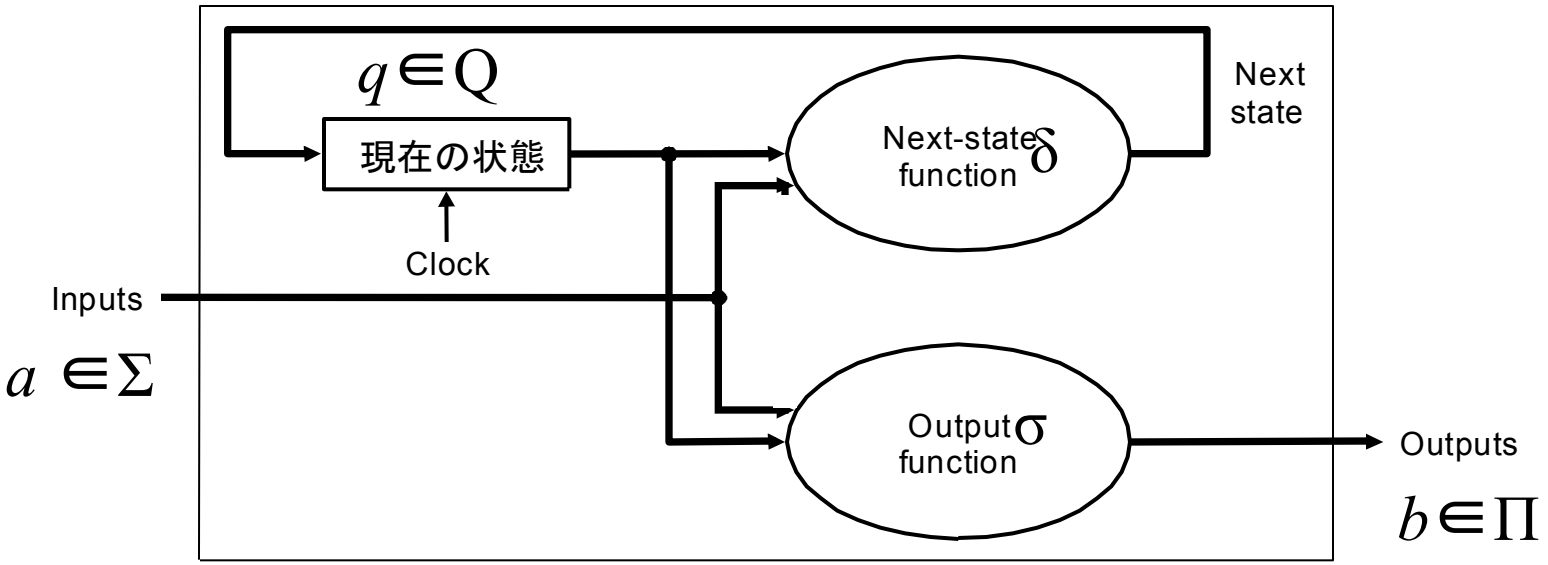
第7回 マルチサイクル実装
2011/6/25 (リモート授業)

マルチサイクル (Multicycle)による設計

- 複数のクロックサイクルで、一つの命令を実行
  - 命令の複雑さによって実行サイクル数が異なる
  - 遅い命令によりCritical Pathが制限されない
- 機能ユニットは、シングルサイクル設計から再利用する
  - ALUは算術論理演算&アドレス計算とPCの+1に重複使用
  - メモリは命令とデータを保持する➔真のvon Neumannアーキ
- マルチサイクル実装では、シングルサイクル実装と異なり、命令語のみでは制御信号を決定することができない
  - 例: ALUは引き算命令では何を行えば良いか?
  - 先行するサイクルの状態にも依存
- 命令実行中のさまざまな複雑な制御のために、有限状態機械(finite state machine)を用いる
  - どのようにハードで実装するのか?

有限状態機械(Finite State Automaton, FSA)について

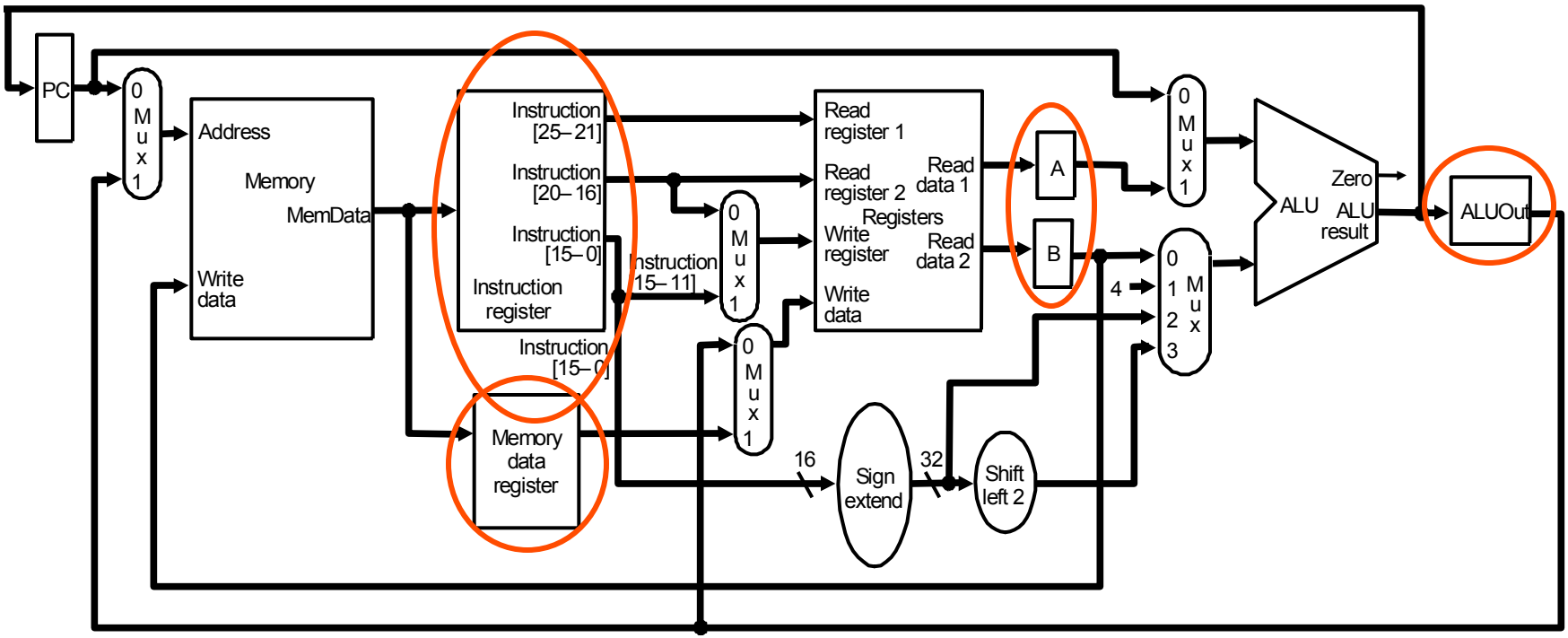
- 有限状態機械の定義:
  - 状態集合 Q
  - 入力アルファベット Σ、出力アルファベット Π
  - 状態遷移関数(Next State Function) δ : Q × Σ → Q (次の状態は、現在の状態と入力によって決定)
  - 出力関数 (output function) σ : Q x Σ →Π (出力も、現在の状態と入力によって決定)



  - このようなFSAを、Mealy機械という(詳しくは別授業) c.f. Moore機械(出力は状態のみに依存)

マルチサイクル設計の基本

- 命令の処理を複数の段階に分ける。それぞれの段階での処理が1クロックサイクルかかるように設計
  - それぞれの段階での処理量をバランスさせる
  - それぞれのサイクルでは一つの機能ユニットのみが動作するようにする
- それぞれのサイクルの終わりには
  - 後のサイクルでの処理のため、処理結果を格納
  - 中間結果を保持する「内部レジスタ」を導入



五段階の実行ステップ

- 命令フェッチ (Instruction Fetch)
- 命令デコードとレジスタ値のフェッチ (Instruction Decode and Register Fetch)
- 実行、メモリアドレス計算、またはブランチ処理 (Execution, Memory Address Computation, or Branch Completion)
- メモリアクセス、またはR-typeの命令の実行完了 (Memory Access or R-type instruction completion)
- (レジスタへの)結果の書き込み (Write-back)

1命令は実行するのに3-5サイクル程度かかる!

Q.シングルサイクル実装に対して有利か?

第 1 ステップ: 命令フェッチ

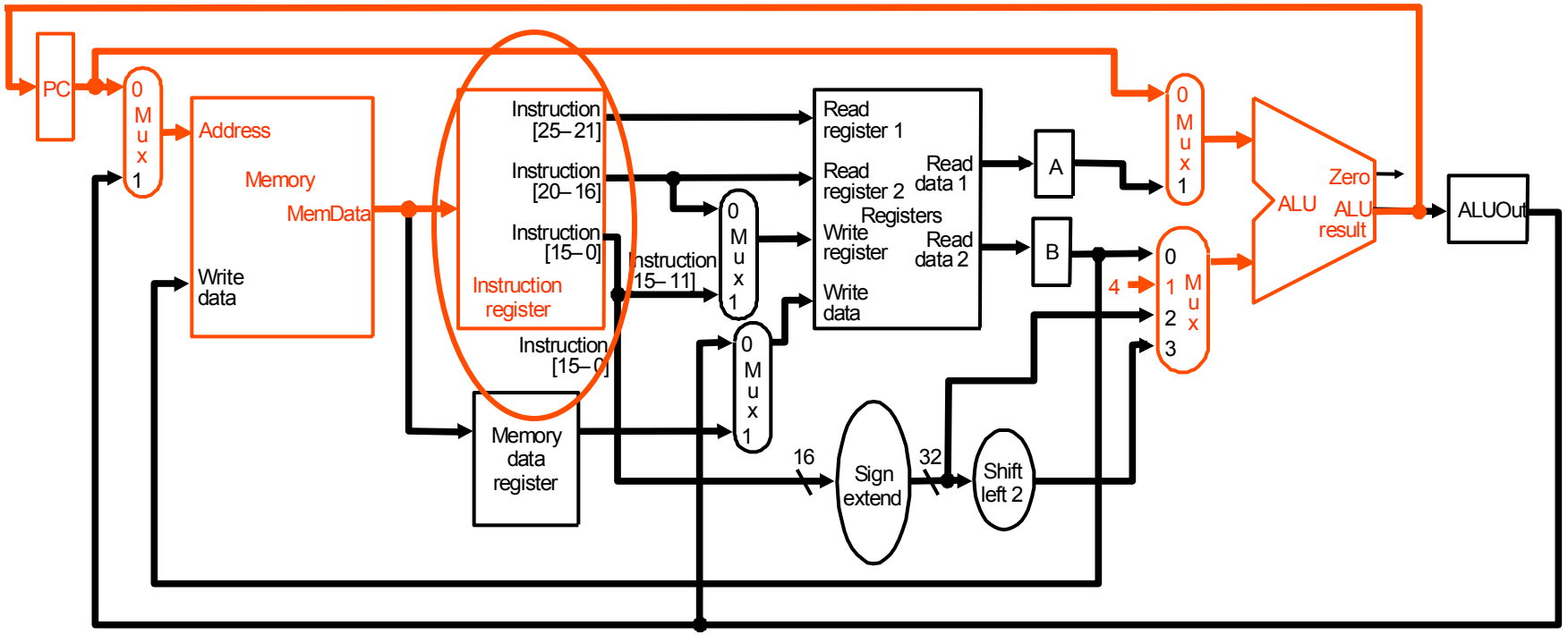
- PCが指し示すメモリの番地から命令を読み込んで、命令レジスタに書き込む
- PCに4を加算して、結果を再びPCに書き込む(次の命令のアドレス)
- 以下の疑似コードで動作を説明できる:

```
IR = Memory[PC];
PC = PC + 4;
```

Q: どのように制御信号を操れば良いか?
この段階でPCに加算をしておくメリットは?

第 1 ステップ： 命令フェッチ(続き)

- 起動されたデータパスと機能ユニット(赤で表示)
  - ALUがPCの加算に使われていることに注意



第 2 ステップ: 命令デコードとレジスタ値のフェッチ

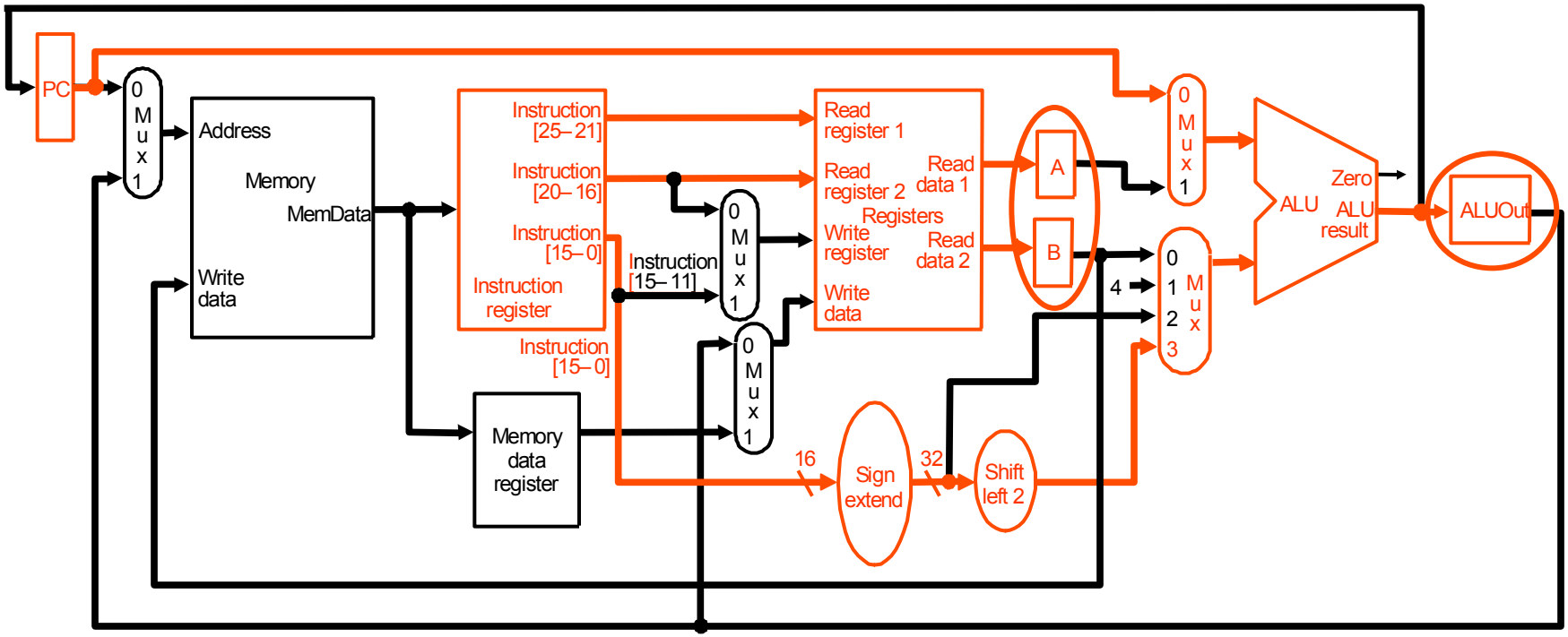
- 命令レジスタに格納された命令語のビットフィールドで指定されるrs レジスタとrt レジスタを読み込んでおく (必要な場合のために←命令形式によっては必要ないかもしれないが)
- 命令がブランチ命令の場合に備えて、ブランチ先のアドレスを計算しておく
- 疑似コードでは:

```
A = Reg[IR[25-21]]; (命令レジスタのビットフィールド25-21)
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
        (ALUによるブランチ先アドレスの計算)
```

- ここでは、命令タイプによって制御信号を変えない
  (制御回路で命令デコードしているのに忙しい)

第 2 ステップ: 命令デコードとレジスタ値のフェッチ(続き)

- 起動されたデータパスと機能ユニット
  - `A = Reg[IR[25-21]];`
  - `B = Reg[IR[20-16]];`
  - `ALUOut = PC + (sign-extend(IR[15-0]) << 2);`



第 3 ステップ: 命令実行 (命令形式に依存)

- ALU は、命令形式の種類によって、以下の三つの異なる動作を行う
- (1) メモリ参照 (I-形式):

```
ALUOut = A + sign-extend(IR[15-0]);
```

- (2) R-形式:

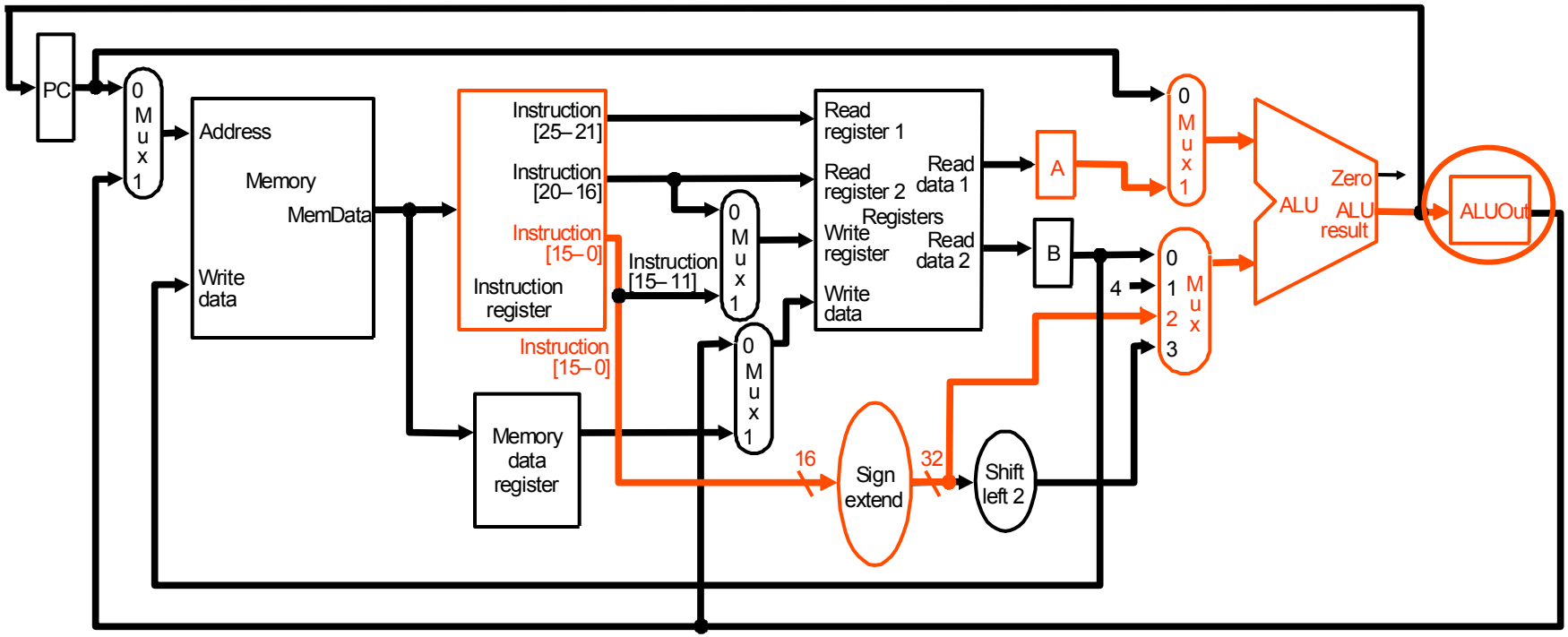
```
ALUOut = A op B;
```

- (3) ブランチ:

```
if (A==B) PC = ALUOut;
```

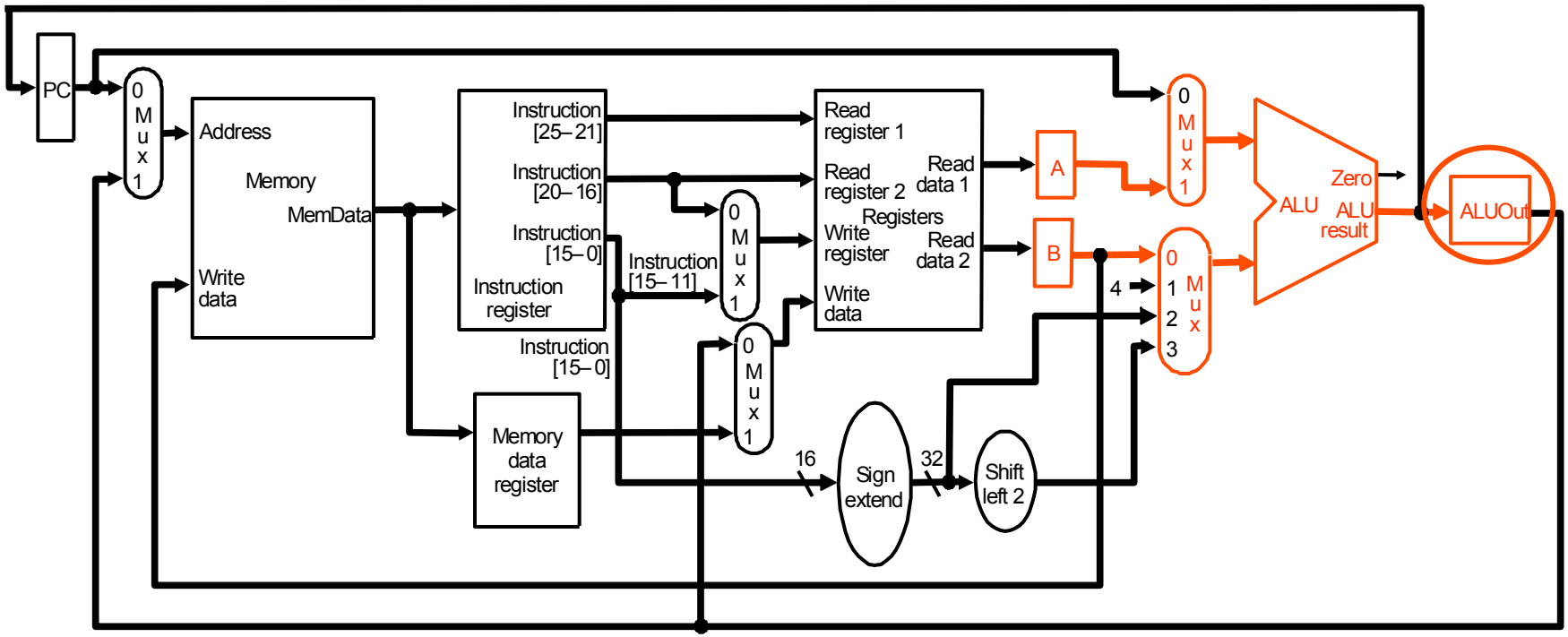
第 3 ステップ:命令実行 (1)メモリ参照 (I-形式)

- 起動されたデータパスと機能ユニット(赤で表示)
  - `ALUOut = A + sign-extend(IR[15-0])`



第 3 ステップ:命令実行 (2) R-形式

- 起動されたデータパスと機能ユニット(赤で表示)
  - `ALUOut = A op B`

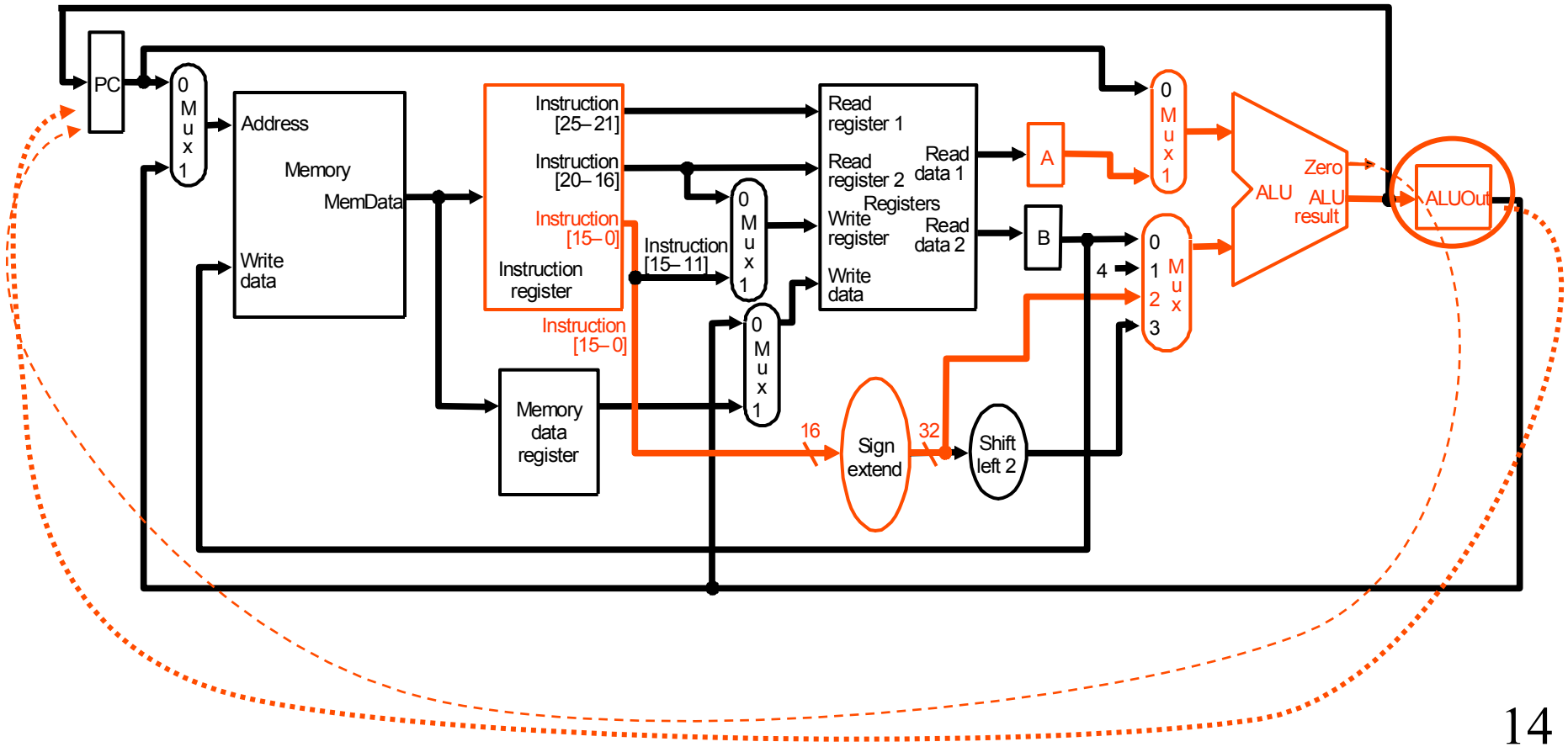


第 3 ステップ:命令実行 (3)ブランチ

- 起動されたデータパスと機能ユニット(赤で表示)
  - `if (A==B) PC = ALUOut`
  - Q. どこかにバグがある → 修正すべし!

第 3 ステップ:命令実行 (3)ブランチのバグ修正のヒント

- 以下を追加回路で実現する
  - `if (A==B) PC = ALUOut`



第 4 ステップ: (R-形式 又は メモリ参照)

- ロード命令とストア命令によるメモリ参照

```
(1) MDR = Memory[ALUOut];
      又は
(2) Memory[ALUOut] = B;
```

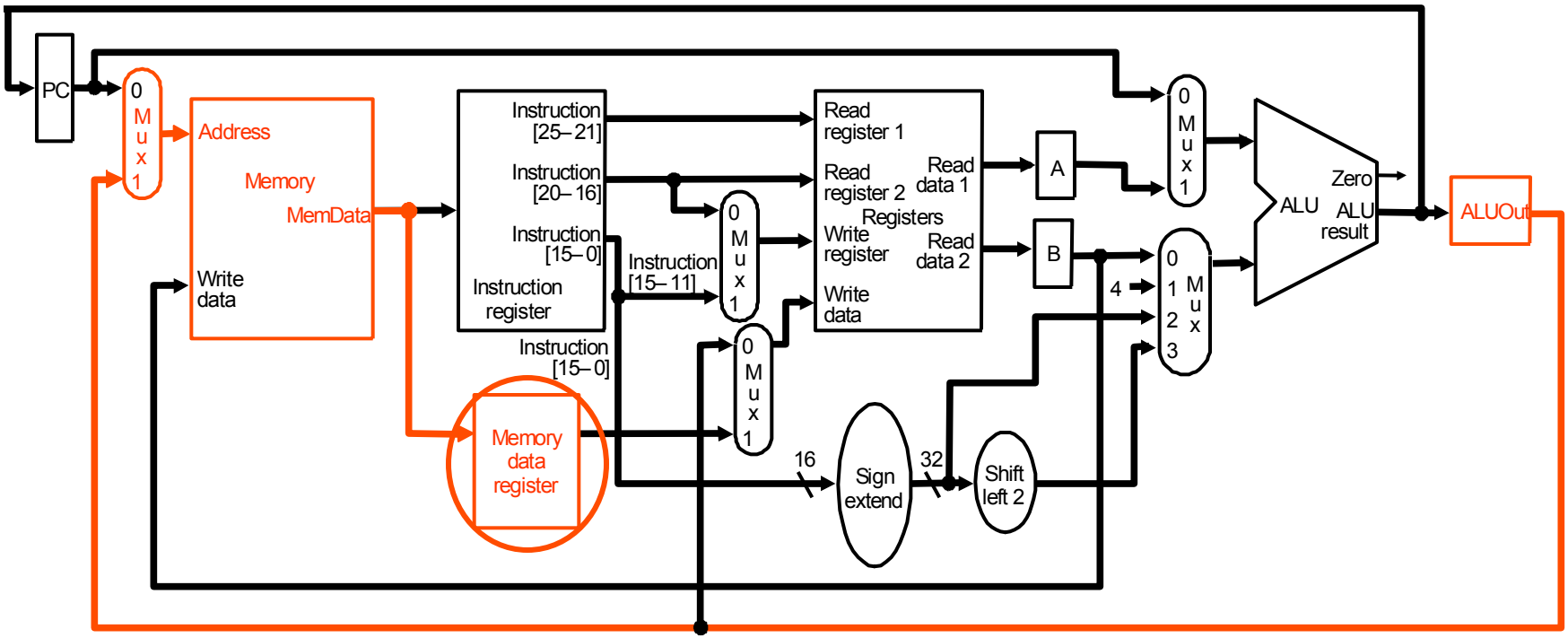
- R-形式命令の完了

```
(3) Reg[IR[15-11]] = ALUOut;
```

メモリやレジスタへの書き込みは、実際はクロックサイクルの完了のエッジで行われる

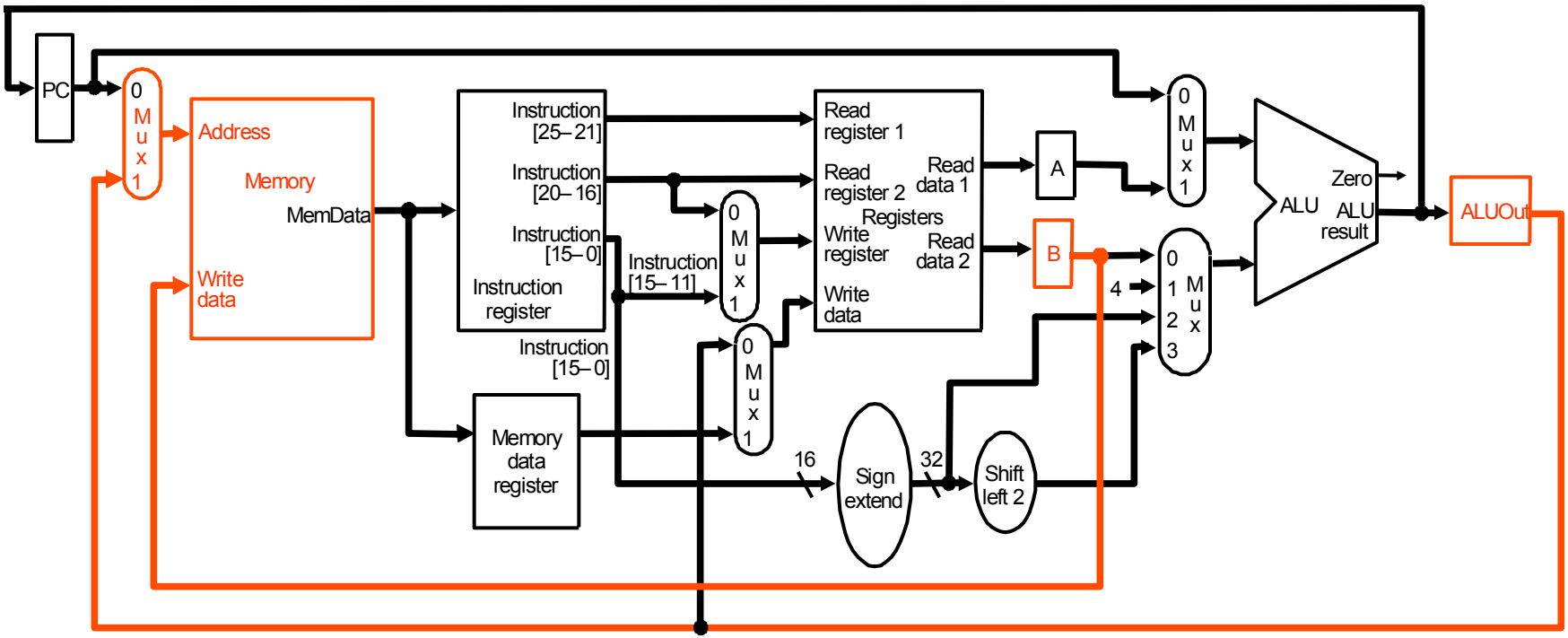
第 4 ステップ:命令実行 (1)メモリ参照 (ロード)

- 起動されたデータパスと機能ユニット(赤で表示)
  - `MDR = Memory[ALUOut]`



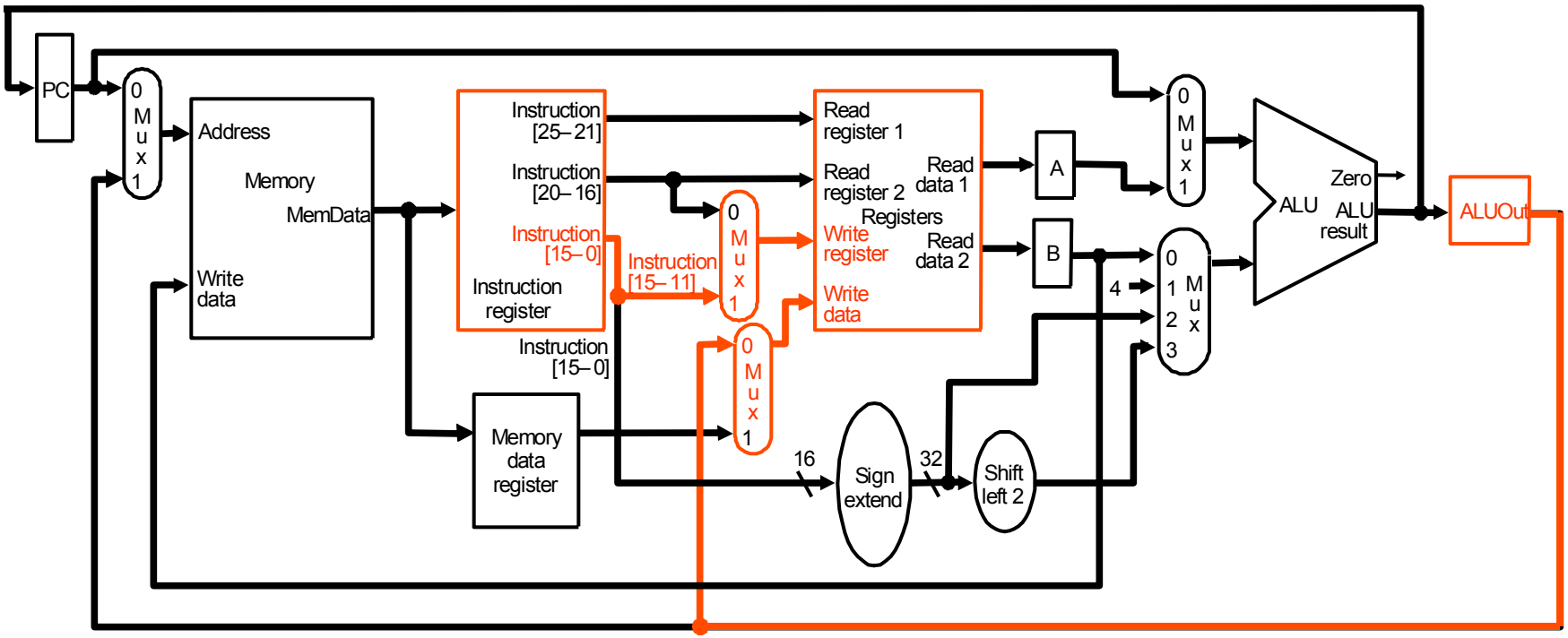
第 4 ステップ:命令実行 (2)メモリ参照 (ストア)

- 起動されたデータパスと機能ユニット(赤で表示)
  - `Memory[ALUOut] = B`



第 4 ステップ:命令実行 (3) R-形式命令の完了)

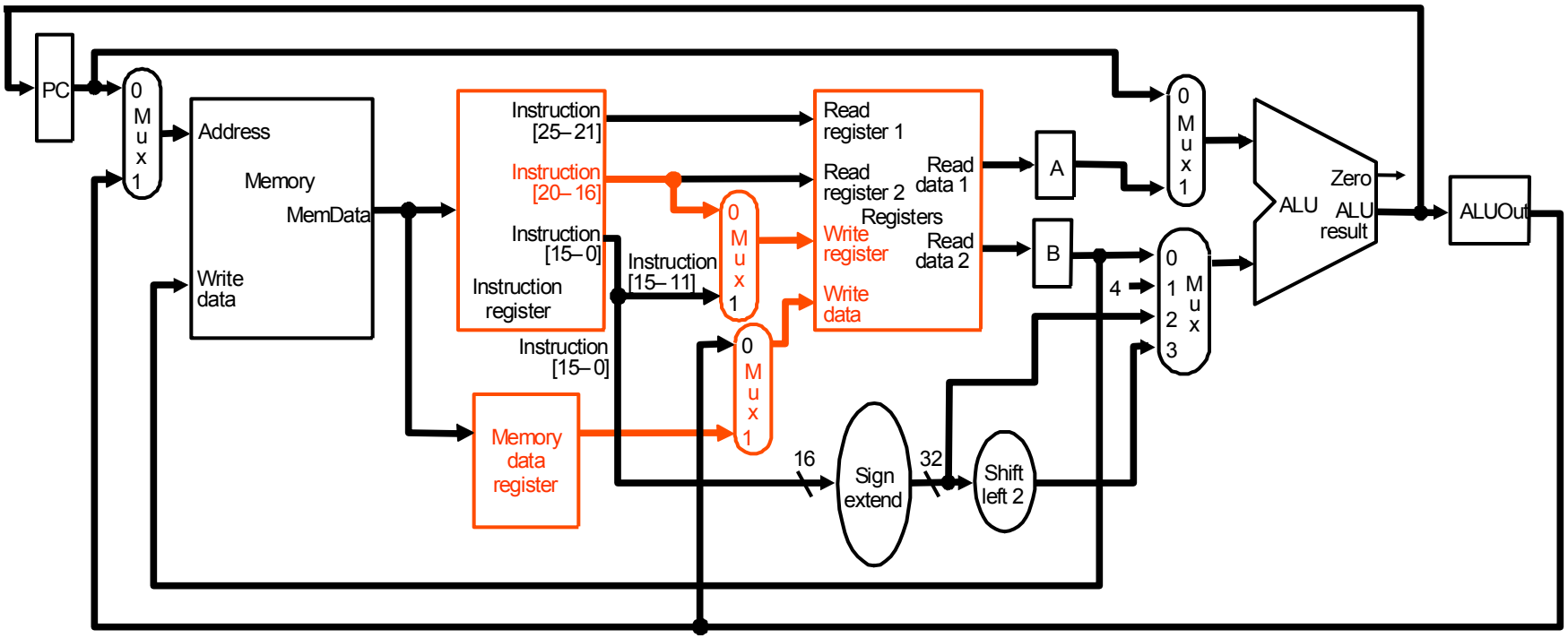
- 起動されたデータパスと機能ユニット(赤で表示)
  - `Reg[IR[15-11]] = ALUOut`



第 5 ステップ: Write-back step

- `Reg[IR[20-16]]= MDR;`

他の命令の場合は(算術演算、ブランチなど)?



まとめ:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	⓪ IR = Memory[PC]			
	PC = PC + 4			
Instruction	① A = Reg [IR[25-21]]			
decode/register	B = Reg [IR[20-16]]			
fetch	ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address	ALUOut=A op B	ALUOut = A + sign-extend	if (A ==B) then	PC = PC [31-28]
compute, branch/	⑥	② (IR[15-0])	PC = ALUOut	II (IR[25-0]<<2)
jump completion			⑧	⑨
Memory access or	Reg [IR[15-11]]=	Load: MDR=Memory[ALUOut] ③		
R-type completion	ALUOut	or		
	⑦	Store: Memory [ALUOut] = B ⑤		
Memory read completion		Load: Reg[IR[20-16]] = MDR ④		

ここでの問題:

- 以下のコードを実行するのに、何サイクルかかるか?

```
    lw $t2, 0($t3)
    lw $t3, 4($t3)
    beq $t2, $t3, Label #ブランチしないと仮定
    add $t5, $t2, $t3
    sw $t5, 8($t3)
Label:   ...
```

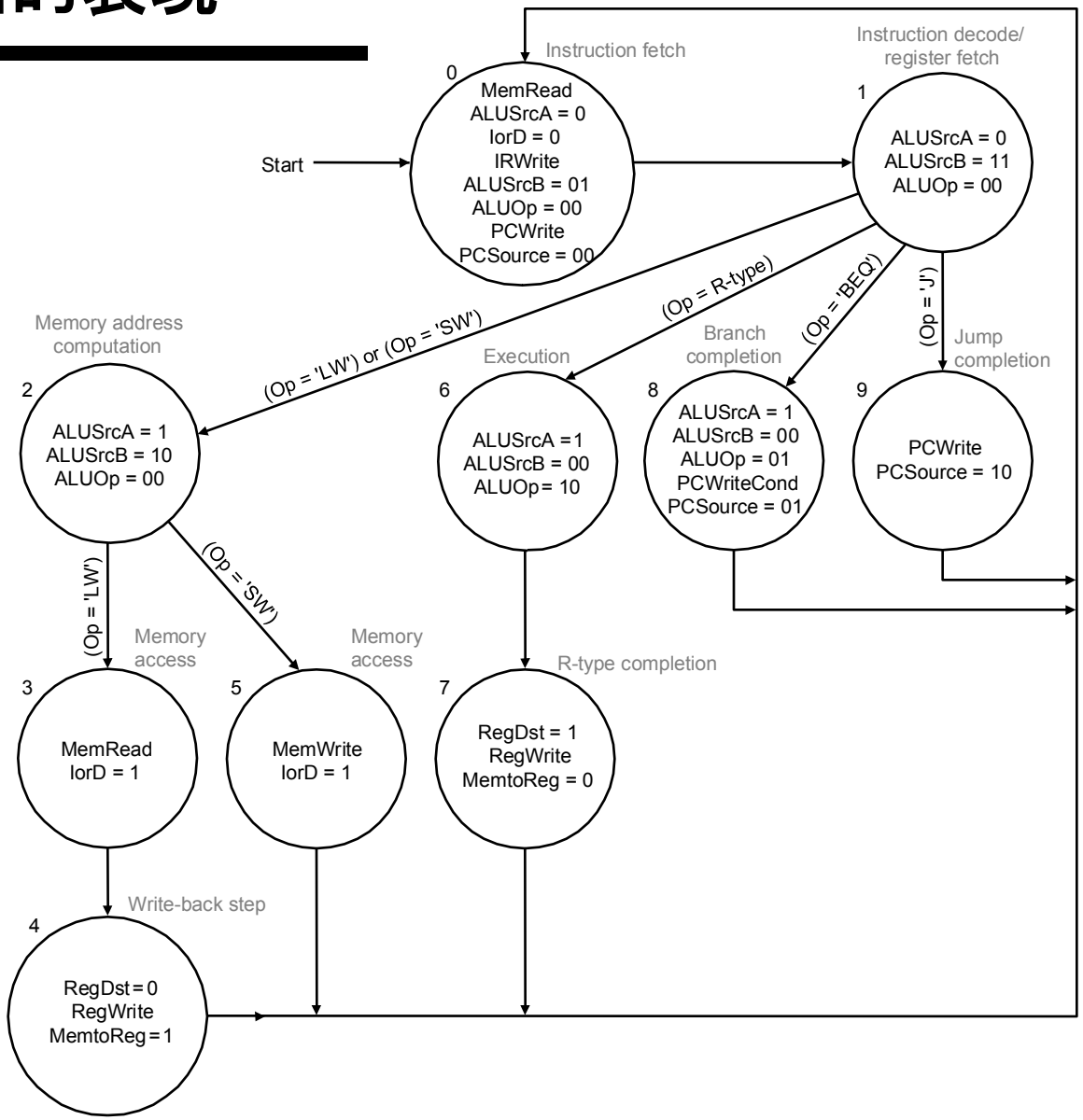
- 実行時の第 8 サイクル目には、何が起こっているか?
- $t2と$t3の加算は第何サイクル目に実行される?

制御回路の実装

- 機能ユニットの制御信号は、以下のものに依存する:
  - 実行する命令
  - 命令の第何ステップを実行しているか

- 今までのデータパスの制御の仕様を基に、命令の状態に応じて制御信号を出力する
- どのように設計するか?
  - (1) 有限状態機械を図示・δ, σを「直接」定義する。
  - (2) マイクロプログラミングの手法を用いる

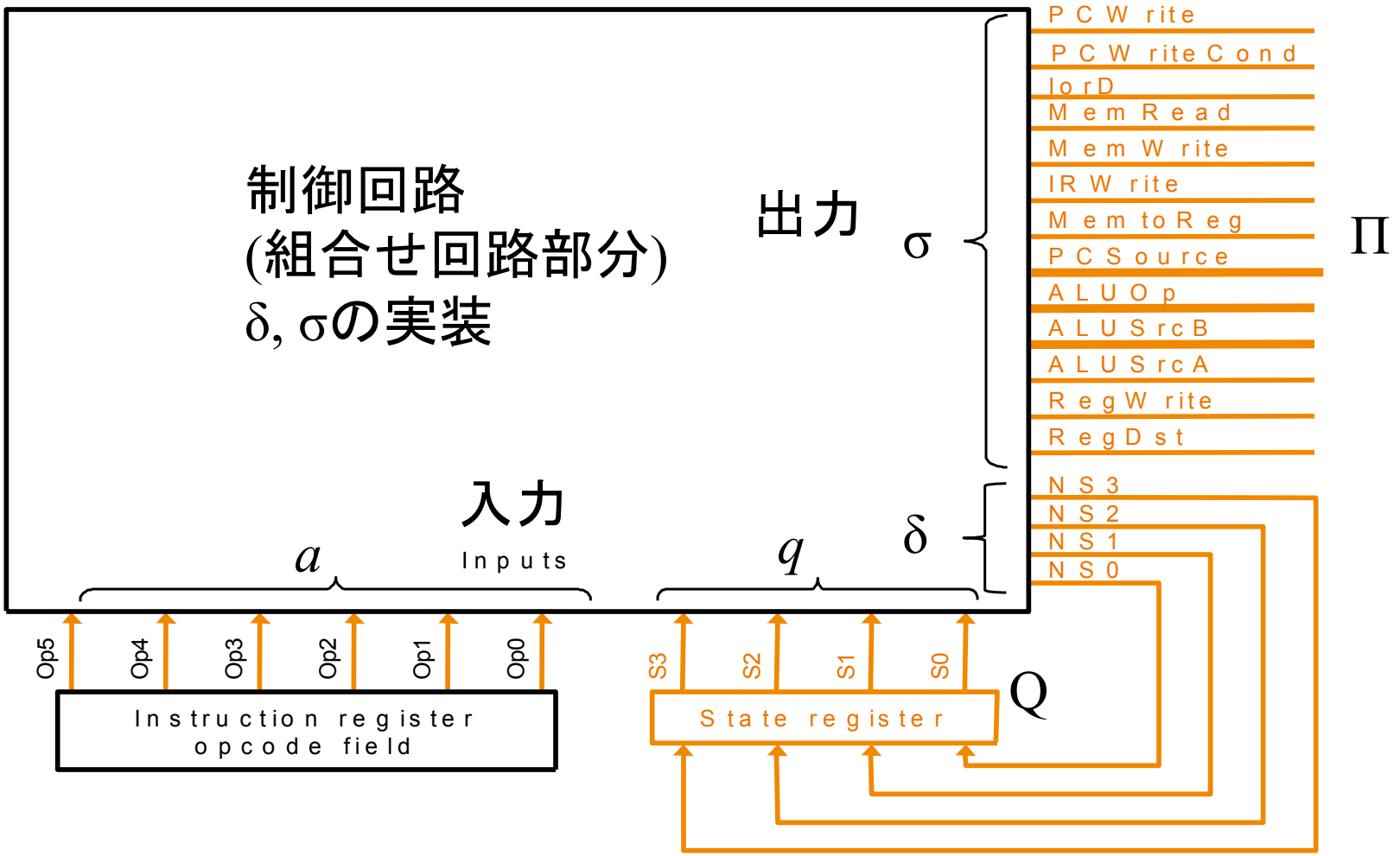
有限状態機械の図的表現



- 状態を表現するには、何ビット必要?

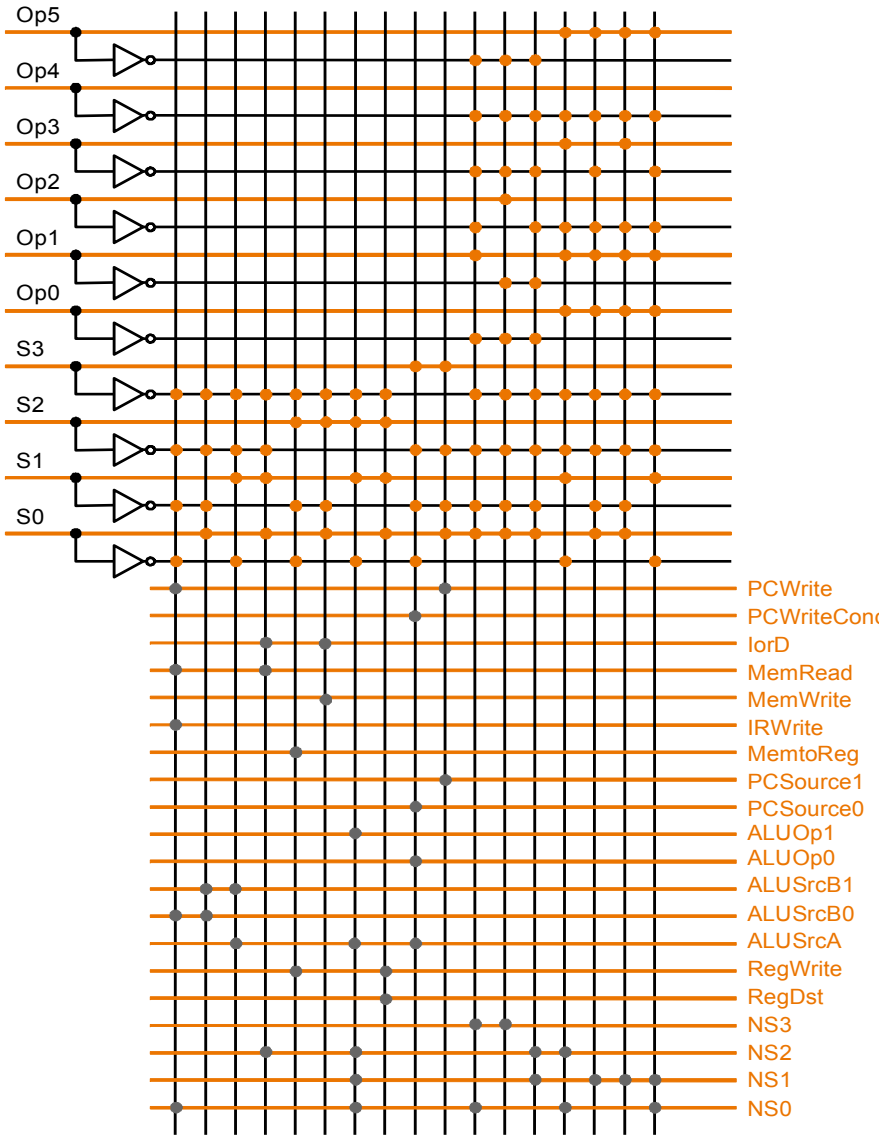
制御用の有限状態機械の設計

- **実装:**



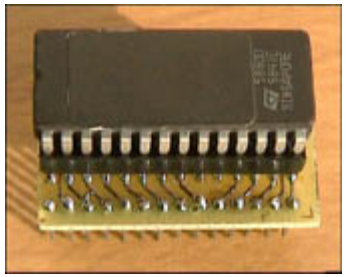
PLA (Programmable Logic Array)による実装

- PLA = 積和標準形を直接回路として「プログラミング」できるもの
- 有限状態機械（Mealy機械)の遷移関数と、出力関数を直接的に表現
- ROMによる実装も可能だが通常はPLAが有利
- 右の図の読み方
  - 上半分の横方向の入力信号線に対し、縦線とオレンジ色の点で結合しているのが積(and)であり、つまり縦線が積項。
  - それらの複数の積項が和(or)で下半分の横線と黒い点結合し、出力されている。
- 例1: $NS0 = S0S1S2S3 + \ldots$
- 例2: $\overline{MemRead} = \overline{S0}S1\overline{S2}\overline{S3} + S0\overline{S1}\overline{S2}\overline{S3}$



状態の命令の読み出し　命令によるデータの読み出し

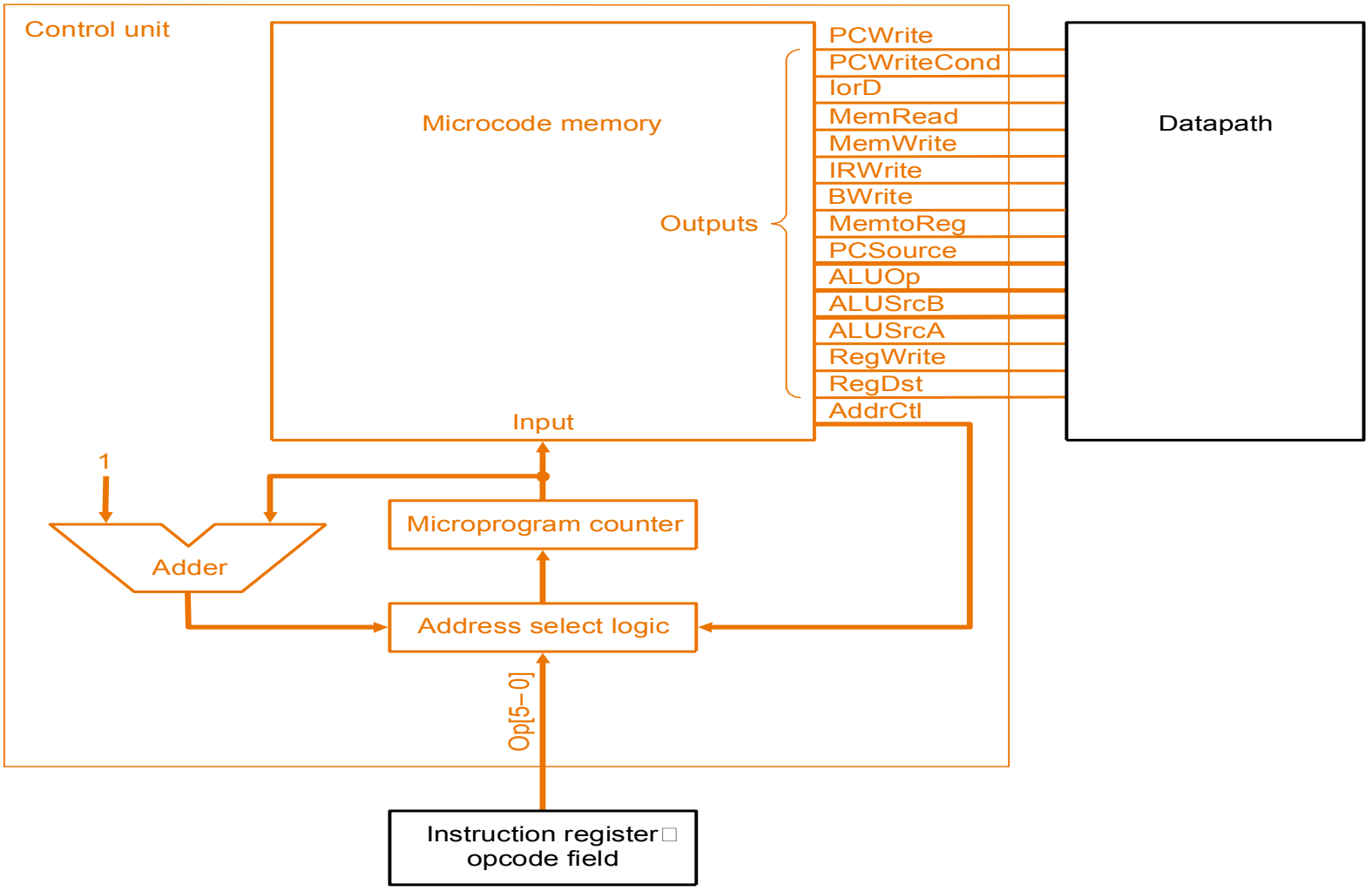
PALとPALプログラマの例



マイクロプログラミング (Microprogramming)

- C.f. ハードワイア (Hardwired logic)
- 有限状態機械の挙動を「プログラミング」する特殊なCPU
- 独自の内部プログラムカウンタや簡単なALUを持つ
- 命令は、機能ユニットの制御に特化(マイクロ命令)
  - 出力として、機能ユニットの制御線を持つ
  - 算術命令などは普通ない
  - 複数の制御線の計算を並列に行う(水平マイクロプログラミング)
- マイクロプログラムはしばしばチップ外のROMかRAMに
- マイクロプログラム部分をプログラマブルにすることにより、命令の追加や変更が可能

マイクロプログラミング (Microprogramming)



- マイクロ命令は、どのような形式 ？

マイクロ命令　(Microinstructions)

- PLAなどによる直接実装に比較して
  - 多数の命令数、アドレッシングモード、サイクルのとき適切
  - マイクロ命令により、信号線を記号的に指定(マイクロアセンブラ)
  - ラベルは実際にはマイクロ命令の「アドレス」に変換

	Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
0	Fetch	Add	PC	4		Read PC	ALU	Seq
1		Add	PC	Extshft	Read			Dispatch 1
2	Mem1	Add	A	Extend				Dispatch 2
3	LW2					Read ALU		Seq
4					Write MDR			Fetch
5	SW2					Write ALU		Fetch
6	Rformat1	Func code	A	B				Seq
7					Write ALU			Fetch
8	BEQ1	Subt	A	B			ALUOut-cond	Fetch
9	JUMP1						Jump address	Fetch

マイクロ命令の形式

Field name	Value	Signals active	Comment
	Add	ALUOp = 00	Cause the ALU to add.
ALU control	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for
			branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
	B	ALUSrcB = 00	Register B is the second ALU input.
SRC2	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
	Read		Read two registers using the rs and rt fields of the IR as the register
			numbers and putting the data into registers A and B.
	Write ALU	RegWrite,	Write a register using the rd field of the IR as the register number and
Register		RegDst = 1,	the contents of the ALUOut as the data.
control		MemtoReg = 0	
	Write MDR	RegWrite,	Write a register using the rt field of the IR as the register number and
		RegDst = 0,	the contents of the MDR as the data.
		MemtoReg = 1	
	Read PC	MemRead,	Read memory using the PC as address; write result into IR (and
		IorD = 0	the MDR).
Memory	Read ALU	MemRead,	Read memory using the ALUOut as address; write result into MDR.
		IorD = 1	
	Write ALU	MemWrite,	Write memory using the ALUOut as address, contents of B as the
		IorD = 1	data.
	ALU	PCSource = 00	Write the output of the ALU into the PC.
		PCWrite	
PC write control	ALUOut-cond	PCSource = 01,	If the Zero output of the ALU is active, write the PC with the contents
		PCWriteCond	of the register ALUOut.
	jump address	PCSource = 10,	Write the PC with the jump address from the instruction.
		PCWrite	
	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
Sequencing	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

マイクロコード: トレードオフ

- 仕様上の有利さ:
  - デザインが簡単→CISCの複雑な命令も可能
  - アーキテクチャとマイクロコードの設計を同時に行なえる
- 実装上の有利さ (プロセッサチップ外ROM/RAMにプログラム)
  - プログラムがチップ外にあるため、変更が容易
  - 他のアーキテクチャのエミュレーションを可能
- 実装上の不利さ - 遅くなる。:
  - 制御回路を単一チップに収めないと、遅くなる
  - ROM はRAMに対して、遅い (昔は速かった)
- 現在では、複雑な命令にのみ用いる
  - (c.f., RISC vs. CISC)