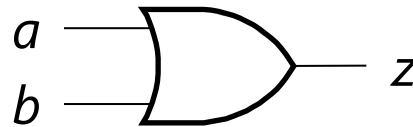
組み合わせ回路の例：2入力論理ゲート

AND
(論理積)

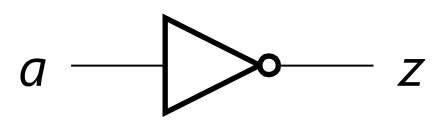
MIL記号
MIL symbol



OR
(論理和)



NOT
(論理否定)



論理式
logic expression

$$z = a \cdot b$$

$$z = a + b$$

$$z = a'$$

$$z = \bar{a}$$

$$z = \neg a$$

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

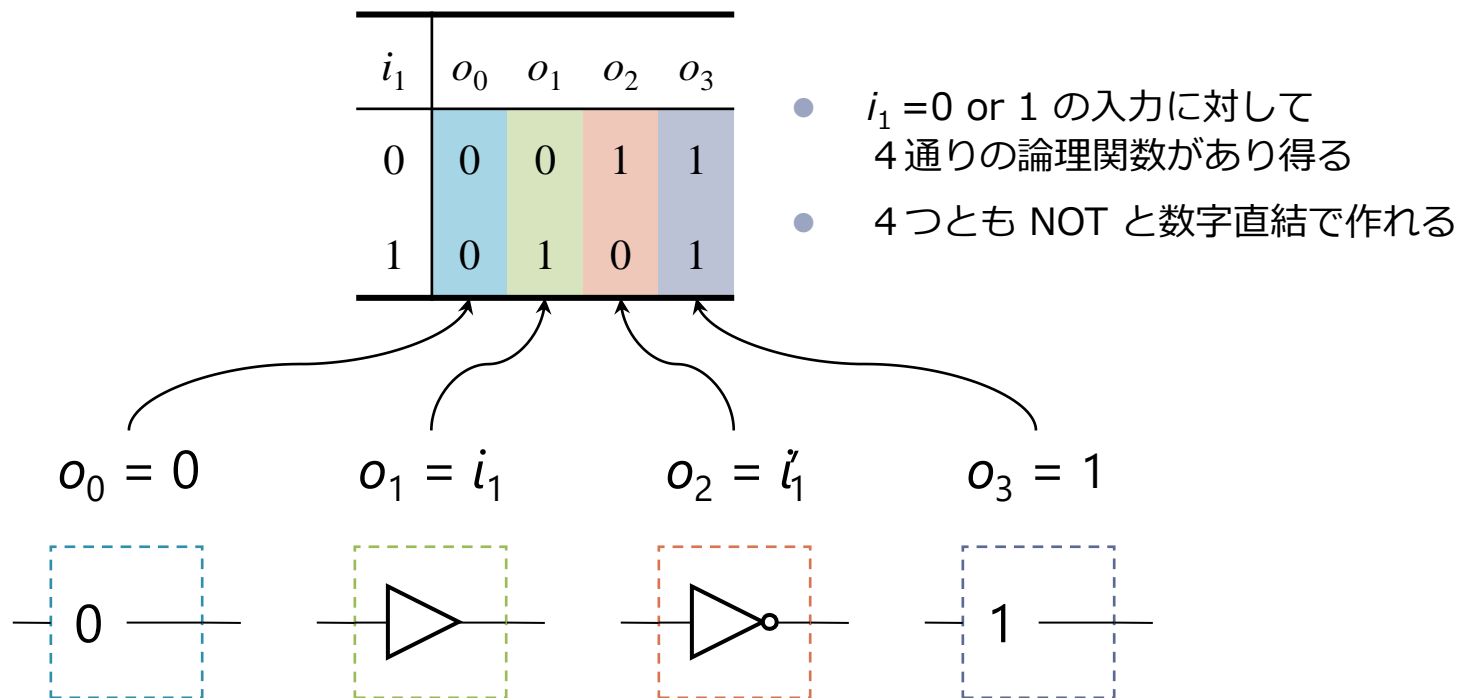
完全性 (Completeness, 完備性)

- 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できる

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

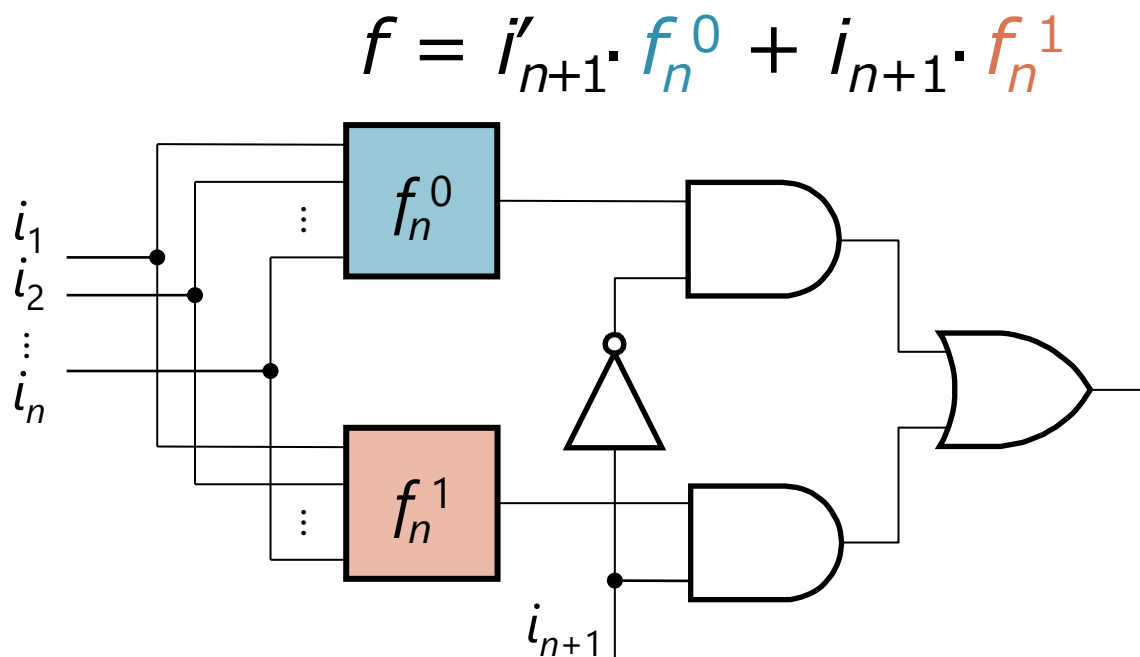


完全性の証明 {AND, OR, NOT}

■ 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

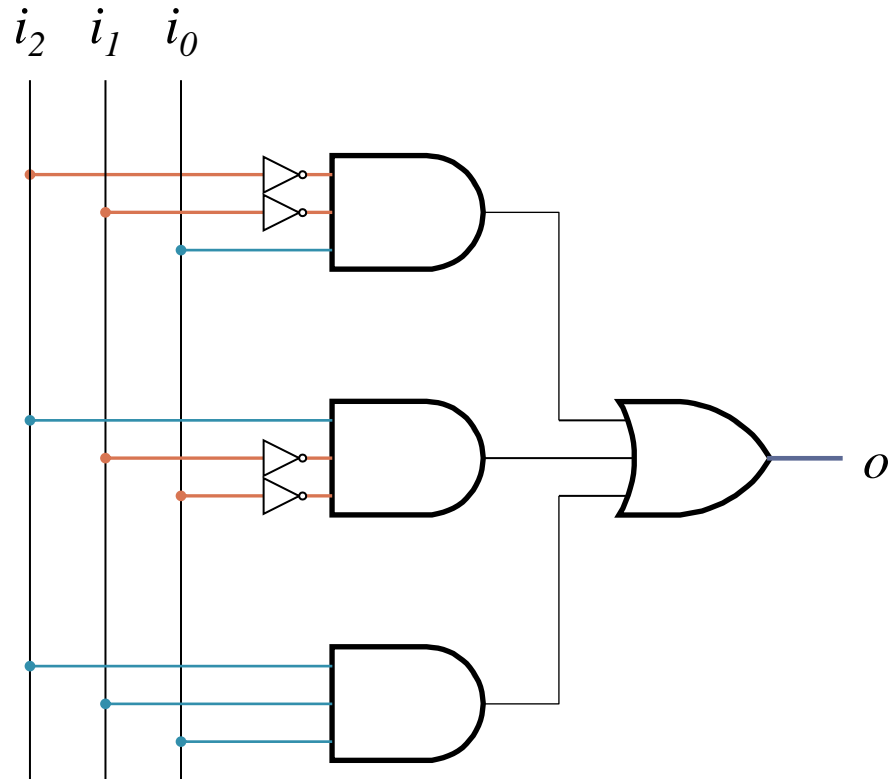
i_{n+1}	i_n	...	i_2	i_1	o
0	0	...	0	0	
	0	...	0	1	
	\vdots	\vdots	\vdots	\vdots	
	1	...	1	1	
1	0	...	0	0	
	0	...	0	1	
	\vdots	\vdots	\vdots	\vdots	
	1	...	1	1	



- 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



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

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

0000000	rs2	rs1	<u>100</u>	rd	0110011
---------	-----	-----	------------	----	---------

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

0000000	rs2	rs1	<u>110</u>	rd	0110011
---------	-----	-----	------------	----	---------

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

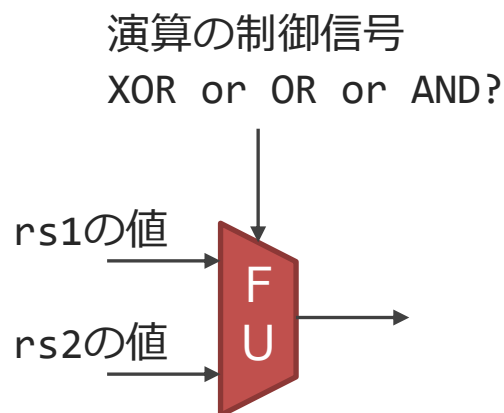
0000000	rs2	rs1	<u>111</u>	rd	0110011
---------	-----	-----	------------	----	---------

■ 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

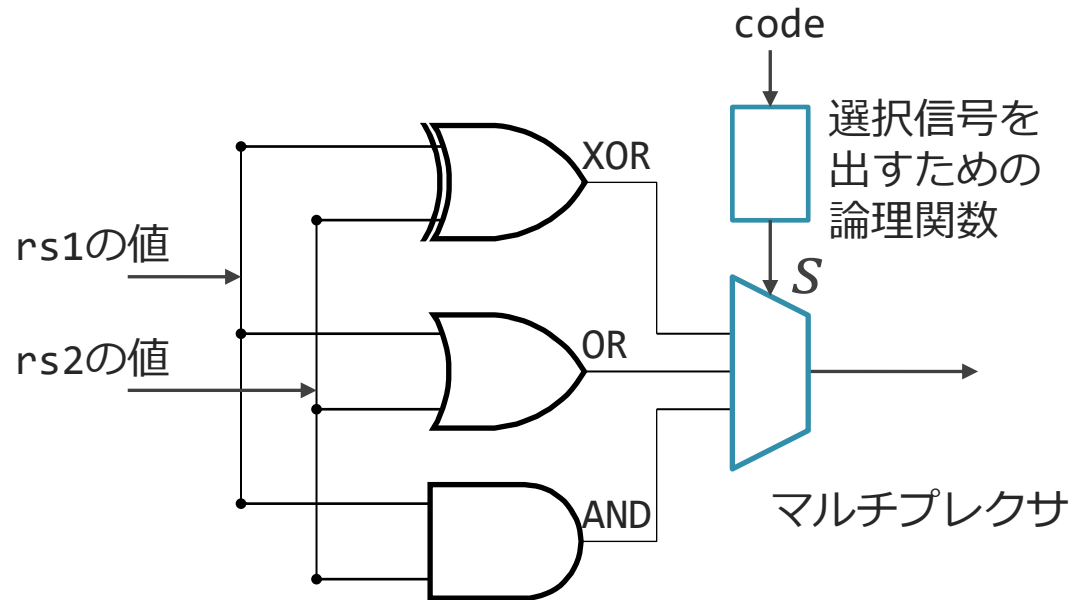
```
if code == 0b100:  
    rs1 xor rs2  
elif code == 0b110:  
    rs1 or rs2  
else:  
    rs1 and rs2
```



- 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
```

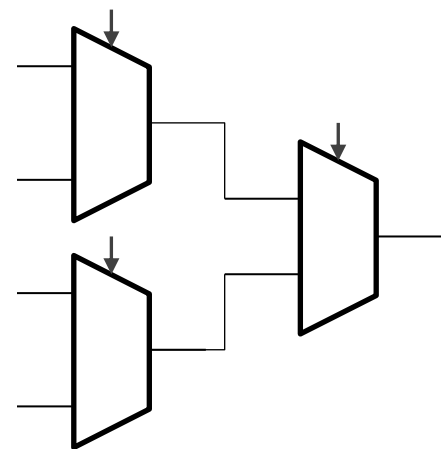
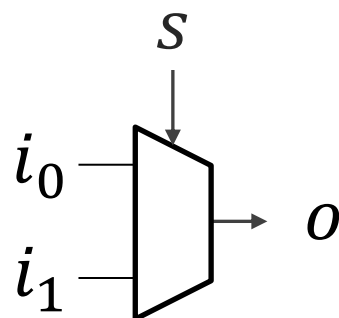


■ 典型的には,

- 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

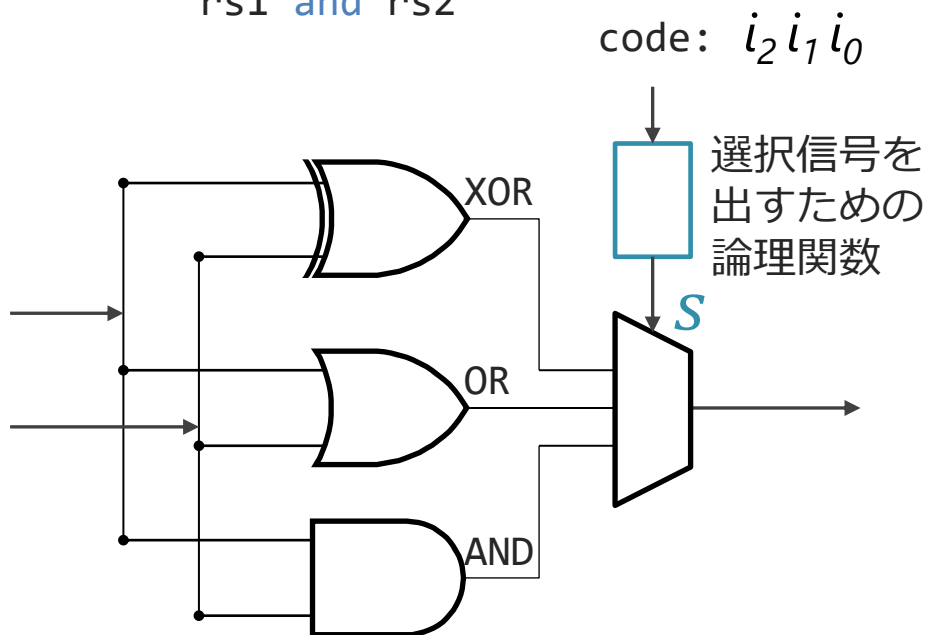
s	i_0	i_1	o
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



- 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
```

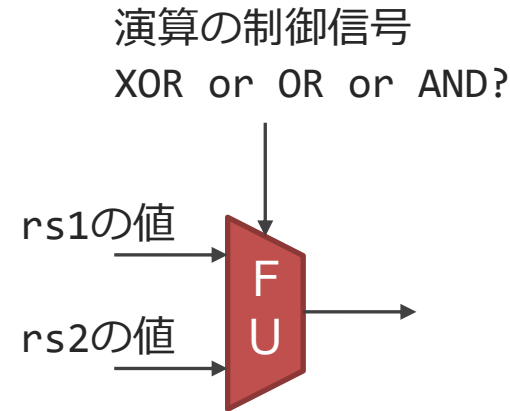


	i_2	i_1	i_0	S_1	S_0
	0	0	0	x	x
	0	0	1	x	x
	0	1	0	x	x
	0	1	1	x	x
XOR	1	0	0	0	0
	1	0	1	x	x
OR	1	1	0	0	1
AND	1	1	1	1	0

- 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

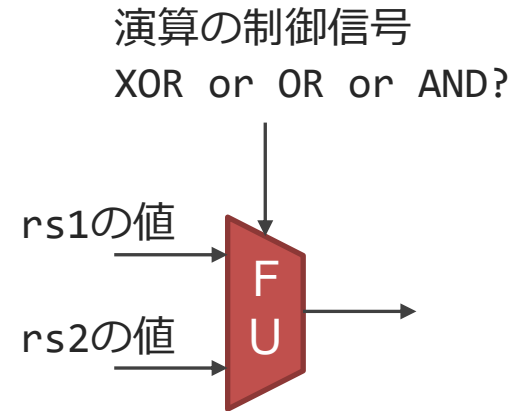
```
if code == 0b100:  
    rs1 xor rs2  
elif code == 0b110:  
    rs1 or rs2  
else:  
    rs1 and rs2
```



- 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

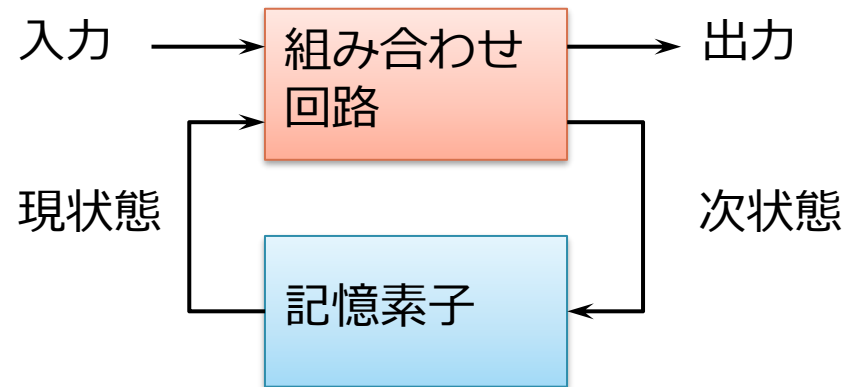
1. 組み合わせ回路

- 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

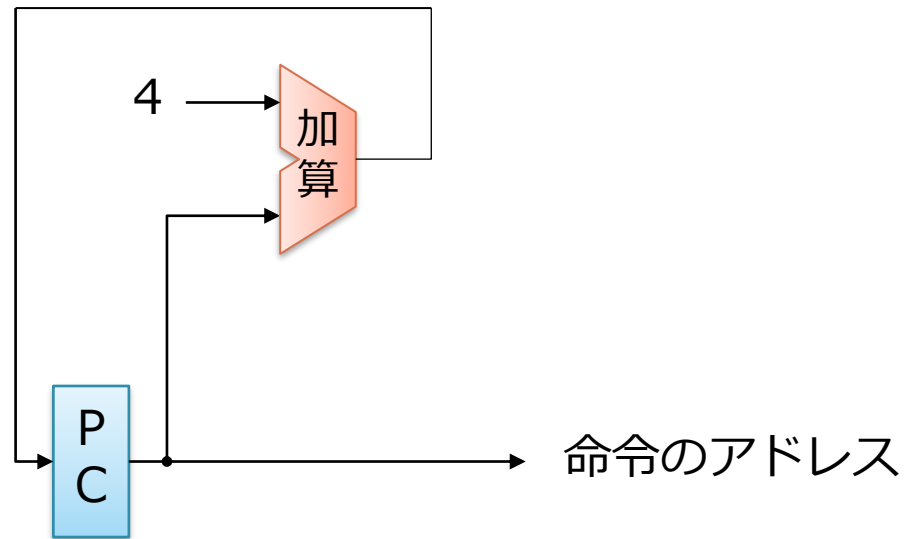
- 出力が、過去の入力（の履歴）にも依存する論理回路

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

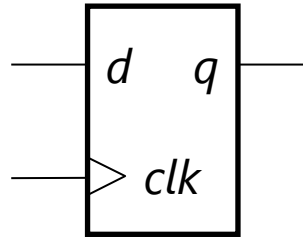


- 毎サイクル PC が加算される部分は，典型的な順序回路

記憶素子の例：D-FF (Flip Flop)

■ 入出力端子

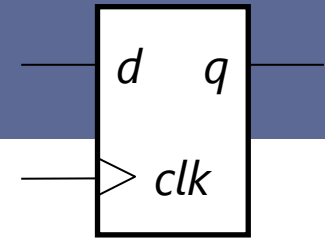
- ◇ データ入力 : d
- ◇ データ出力 : q
- ◇ クロック入力 : clk



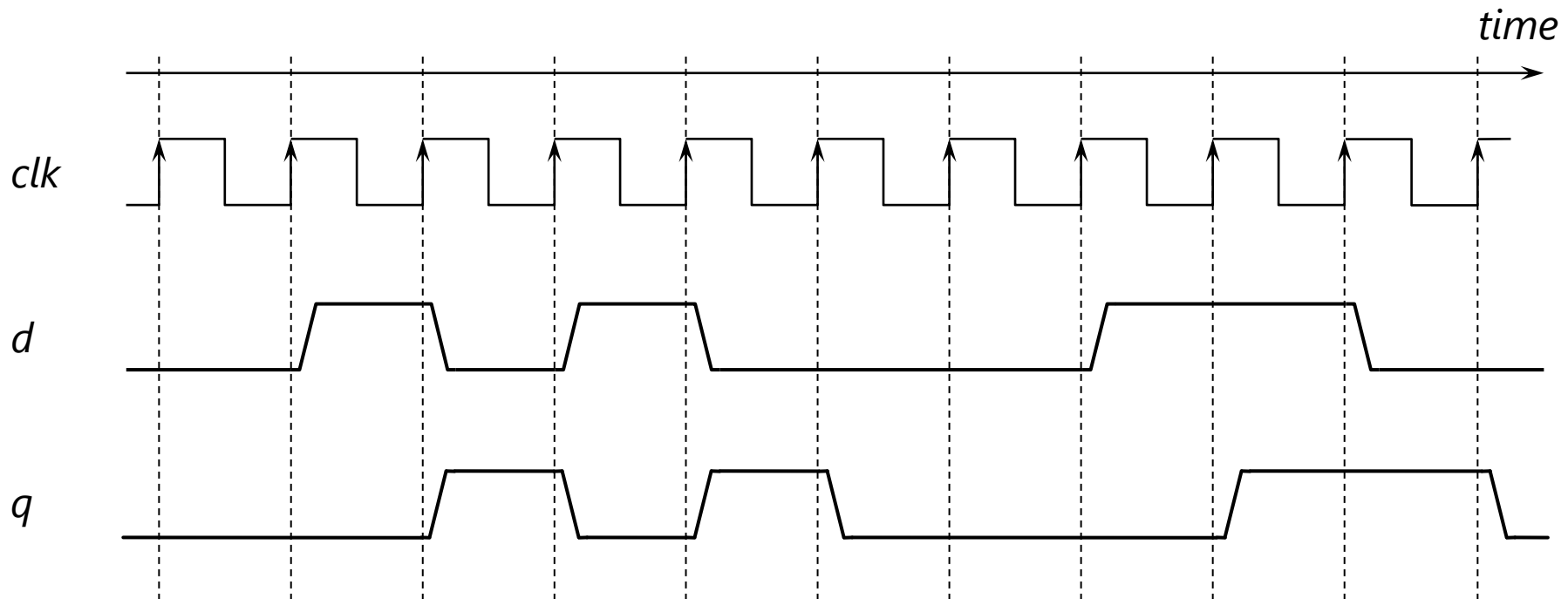
■ 働き：

- ◇ 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



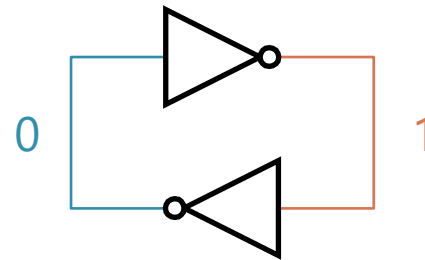
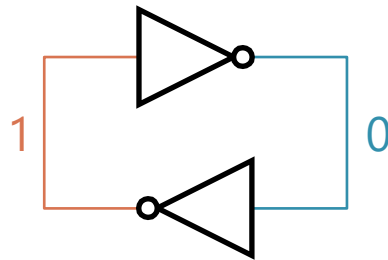
d	0	1	0	1	0	0	0	1	1	0
q	0	0	1	0	1	0	0	0	1	1



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

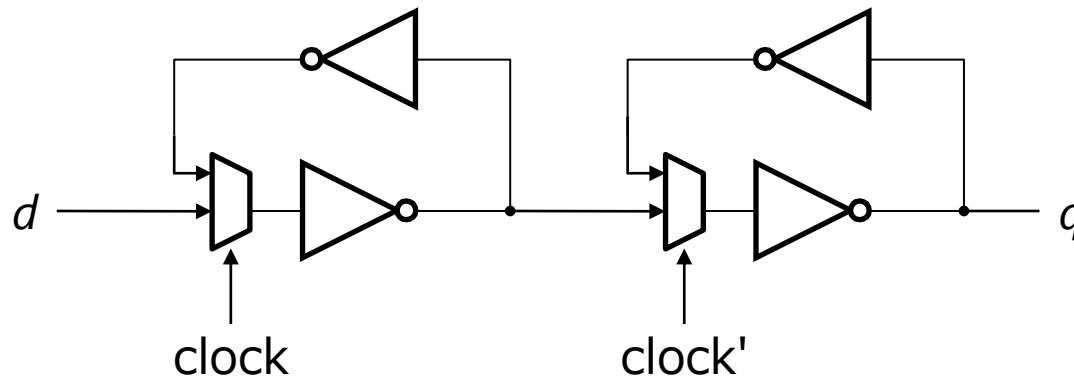


D-FF の実装

■ 構造：

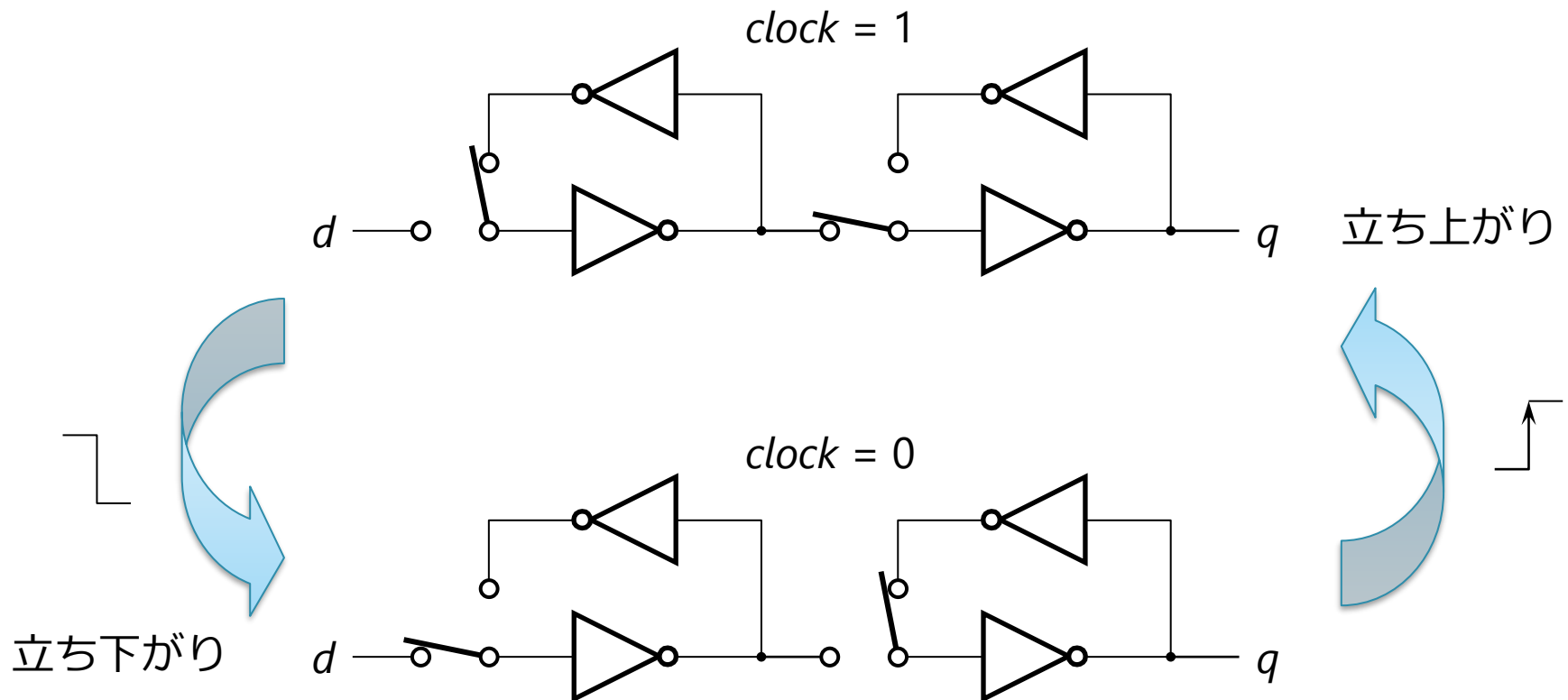
- ◇ NOT ゲートのループを二段に接続
- ◇ 各ループにはマルチプレクサが入っている

■ この構造は、クロックのエッジで記憶を更新するため

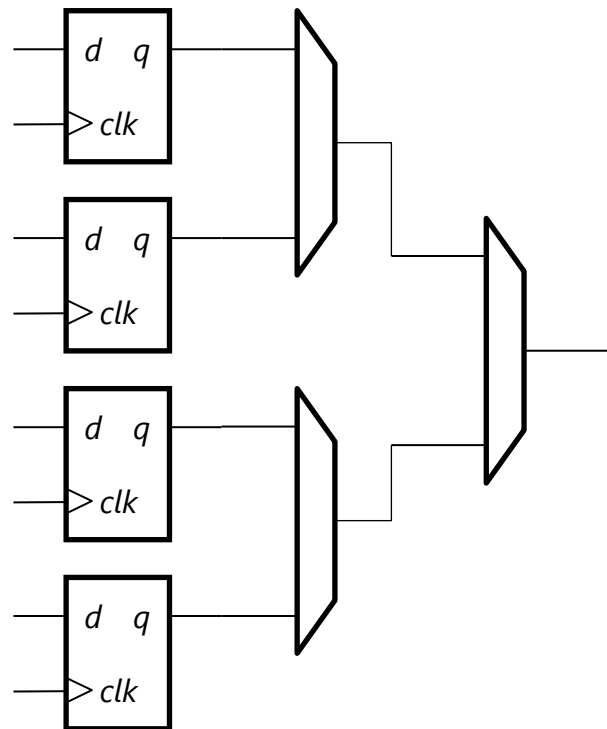


D-FF の実現例

- 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

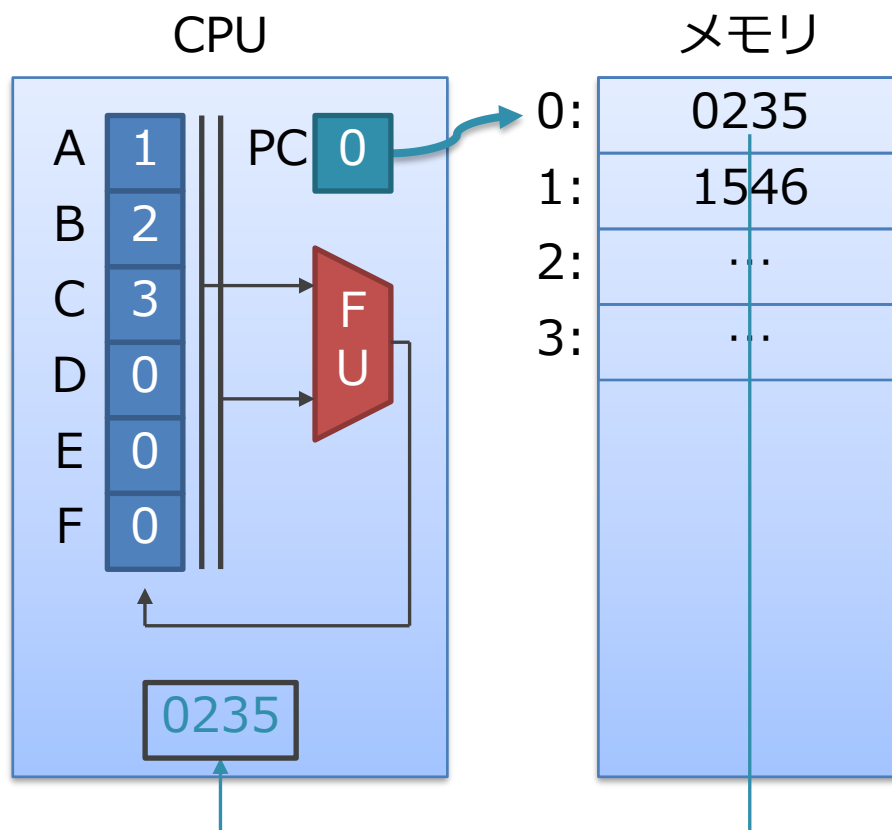


- D-FF を必要なだけ並べて，マルチプレクサで選択することで実現できる
 - 実際には，もっと最適化された回路（SRAM）が使用される事が多い
 - （後の講義で詳しく説明する予定

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. 実際の命令セットの例
 - RISC-V
 3. 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「デジタル回路」の講義資料の図を使用しています

- 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
- 注意：
 - 初回で話したように、これは必須です（成績に影響します）