第6回
6/4/2011
状態遷移回路とシングルサイクルCPU設計

プロセッサの構成:
データパス(databath)と制御(control)

- ALUは組み合わせ回路→内部に状態を持たず、出力は入力に関数的に依存する
- しかし、これだけでは電卓で、計算機は作れない
- 実際には、レジスタやメモリからデータを読み、ALUを通して演算した後、再びレジスタやメモリに書き戻さなくてはならない➔状態を持つ回路?

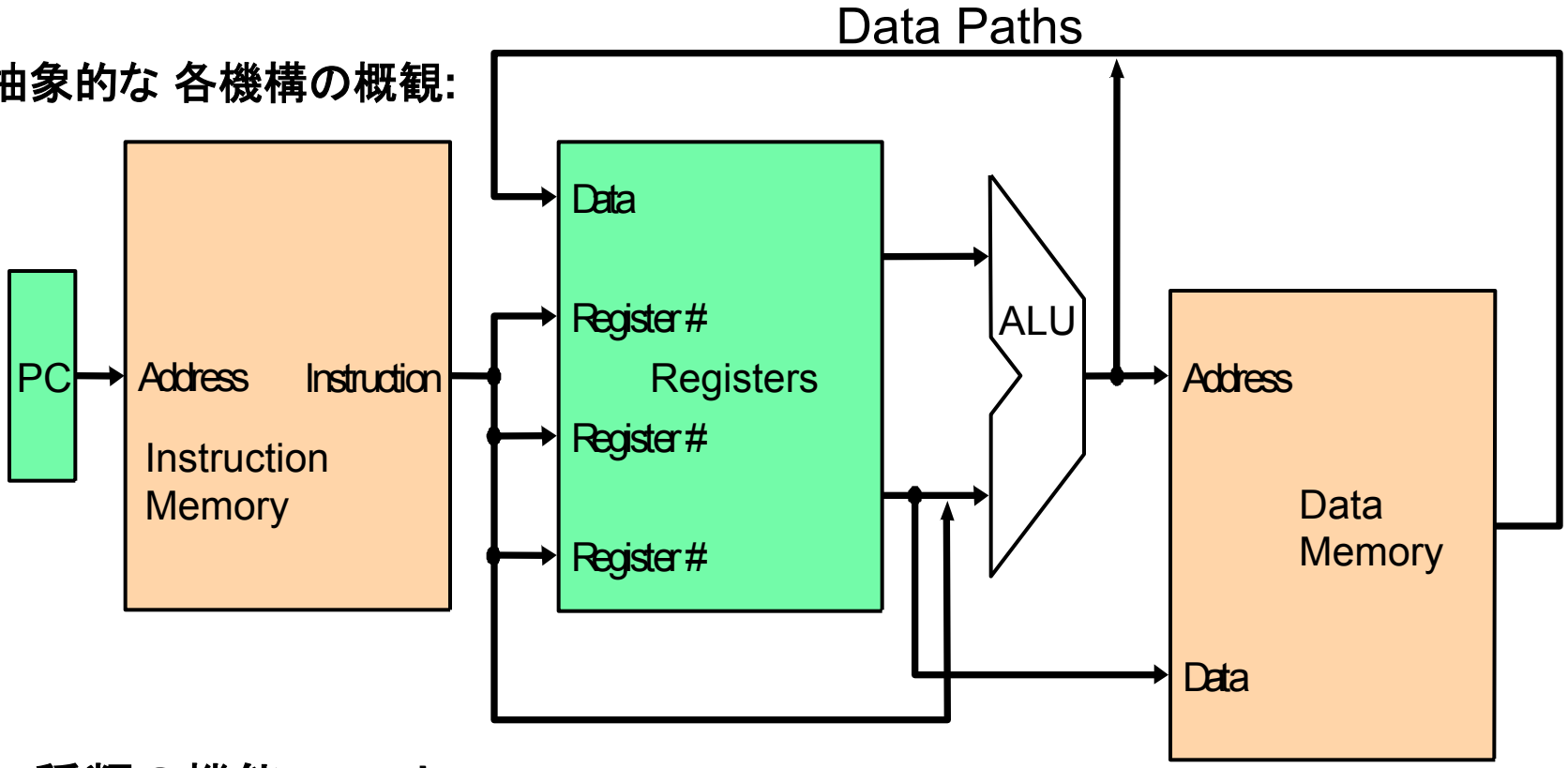
- データパス(datapath): プロセッサ内のデータの流れ
- 制御(control): データパスにどの機能ユニットのデータが、どのタイミングで流れるかを制御する
- プロセッサの構成
  - 機能ユニット(ALU, レジスタ) + データパス + 制御(回路)
  - 機能ユニットは、状態を持つものと、持たないものに分かれる

MIPSプロセッサの設計

- MIPS全体の構成を考え、データパスと制御を考えよう
- ただし、機能を単純化する:
  - メモリ参照命令: `lw, sw`
  - 算術論理命令: `add, sub, and, or, slt`
  - 制御命令: `beq, j`
- Von Neumann型計算機の汎用的な実装は、以下のようになる
  (命令サイクル):
  - プログラムカウンタ(PC)が命令のアドレスを指示する
  - 命令をメモリから読み込む(命令フェッチ)
  - 指定されたレジスタの値を読み出す
  - 命令に従って、どのような操作をするかを決定する
- 全ての命令は、レジスタを読んだ後、ALUを用いる
  これはなぜか?メモリ参照命令、算術論理命令、制御命令
- 状態は、レジスタおよびメモリに保存される => 状態を保持する回路が必要

MIPSプロセッサ実装の概観

- 抽象的な 各機構の概観:



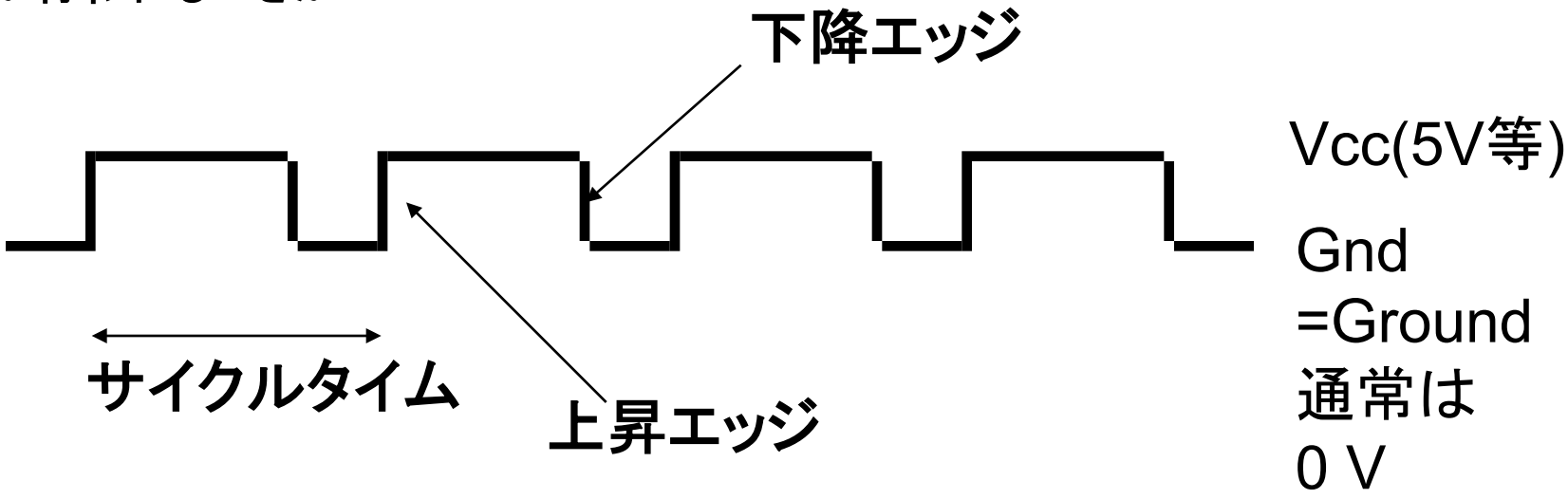
二種類の機能ユニット:

- データの値の入力に対し、組み合わせ的に出力を行う(組み合わせ回路)
- 状態を含む回路 (状態遷移回路)

それらを結ぶデータパス

状態(遷移)回路について

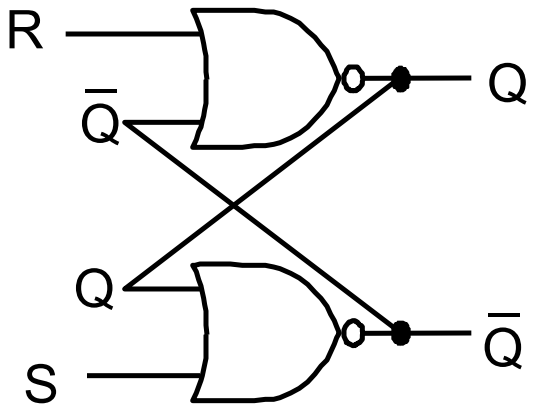
- 同期を行うクロックなし (非同期回路, asynchronous circuits)
  vs. クロックあり (同期回路, synchronous circuits)
- 同期回路は、「クロックに従って」状態遷移を行う
  - 状態を含む回路の要素は、クロックのどのような状態をトリガとして更新が行われるべきか?



同期回路の状態遷移は、クロック電圧の上昇(GndからVccへ)・下降(VccからGndへ)・又は両方の遷移にておこる(トリガされる)

非同期な状態回路の要素: SRフリップフロップ

- 最も基本的な状態回路: SRフリップフロップ (SR-Flip-Flop)
  - 出力は、R, Sの入力のみならず、過去の入力に依存する。
  - 入力に対する関数的動作はないが、現在の「状態」x入力に対して関数的



R	S	入力 Q	出力 Q
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	?
1	1	1	?

ラッチとフリップ-フロップ (Latch and Flip-flop)

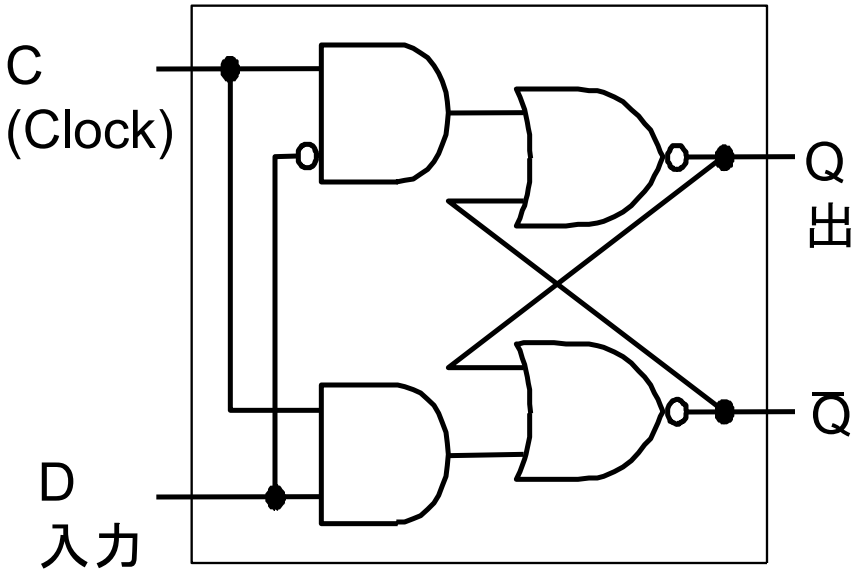
- 両者とも、出力は回路要素内に格納された値に常に等しい
  (つまり、出力値は常に出ており、特に読むのに許可を必要としない)
- 状態(出力値)の変化は、クロックの状態変化による
  - ラッチ: 入力が変化し、クロックの論理値が真のとき、外部入力に対して状態が変化
  - フリップフロップ: クロックのエッジの瞬間の外部入力により状態が変化
    (エッジトリガ方式)

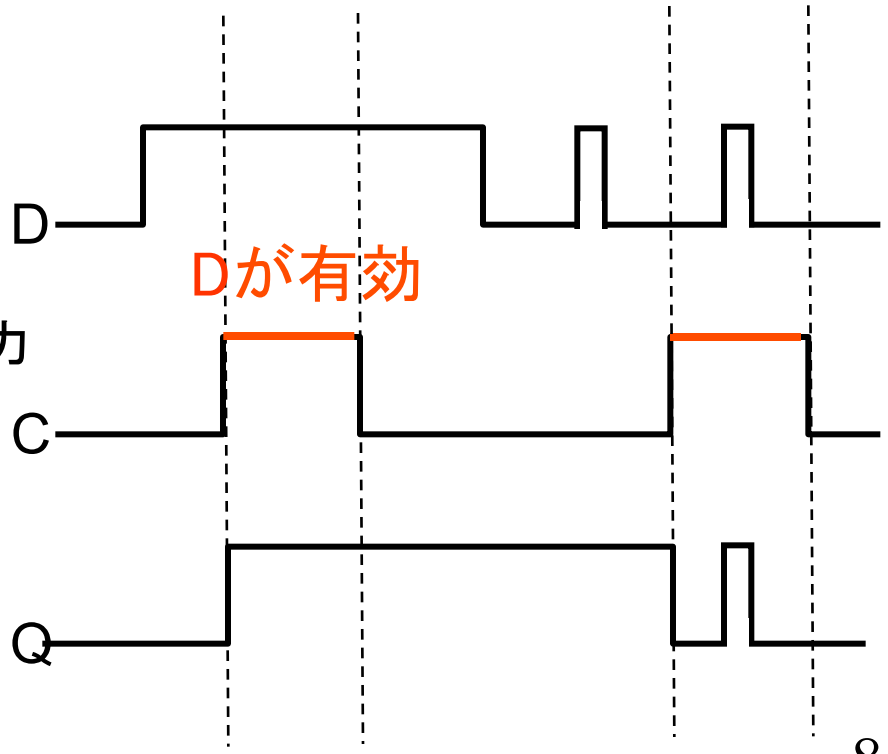
「論理値が真」,
— 電気的には、通常のVcc=5Vの回路では2.5V
2.5V以上が真それ以下でGnd=0Vまでが偽。

クロッキングの手法が、外部入力をいつ反映するのか定義する。
— 読むのと同時に書いたりしてはいけないので

D-ラッチ

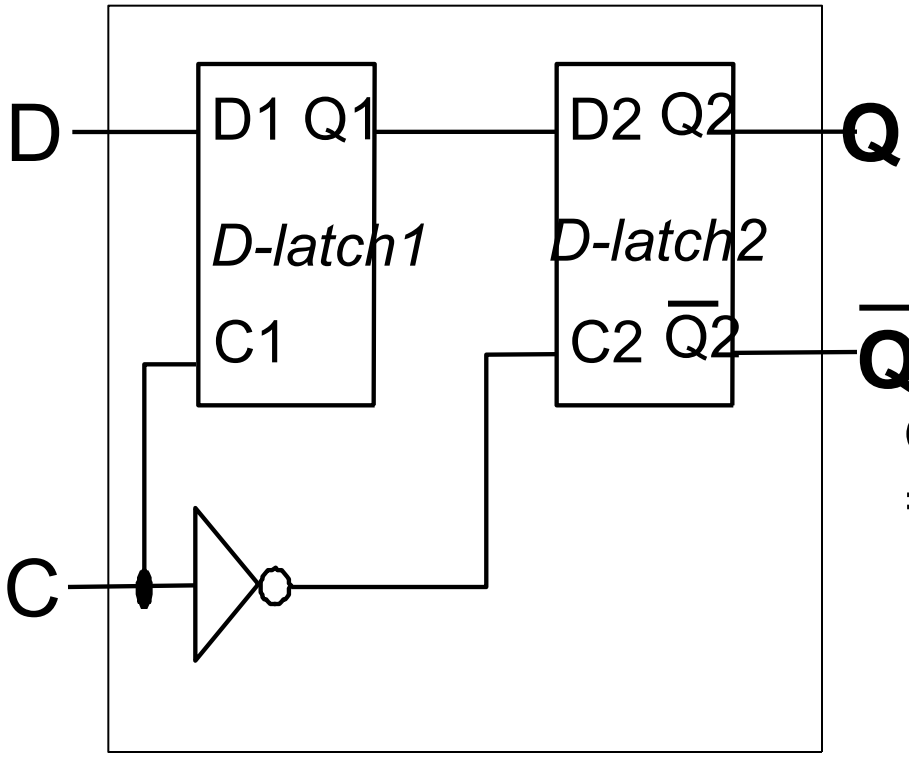
- 二つの入力:
  - 格納されるべきデータの値 (D)
  - いつDを読み書きして良いか示すクロック信号 (C)
- 二つの出力Two outputs:
  - 内部状態の値 (Q) と、その補値 (complement)

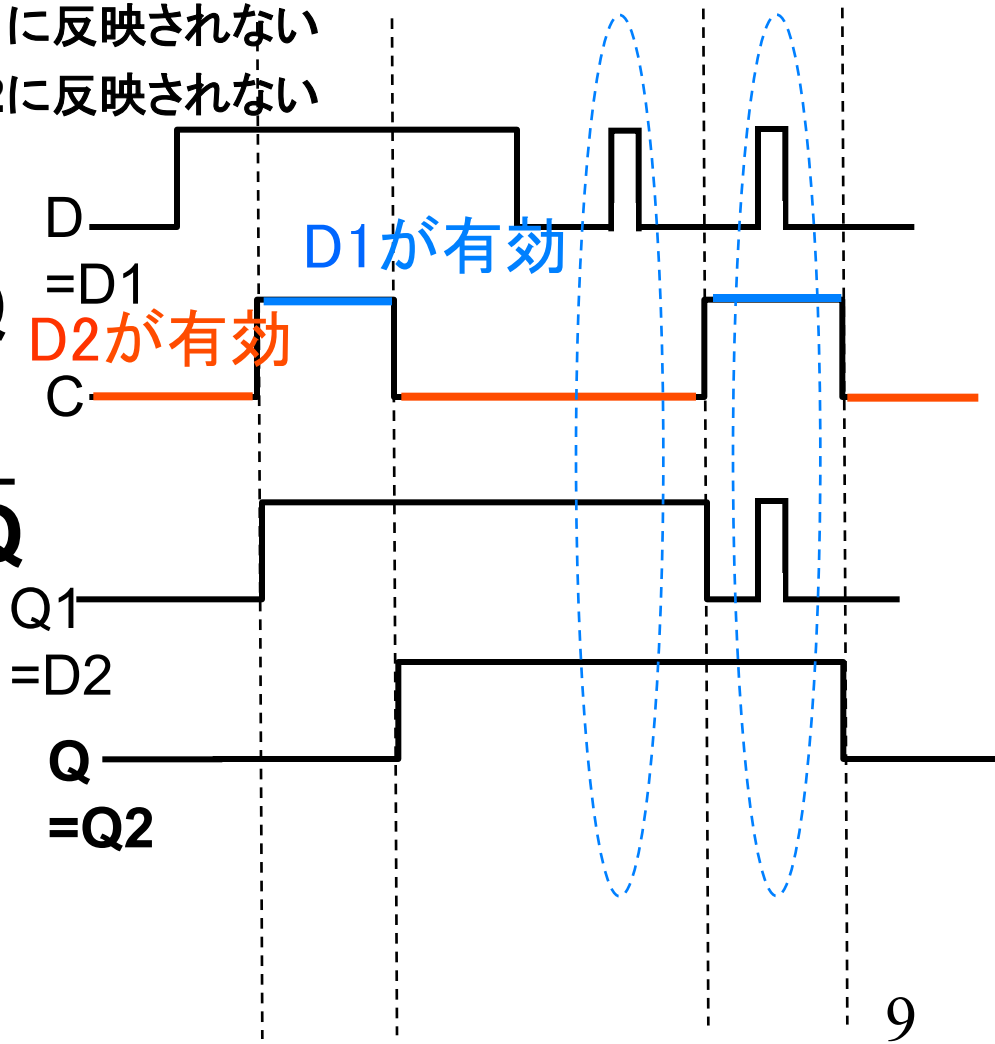




D フリップフロップ

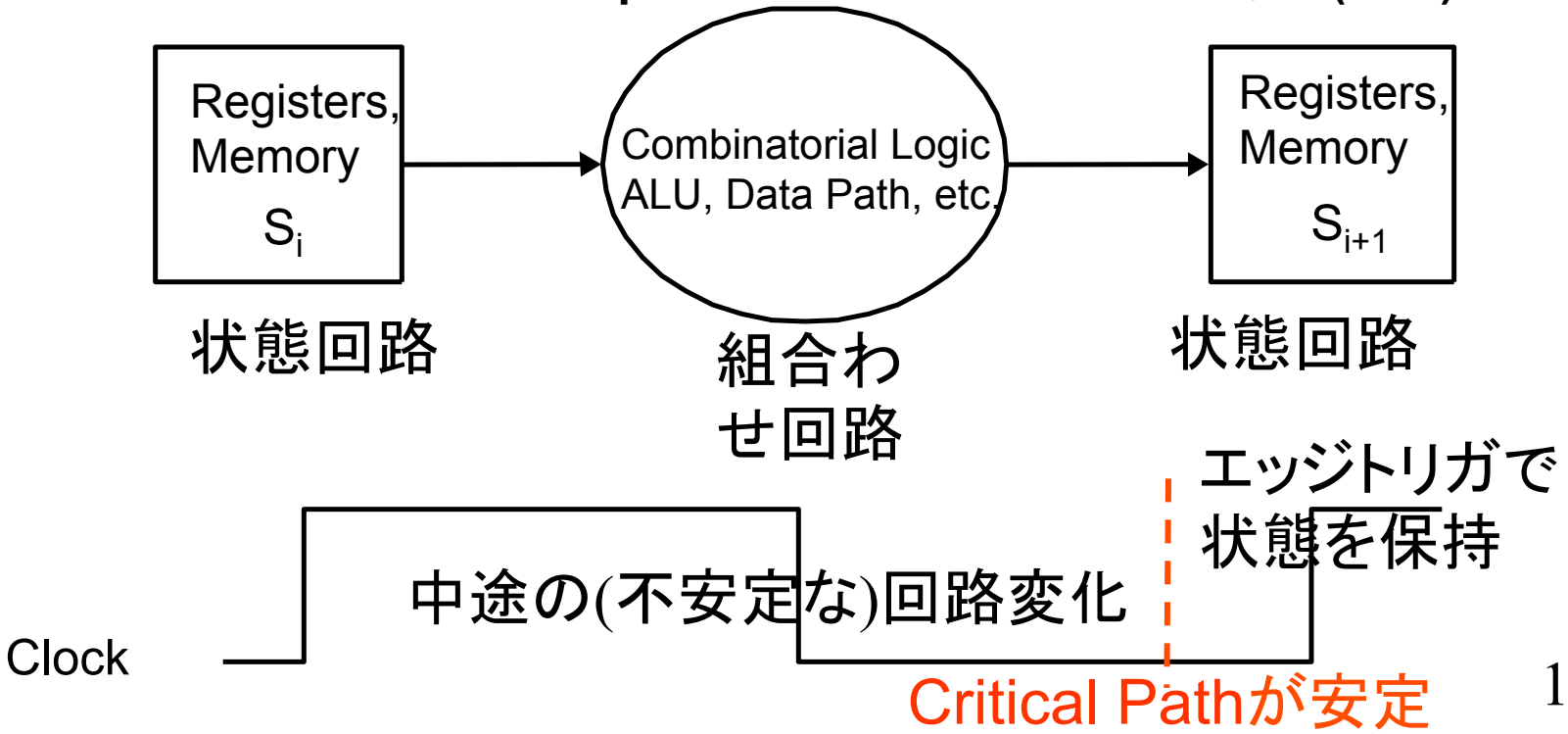
- 入力値が出力に反映されるが、その変化のタイミングはクロックエッジのみ。
- 中途の入力値(D)は、最終的にはQには反映されないことに注意
  - クロックが0の場合→D-latch1のQ1に反映されない
  - クロックが1の場合→D-latch2のQ2に反映されない





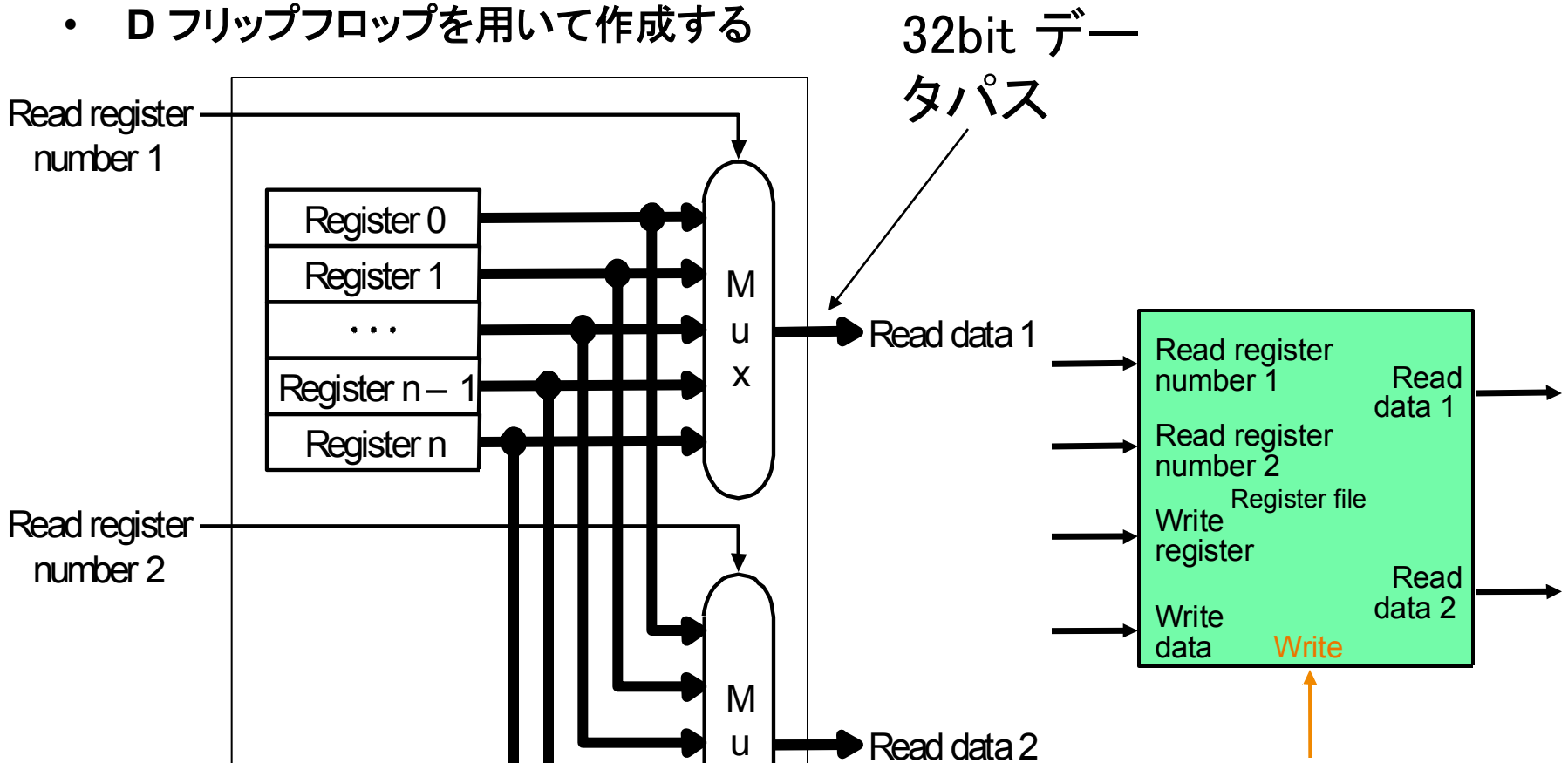
我々のMIPSプロセッサ実装の方式

- 基本的に、エッジトリガ法を用いる
- 典型的なユニット間の実行方式:
  - 状態を表す回路要素の値を読み、
  - 何らかの組合せ回路を通して計算をさせ、
  - 一つ以上の状態を表す(別な)要素回路に結果を書き込む
- 中途の組み合わせ回路のcritical pathが最短のサイクルタイムを規定(後述)



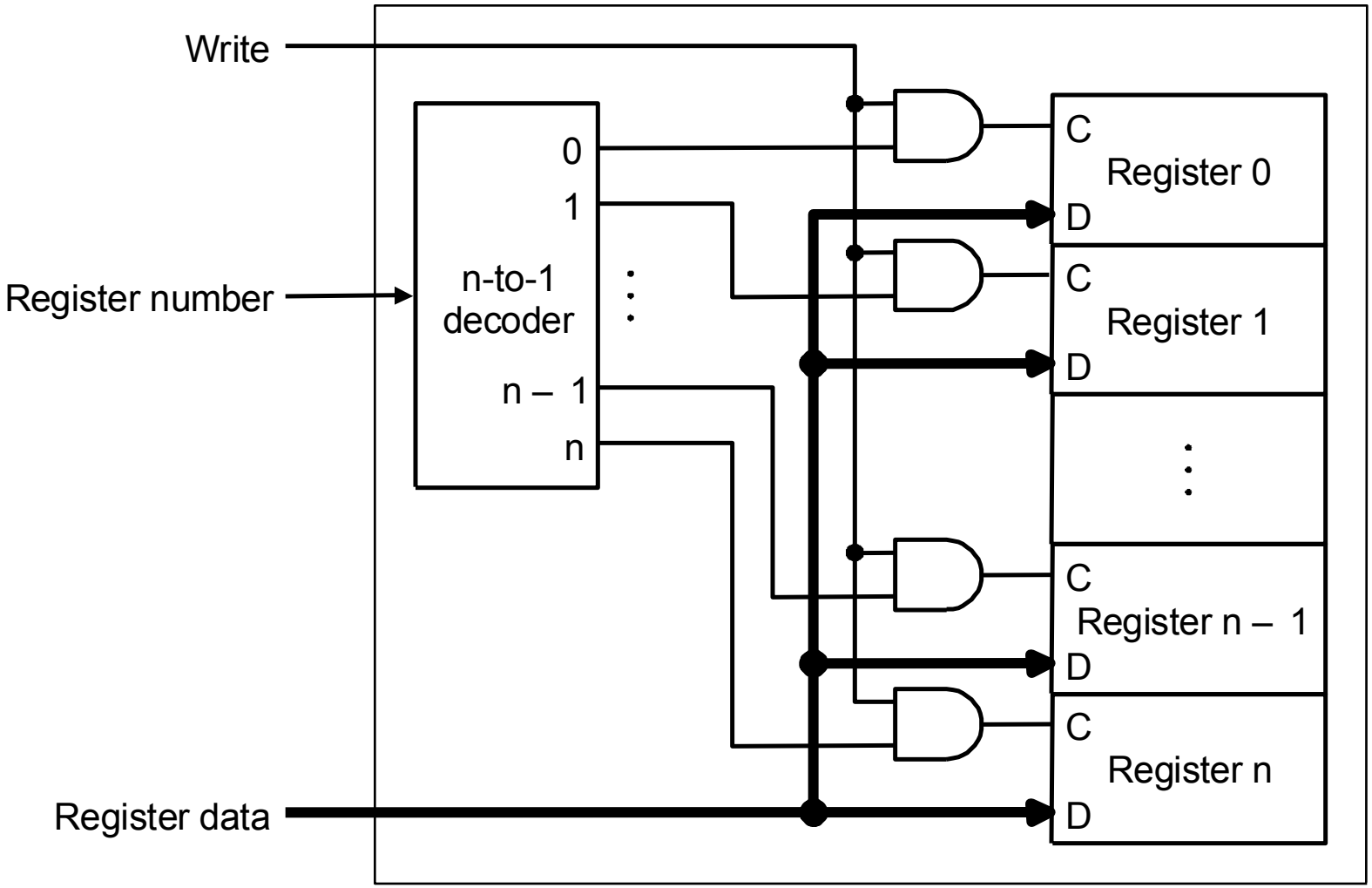
レジスタファイル(読み出し側)

- D フリップフロップを用いて作成する



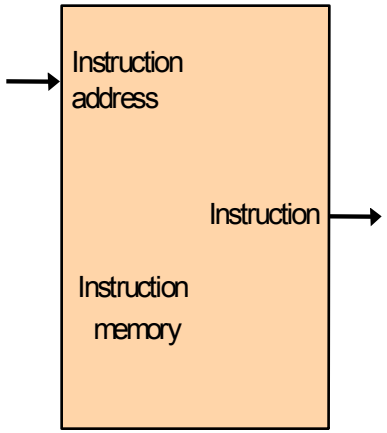
レジスタファイル(書き込み側)

- クロックを用いて、いつ書き込むかを決定していることに注意

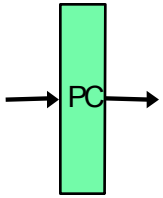


機能ユニットの組み合わせによるMIPSの実装

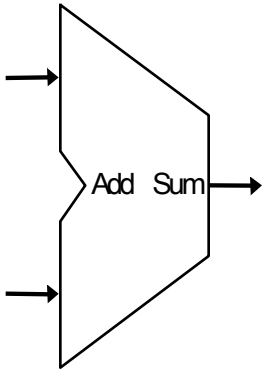
- それぞれの命令の実装に必要な機能ユニットを組み合わせる。



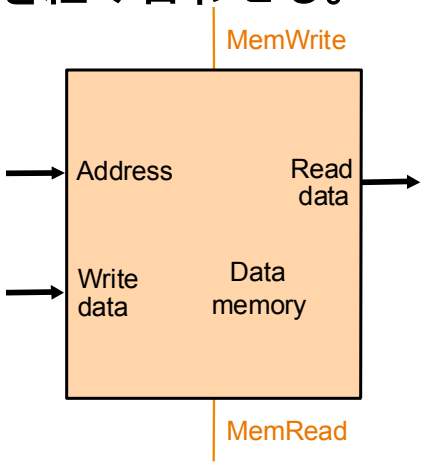
a. Instruction memory



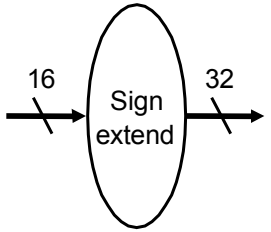
b. Program counter



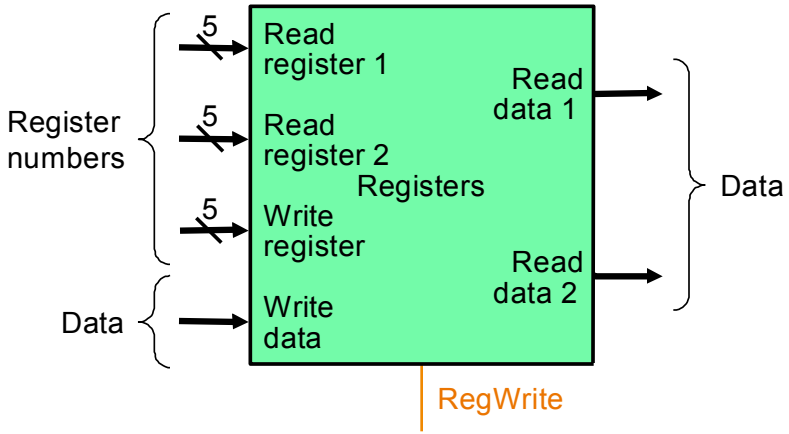
c. Adder



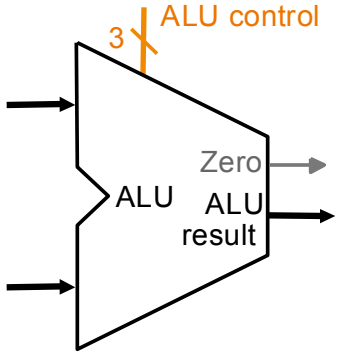
a. Data memory unit



b. Sign-extension unit



a. Registers

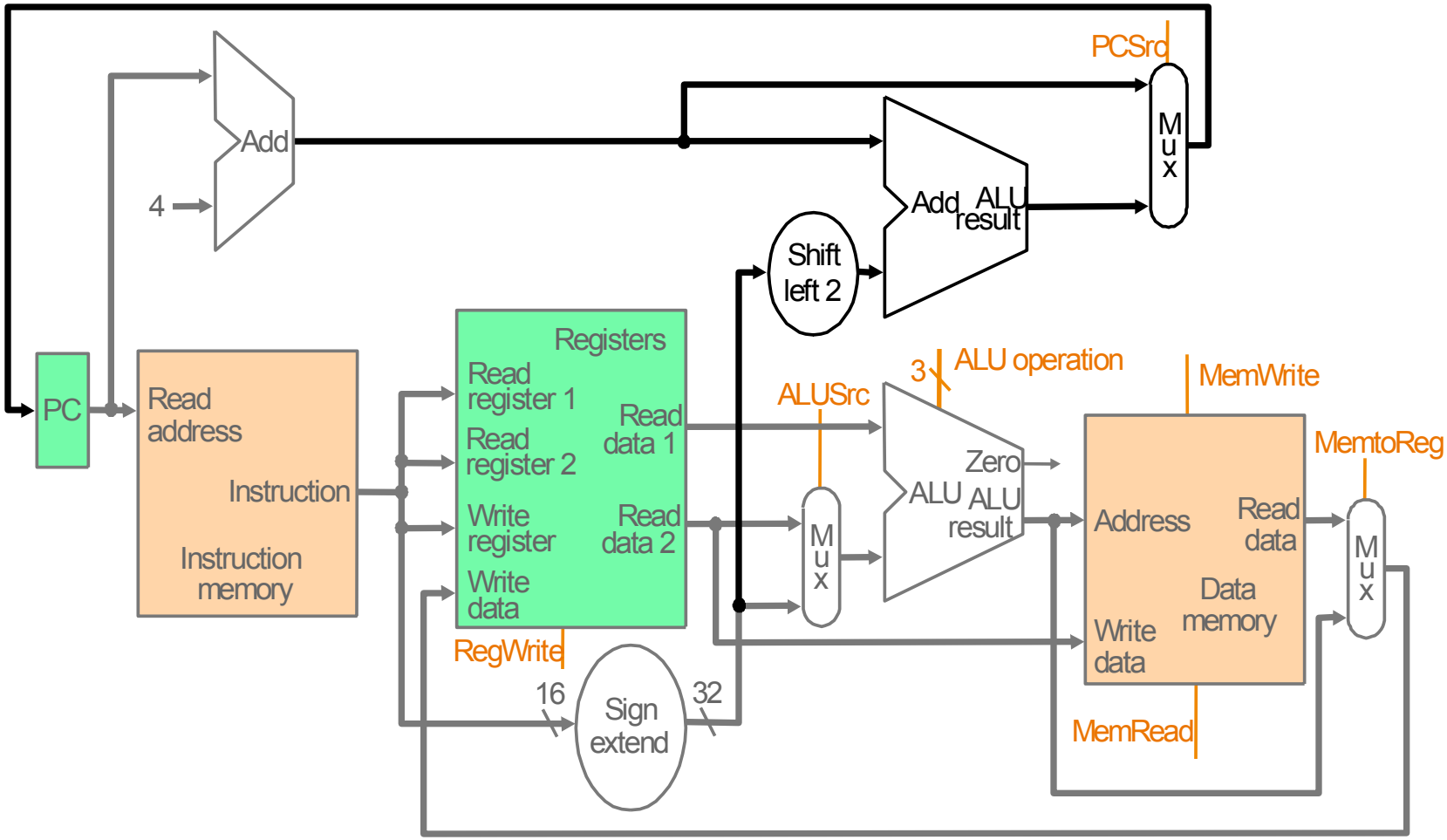


b. ALU

どのように組み合わせて、データパスを構築すれば良いのであろうか?

データパスの構築

- マルチプレクサを用いて、それぞれのユニットを結合する
  - 機能ユニット間のデータの流れ (データパス)
  - しかし、まだ「制御」がない



(機能ユニットの)制御

- それぞれの機能ユニットがどのような操作を行うかを決定 (ALU, read/write, etc.)
- データパス上のデータの流れを制御(マルチプレクサの入力)
- 制御の指定は、もともと命令の32bitが行う
- 例:

  add $8, $17, $18　　R形式の命令フォーマット:

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- ALU の操作は命令タイプ (op)と、機能コード (funct)によって決定される

制御(続き)

- ALUは以下の命令はどのように処理すべきか?
- 例: lw $1, 100($2)

35	2	1	100

op	rs	rt	16 bit offset

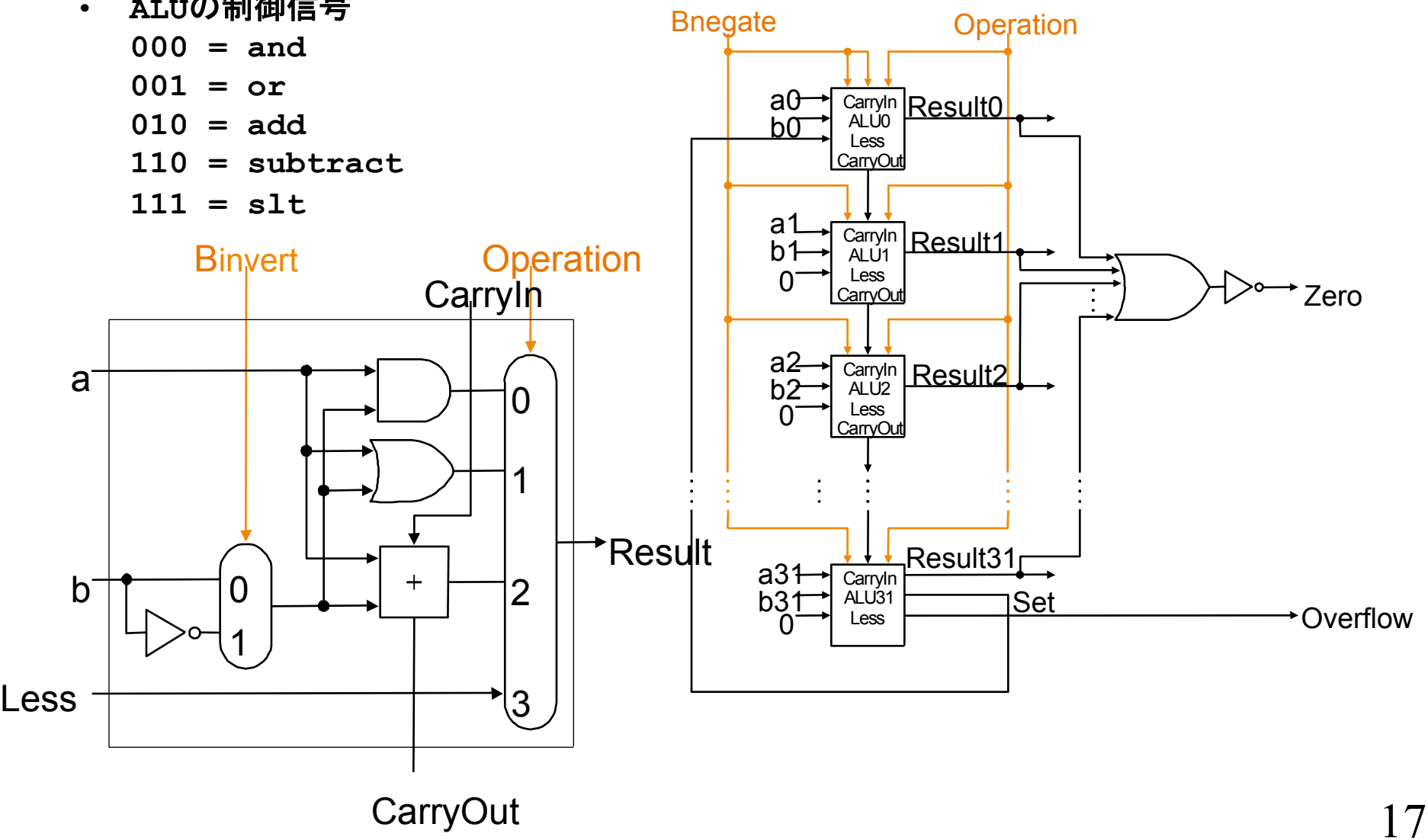
- ALU に対する制御入力(復習)

```
000     AND
001     OR
010     add
110     subtract
111     set-on-less-than
```

- Q: なぜsubtractのコードは011ではなく110なのであろうか?

ALUの復習

- ALUの制御信号
  000 = and
  001 = or
  010 = add
  110 = subtract
  111 = slt



ALUの制御(1)

- MIPSの命令フィールド
  - 命令の上位6ビットから、命令クラスを現す以下のALUopを導出できたとする

    00 = lw, sw
    01 = beq,
    10= arithmetic&logical

    ALUOp
    インストラクションタイプより計算
  - 一方下位6ビットは、R形式の場合は命令クラス内の具体的な操作を現す。尾っぽ羽、I形式の場合などは、16-bitオフセット値の一部

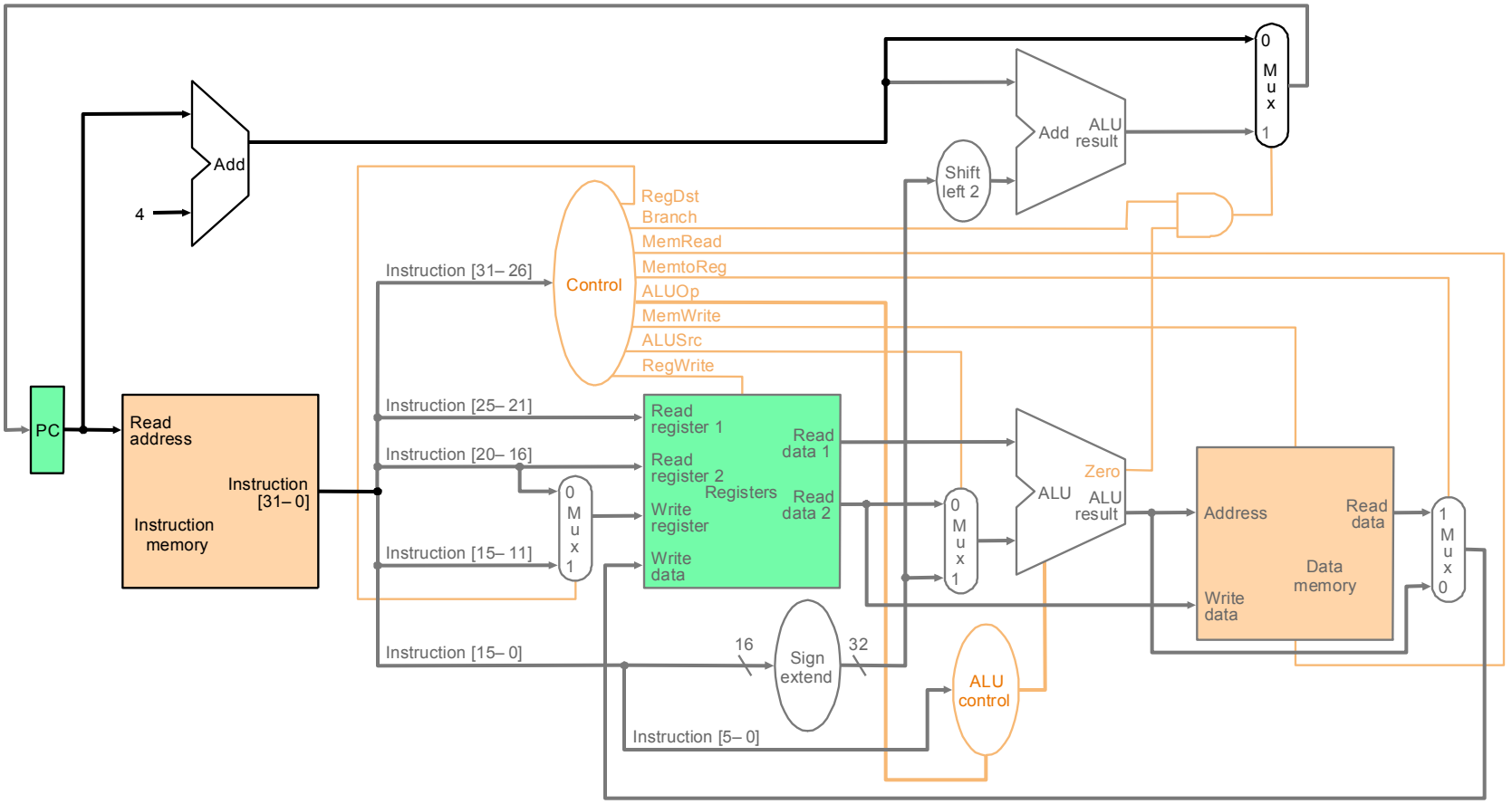
命令操作コード	ALUOp	命令操作	Funct field						ALU制御コード
LW	00	load word	X	X	X	X	X	X	010
SW	00	store word	X	X	X	X	X	X	010
Branch Eq	01	branch equal	X	X	X	X	X	X	110
R形式	10	add	1	0	0	0	0	0	010
R形式	10	subtract	1	0	0	0	1	0	110
R形式	10	AND	1	0	0	1	0	0	000
R形式	10	OR	1	0	0	1	0	1	001
R形式	10	set on less than	1	0	1	0	1	0	111

ALUの制御(2)

- 先の3bitのALUの制御入力を計算する
  - 算術演算の場合は、function codeも見る
  - lw, swの場合は、そのフィールドはオフセット値なので、無視
- 真理値表による表現: (マージして最適化後)
  - Xは don't care (0でも1でもよい)➔回路を都合よく単純化
  - Q: この真理値表を実現する組合せ回路を実現せよ

ALUOp		Funct field						ALU入力
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

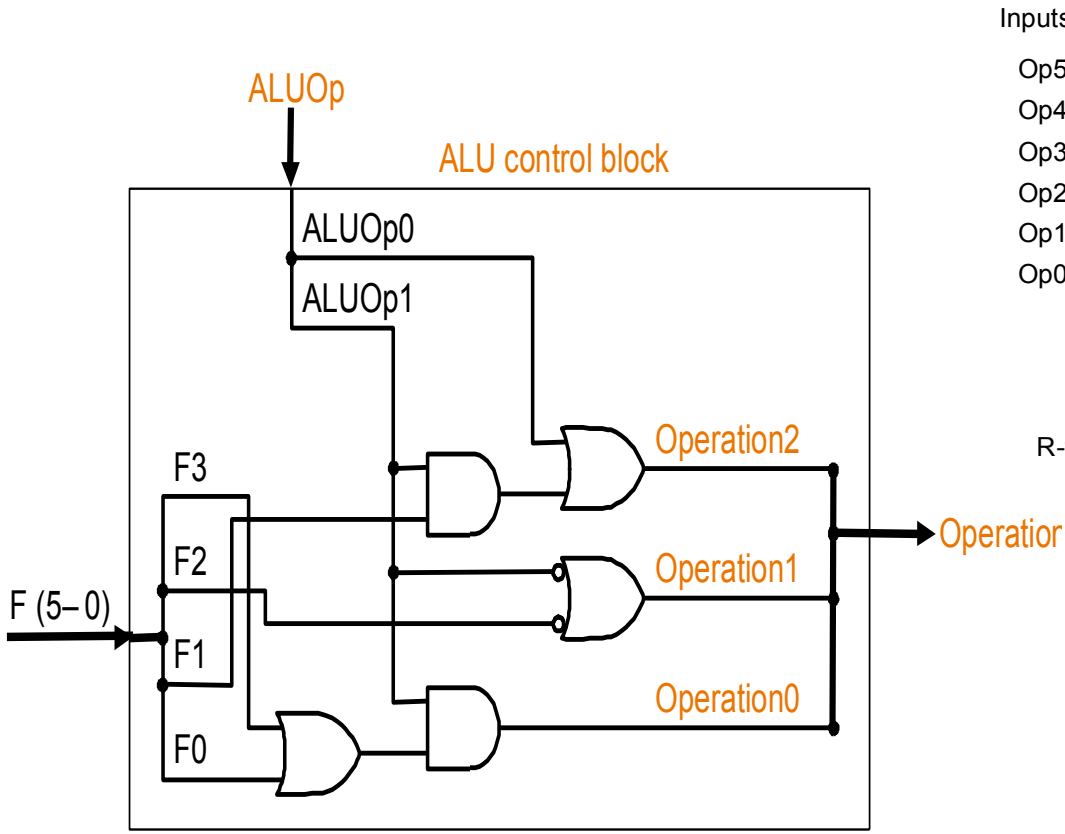
制御 (全体)



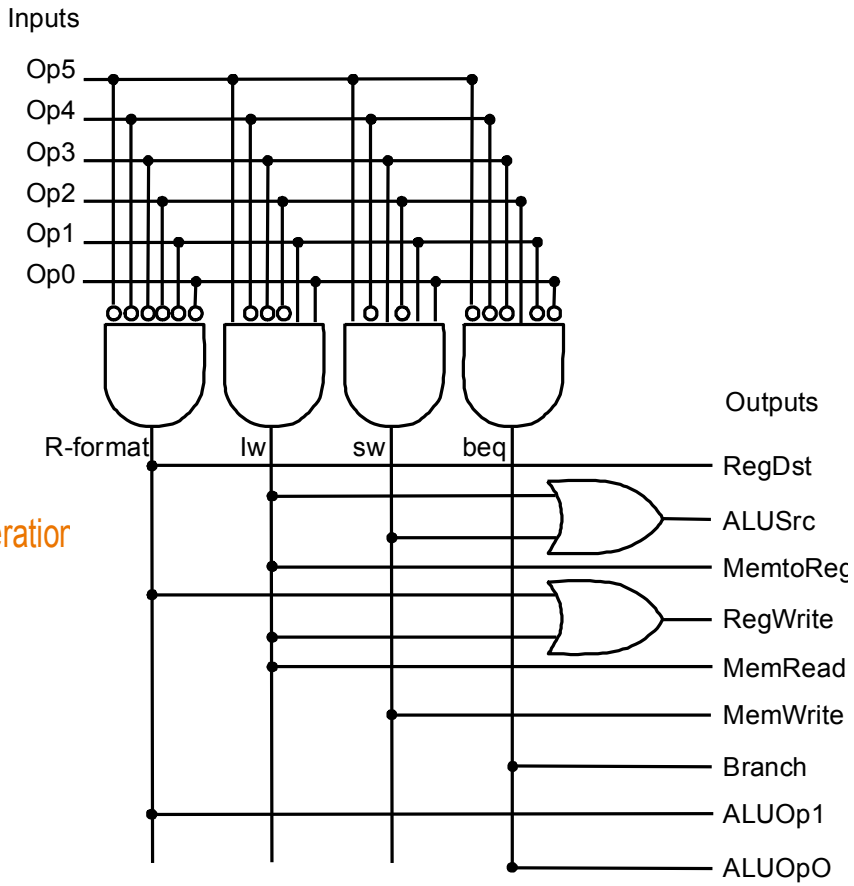
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

制御 (回路)

- 真理値表を、組合せ回路として設計すれば良い



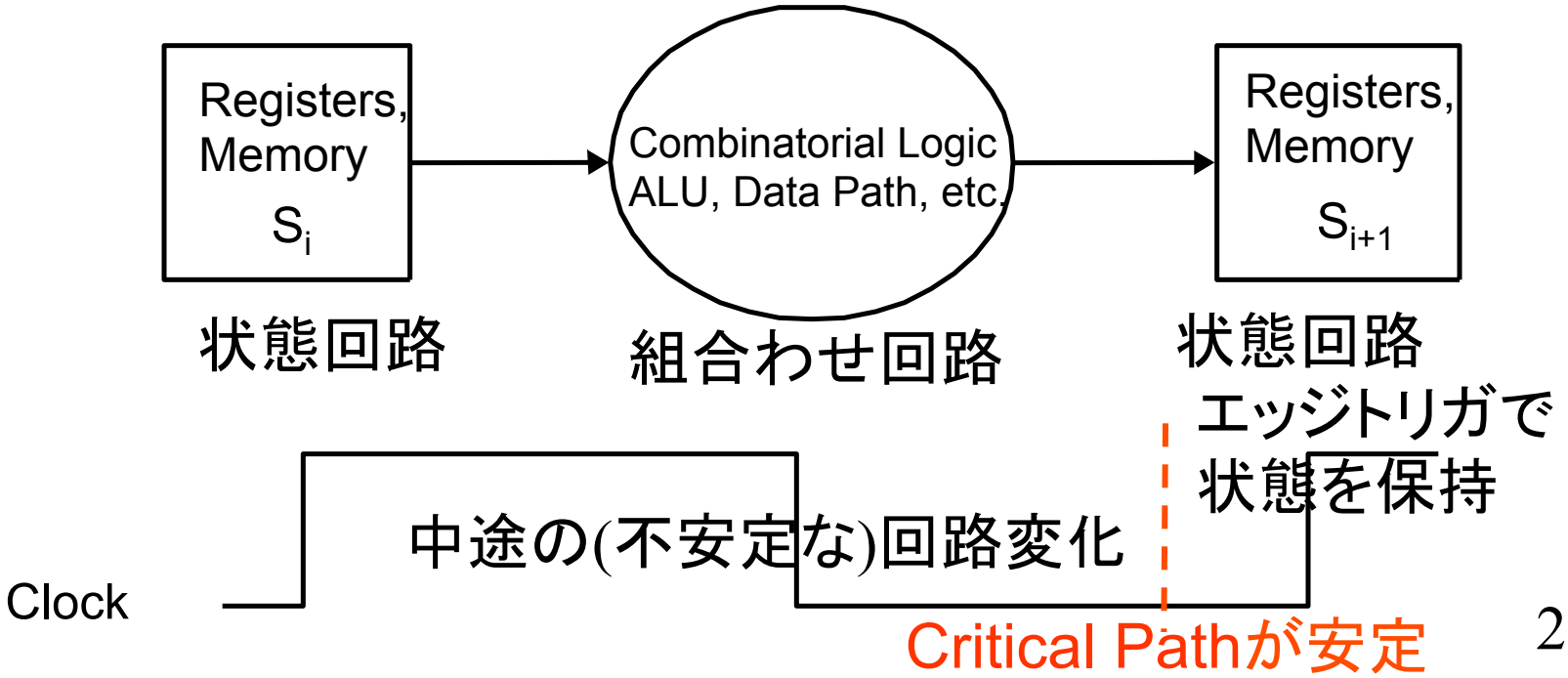
ALU Control



(Main) Control

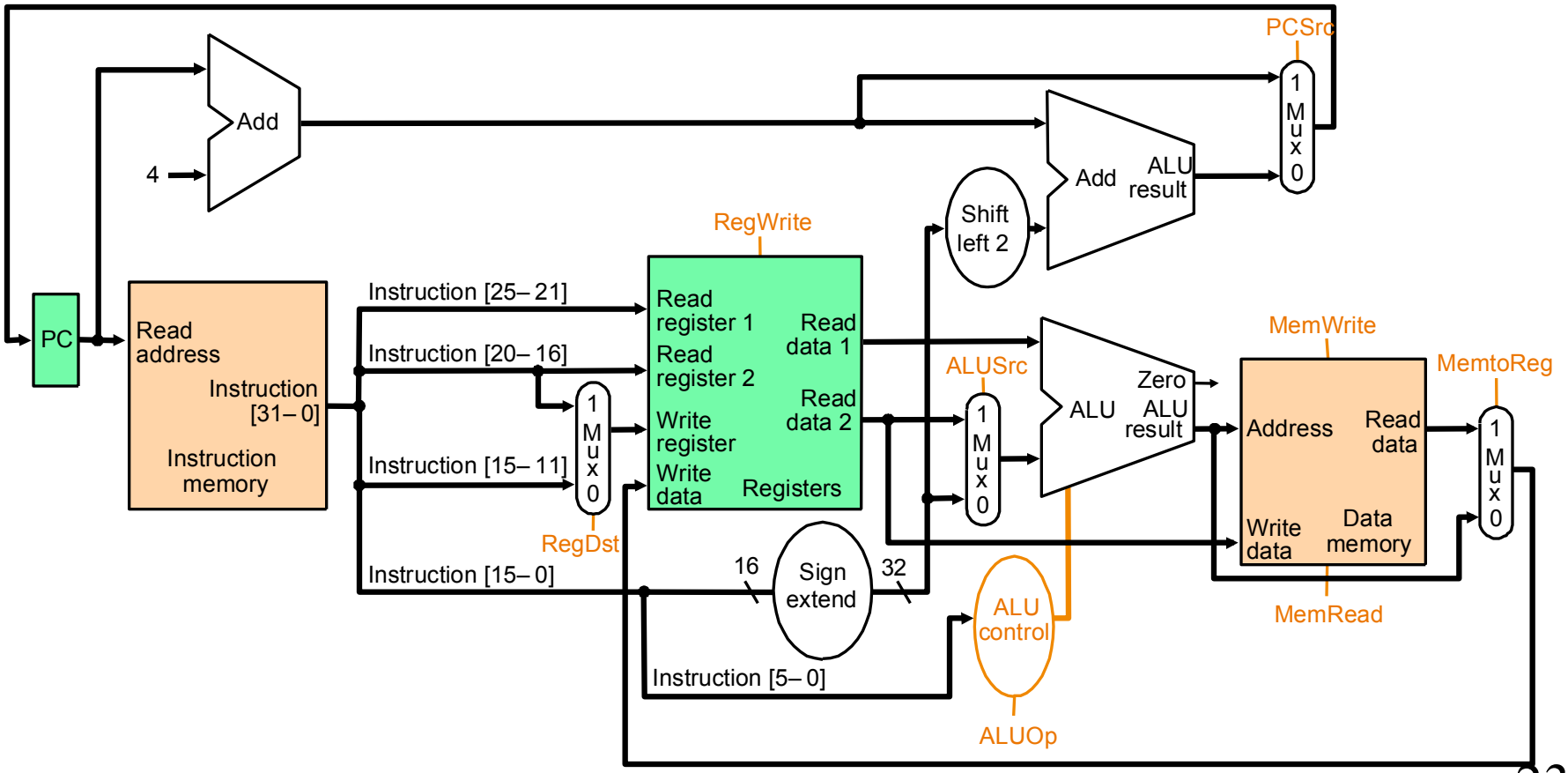
シングルサイクル設計におけるクリティカルパス

- すべての組み合わせ回路の出力が安定し、正しい出力を出すまでつ必要
  - ALU はすぐには「正しい答え」を出さない
  - そこで、クロック信号と書き込み信号を用いて、いつ書き込むべきか、を決定する。(つまり、書き込みの準備が整ったら、書き込みの信号線を真にしておき、クロックのエッジで書き込むようにする。)
- サイクルタイムは、回路のクリティカルパス(もっとも遅い部分)で決定される。
- クリティカルパスの安定時間よりサイクルタイムが短い場合=>どうなる?



シングルサイクル実装におけるクリティカルパスと、最短のサイクルタイム計算

- **Q: 以下の回路で、サイクルタイムを計算せよ。ただし、以下の遅延以外は無視できるものとする:**
  - **メモリ (2ns), ALU と加算器 (2ns), レジスタファイルの読み書き (1ns)**



今後の設計について

- シングルサイクル実装の問題点:
  - クリティカルパスによる性能の低下→浮動小数点演算のように、時間がかかる命令があったらどうなるか?
  - チップ面積の無駄にもなる
- 解決法:
  - より短いサイクルタイムを実現する
  - 命令によってサイクル数を可変にする
  - マルチサイクルなデータパスの設計: