計算機アーキテクチャ

第12回　命令形式とアセンブリ言語

2014年　6月30日

電気情報工学科　　田島　孝治

授業スケジュール（前期）

回	日付	タイトル
1	4/7	コンピュータ技術の歴史とコンピュータアーキテクチャ
2	4/14	ノイマン型コンピュータ
3	4/21	コンピュータのハードウェア
4	4/28	数と文字の表現
5	5/12	固定小数点数と浮動小数点表現
6	5/19	計算アーキテクチャ（ARU）
7	5/26	計算装置のハードウェア実装
8	6/2	文字コード
9	6/11	中間試験（9:00-9:50）

回	日付	タイトル
10	6/16	主記憶装置とレジスタ
11	6/20	命令実行の流れ
12	6/30	命令形式とアセンブリ言語
13	7/7	命令セット
14	7/14	サブルーチンの実現
15	7/28	PCSpimによるアセンブリ言語プログラム
	8/4?	期末試験（日程は仮）
16	9/29?	フォローアップ（日程は仮）

※5/5はこどもの日、7/21は海の日のため休講

※授業変更：6/23 1時限→6/20 3時限

今日の授業の目的

命令の表現形式について学ぶ
命令のバイナリ表現について学ぶ

今日の具体的な目標

- 算術演算命令の記述方法を覚える
- MIPS形式のプログラムを学び、C言語との変換ができるようになる
- 命令がメモリ上にどんな形式で保存されているのかを知る

命令(instruction)の表現方法

- 命令はーつのデータ(命令語)として表現する
  - メモリ上に保存されている
- データは32bit程度の長さで表される
  - 命令の長さを命令長と呼ぶ
  - 0/1をどう割り当てるかが問題
- 命令の意味を解釈し分類を行う
  - 操作(operation):命令語で表される
  - 操作の対象(operand)
    - ソースオペランド:入力
    - ディスティネーションオペランド:出力

命令の分割例

- 命令：111番地の内容をレジスタ1へ読み込み

分類	内容
操作	
ソースオペランド	
ディスティネーションオペランド	

- 命令：レジスタ1にレジスタ2の値を加算

分類	内容
操作	
ソースオペランド1	
ソースオペランド2	
ディスティネーションオペランド	

命令の分割例

- 命令：111番地の内容をレジスタ1へ読み込み

分類	内容
操作	メモリ読み込み
ソースオペランド	111番地（メモリアドレス）
ディスティネーションオペランド	レジスタ1

- 命令：レジスタ1にレジスタ2の値を加算

分類	内容
操作	加算
ソースオペランド1	レジスタ1
ソースオペランド2	レジスタ2
ディスティネーションオペランド	レジスタ1

命令の種類と形式（復習）

- 算術論理演算命令
  - 加減算や論理演算を行う

ALU制御 （＋,－, AND, OR等）	入力レジスタ1	入力レジスタ2	出力レジスタ

- メモリ操作命令
  - メモリの読み書きを行う

メモリ操作 （load, store等）	レジスタ	アドレス

- 条件分岐命令
  - プログラムの実行順を変更する

分岐操作 （条件分岐、ジャンプ等）	アドレス

各命令へのビット割り当て

- この授業ではMIPSのビット割り当てを説明
  - アーキテクチャによって異なるので注意
  - MIPSは引数の数などで命令を3種類に分ける

- R形式

op	rs	rt	rd	shamt	funct
6bit	5bit	5bit	5bit	5bit	6bit

- I形式

op	rs	rt	Imm/dpl

- A形式

op	addr

具体的な命令の例(R形式)

- R形式の命令

演算命令	命令コード	使用例	意味
加算	add	add $t0,$s0,$s1	t0 = s0 + s1
減算	sub	sub $t0,$s0,$s1	t0 = s0 - s1
乗算1	mul	mul $t0,$s0,$s1	t0 = s0 * s1
乗算2	mult	mult $s0,$s1	HI = s0 * s1 LO = s0 * s1 (上位/下位に分ける)
除算	div	div $s0,$s1	LO = s0 / s1 HI = s0 % s1
比較	slt	slt $t0,$s0,$s1	t0 = (s0 < s1)
算術 右シフト	sra	sra $t0,$s0,5	t0 = s0 >> 5

MIPS形式のプログラムを作ろう

- C言語で次のプログラムが書かれている

```
c = a + b;
d = d + a;
ans = c – d + a;
```

- これをMIPS形式プログラムで記述せよ
  - 一次的な変数としてはレジスタt0~t9を使う
  - その他の変数は次のように割り当て済み
    - ans:$s0, a:s1, b:s2, c:s3, d:s4

MIPS形式のプログラム

```
add      $s3,$s1,$s2            # c = a + b;
```

#以後はコメント文

演習問題1

- 次のC言語で書かれたプログラムをMIP形式に変換せよ

```
c = a * b;
a = a + b;
ans = c – (a + b) * b;
```

  - 一次的な変数としてはレジスタt0~t9を使う
  - その他の変数は次のように割り当て済み
    - ans:$s0, a:s1, b:s2, c:s3

解答欄(演習問題1)

```
add      $s3,$s1,$s2          # c = a + b;
```

#以後はコメント文

R形式の命令をビット列に直す

op	rs	rt	rd	shamt	funct
6bit	5bit	5bit	5bit	5bit	6bit

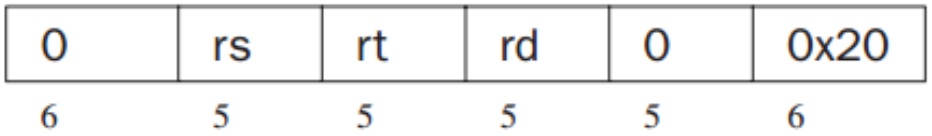
- op: Operation Code(オペコード)
  - 命令語を表す：ただしR形式の場合はfunctも併用する
- rs: source operand 1
- rt: source operand 2
- rd: destination operand
- shamt: shift amount
  - シフト命令の時に利用する
- funct: function
  - 加減乗除などの指定はここで行う

ビット列の作り方

（Reference manual for spim and the MIPS32 instruction setより引用）

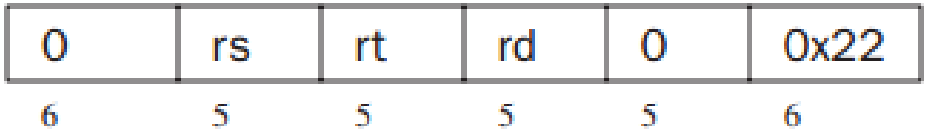
Addition (with overflow)

```
add rd, rs, rt
```



Subtract (with overflow)

```
sub rd, rs, rt
```

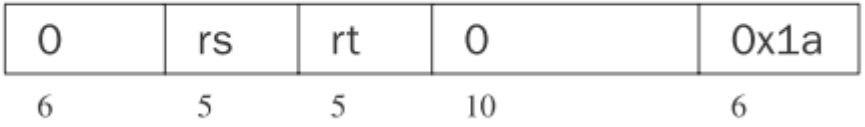


Multiply (without overflow)

```
mul rd, rs, rt
```

Divide (with overflow)

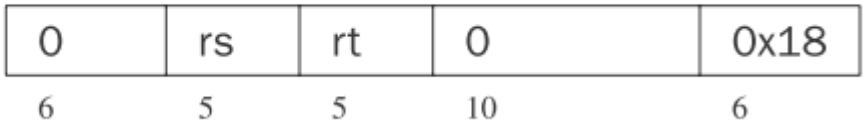
```
div rs, rt
```



ビット列の作り方2

Multiply

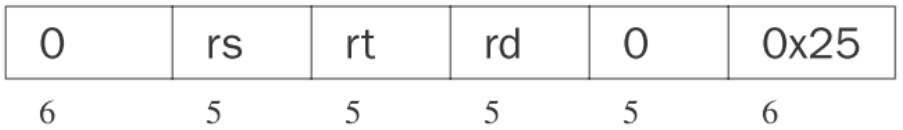
```
mult rs, rt
```



Multiply registers `rs` and `rt`. Leave the low-order word of the product in register `lo` and the high-order word in register `hi`.

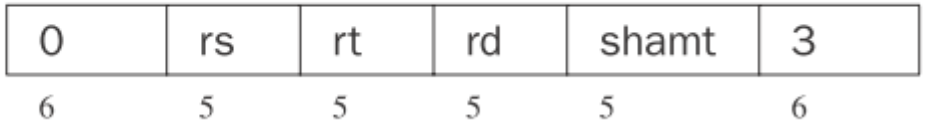
OR

```
or rd, rs, rt
```



Shift right arithmetic

```
sra rd, rt, shamt
```



レジスタはどう表すの？

汎用レジスタは番号が決まっています

Register name	Number	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expression evaluation and results of a function
$v1	3	expression evaluation and results of a function
$a0	4	argument 1
$a1	5	argument 2
$a2	6	argument 3
$a3	7	argument 4
$t0	8	temporary (not preserved across call)
$t1	9	temporary (not preserved across call)
$t2	10	temporary (not preserved across call)

レジスタの番号（続き）

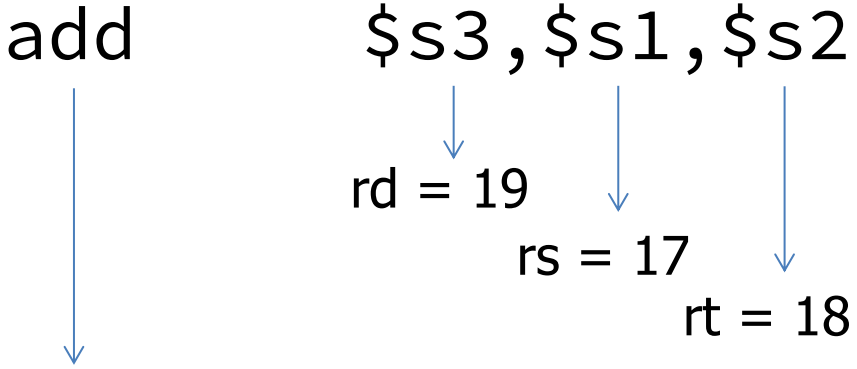
Register name	Number	Usage
$t3	11	temporary (not preserved across call)
$t4	12	temporary (not preserved across call)
$t5	13	temporary (not preserved across call)
$t6	14	temporary (not preserved across call)
$t7	15	temporary (not preserved across call)
$s0	16	saved temporary (preserved across call)
$s1	17	saved temporary (preserved across call)
$s2	18	saved temporary (preserved across call)
$s3	19	saved temporary (preserved across call)
$s4	20	saved temporary (preserved across call)

レジスタの番号（続き）

Register name	Number	Usage
$s5	21	saved temporary (preserved across call)
$s6	22	saved temporary (preserved across call)
$s7	23	saved temporary (preserved across call)
$t8	24	temporary (not preserved across call)
$t9	25	temporary (not preserved across call)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (used by function call)

具体的なビット列

- 先ほど作ったプログラムのビット列に直そう



R形式：op = 0, add:shamt=0,funct =0x20

op	rs	rt	rd	shamt	Funct
0	17	18	19	0	0x20
000000	10001	10010	10011	00000	100000
6	5	5	5	5	6

⇨ 0000 0010 0011 0010 1001 1000 0010 0000 = 0x02329820

例題

- 次のプログラムをビット列に直せ

プログラム	op	rs	rt	rd	shamt	funct
add $t3 $s1 $s2						
sra $t1 $s1 3						
sub $t2 $t1 $s3						
add $s3 $t2 $t3						

演習問題2

- 演習問題1のプログラムをビット列に直せ

プログラム	op	rs	rt	rd	shamt	funct

演習問題2(解答欄続き)

- 演習問題1のプログラムをビット列に直せ

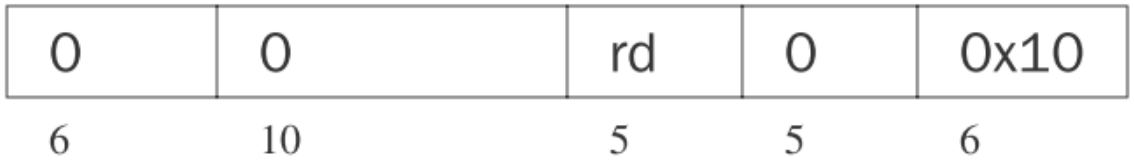
プログラム	op	rs	rt	rd	shamt	funct

R形式の特殊命令

- HIとLOのレジスタはどうやって読み込む？
  - 専用の命令があります

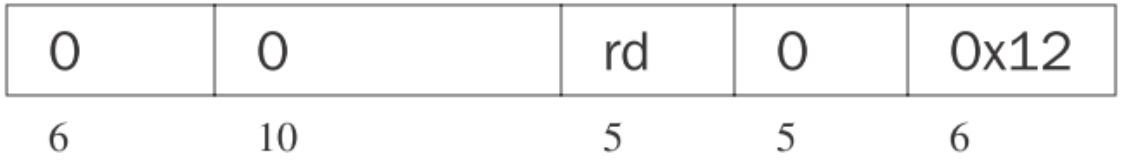
Move from hi

```
mfhi rd
```



Move from lo

```
mflo rd
```



具体的な命令の例（I形式）

- I形式の命令

演算命令	命令コード	使用例	意味
メモリ読み込み	lw	lw $t0 C($s0)	t0 = Memory[$s＋C]
メモリ書き込み	sw	sw $t0 C($s0)	Memory[$s + C] = t0
条件分岐	beq	beq $s,$t,10	2つのレジスタの値が等しい場合に指定したアドレスに処理を移す
条件分岐	bne	bne $s	2つのレジスタの値が等しくない場合に指定アドレスに処理を移す
加算（即値）	addi	addi $t0,$a0,10	t0 = a0 + 10
比較	slti	slti $t0,$a0,10	t0 = (a0 < C)

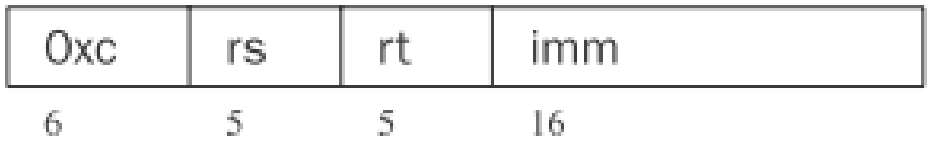
ビット列の作り方（I形式）

Addition immediate (without overflow)

```
addiu rt, rs, imm
```

AND immediate

```
andi rt, rs, imm
```



Set less than immediate

```
slti rt, rs, imm
```

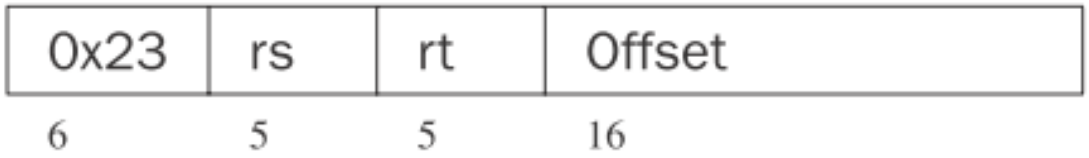
imm は immediate（即値）を表す

つまり、レジスタと直接指定した値を比較したり、直接値を代入したりできる

ビット列の作り方（I形式）２

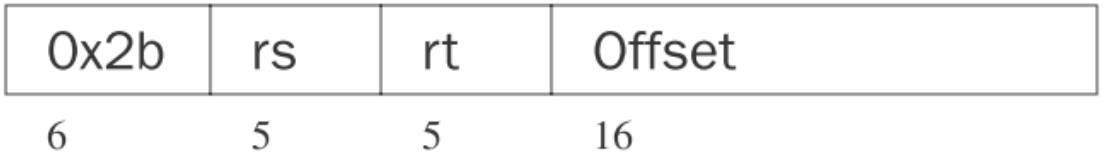
Load word

```
lw rt, address
```



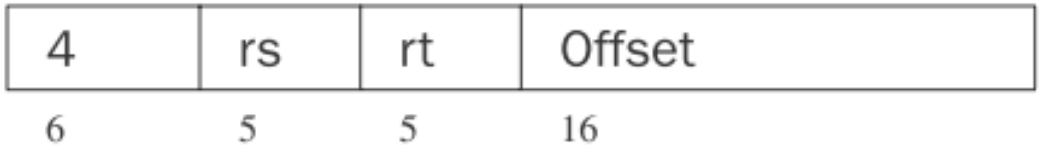
Store word

```
sw rt, address
```



Branch on equal

```
beq rs, rt, label
```



Offsetとアドレスの関係は？？

→ 来週はアドレッシングの説明を行いI形式命令に挑戦します

追加問題(時間が余った時用)

- C言語で次のプログラムが書かれている

```
c = a * b;
d = a % b;
e = b / a;
ans = (a - d) * b / c;
```

- これをMIPS形式プログラムで記述せよ
  - 一次的な変数としてはレジスタt0~t9を使う
  - その他の変数は次のように割り当て済み
    - ans:$s0, a:s1, b:s2, c:s3, d:s4
  - 掛け算にmul命令を使った場合も記述せよ

R形式の命令

op	rs	rt	rd	shamt	funct
6bit	5bit	5bit	5bit	5bit	6bit

- op: Operation Code(オペコード)
  - 命令語を表す：ただしR形式の場合は常に0
- rs: source operand 1
- rt: source operand 2
- rd: destination operand
- shamt: shift amount
  - シフト命令の時に利用する
- funct: function
  - 加減乗除などの指定はここで行う

具体的な命令の例（I形式）

- R形式の命令

演算命令	命令コード	使用例	意味
加算	add	add t0,a0,a1	t0 = a0 + a1
減算	sub	sub t0,a0,a1	t0 = a0 - a1
AND	and	and t0,a0,a1	t0 = a0 && a1
OR	or	or t0,a0,a1	t0 = a0 || a1
比較	slt	slt t0,a0,a1	t0 = (a0 < a1)
算術 右シフト	sra	sra t0,a0,C	t0 = a0 >> C

※掛け算と割り算はR形式ではない（引数は2つだけ）
結果はLOとHIに分かれて専用のレジスタに入る

I形式の命令

op	rs	rt	rd	shamt	funct
6bit	5bit	5bit	5bit	5bit	6bit

- op: Operation Code(オペコード)
  - 命令語を表す：ただしR形式の場合は常に0
- rs: source operand 1
- rt: source operand 2
- rd: destination operand
- shamt: shift amount
  - シフト命令の時に利用する
- funct: function
  - 加減乗除などの指定はここで行う

具体的な命令の例（A形式）

- R形式の命令

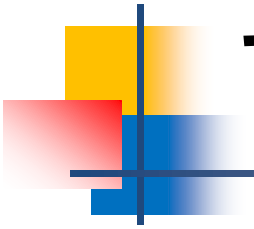
演算命令	命令コード	使用例	意味
加算	add	add t0,a0,a1	t0 = a0 + a1
減算	sub	sub t0,a0,a1	t0 = a0 - a1
AND	and	and t0,a0,a1	t0 = a0 && a1
OR	or	or t0,a0,a1	t0 = a0 || a1
比較	slt	slt t0,a0,a1	t0 = (a0 < a1)
算術 右シフト	sra	sra t0,a0,C	t0 = a0 >> C

※掛け算と割り算はR形式ではない（引数は2つだけ）
結果はLOとHIに分かれて専用のレジスタに入る

A形式の命令

op	rs	rt	rd	shamt	funct
6bit	5bit	5bit	5bit	5bit	6bit

- op: Operation Code(オペコード)
  - 命令語を表す：ただしR形式の場合は常に0
- rs: source operand 1
- rt: source operand 2
- rd: destination operand
- shamt: shift amount
  - シフト命令の時に利用する
- funct: function
  - 加減乗除などの指定はここで行う

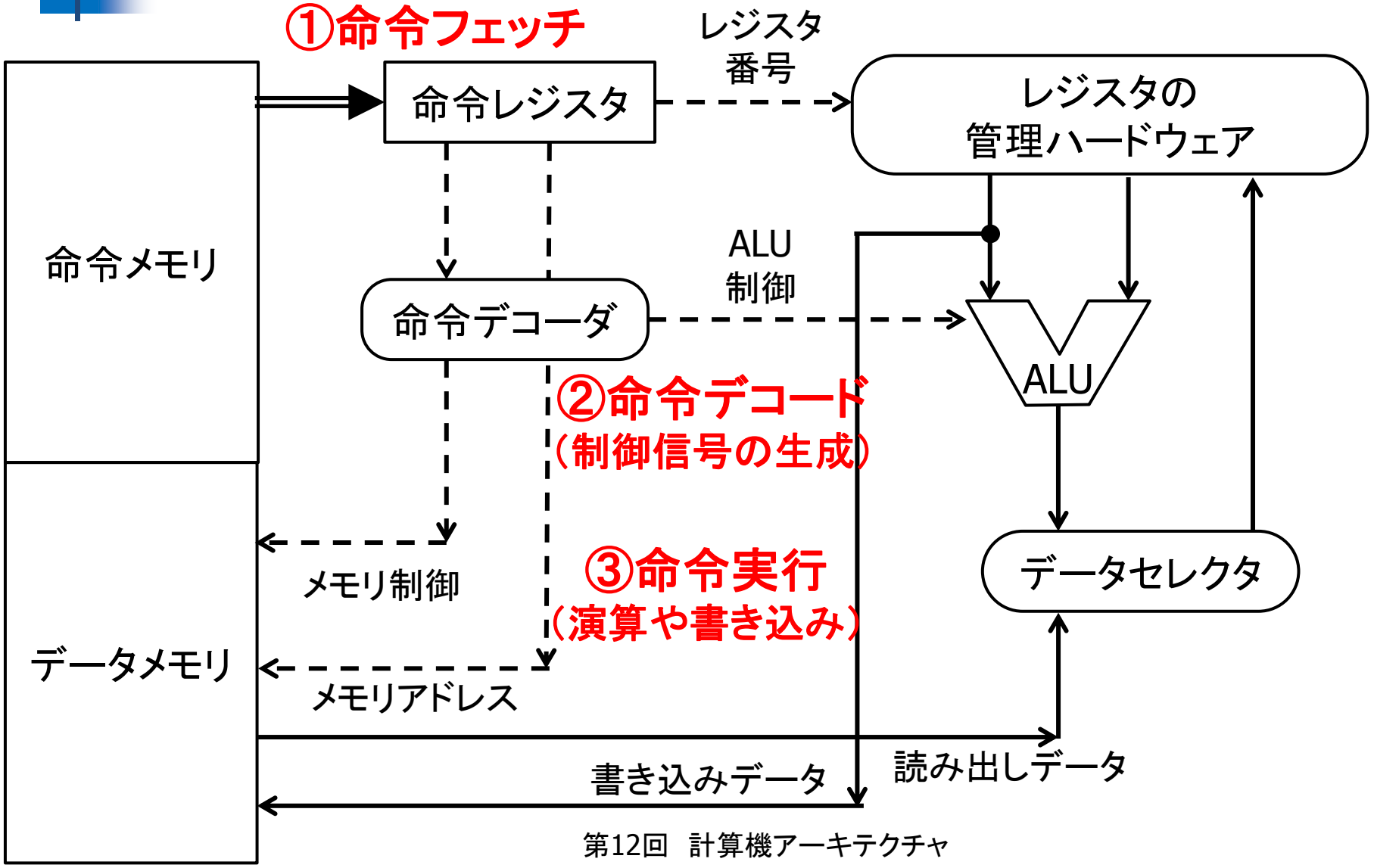


アドレッシングの問題点

命令実行の手順

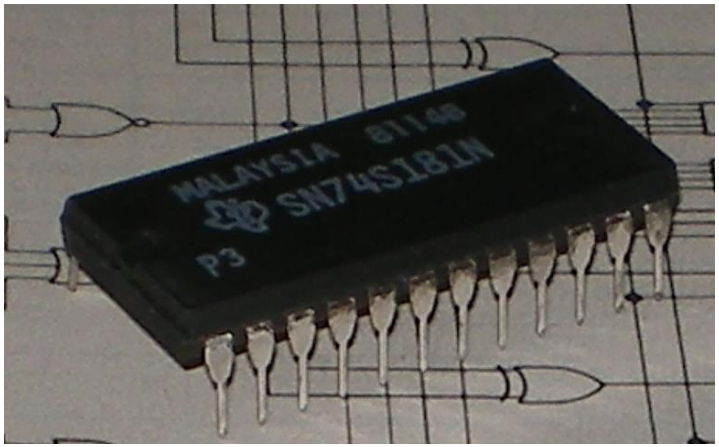
- 命令フェッチ（fetch）
  - 命令が実行される最初の過程
  - 命令メモリから命令レジスタにデータを読み込む
- 命令デコード（decode）
  - 命令を解釈し制御信号を作り出す
    - レジスタの操作（どのレジスタにアクセスするか）
    - メモリの操作（読み込みまたは書き込み）
    - データセレクタの操作（どの信号線を接続するか）
- 命令実行（execute）
  - ハードウェアの動作時間
  - メモリやレジスタに値が書き込まれる

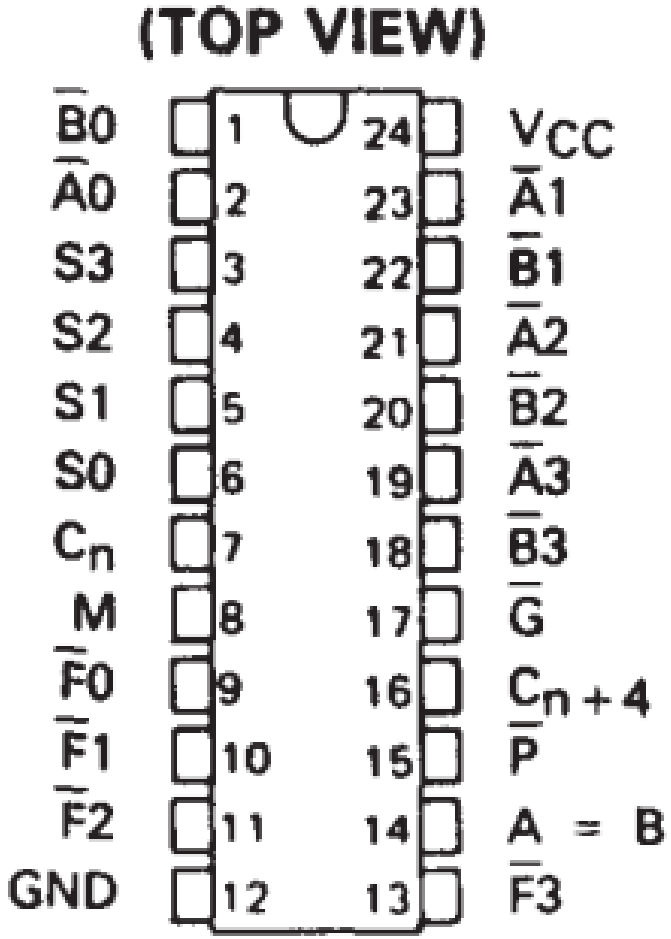
命令実行のしくみ


①命令フェッチ
レジスタ
番号
命令レジスタ
レジスタの
管理ハードウェア
命令メモリ
ALU
制御
命令デコーダ
ALU
②命令デコード
(制御信号の生成)
③命令実行
(演算や書き込み)
データセレクタ
メモリ制御
データメモリ
メモリアドレス
書き込みデータ
読み出しデータ

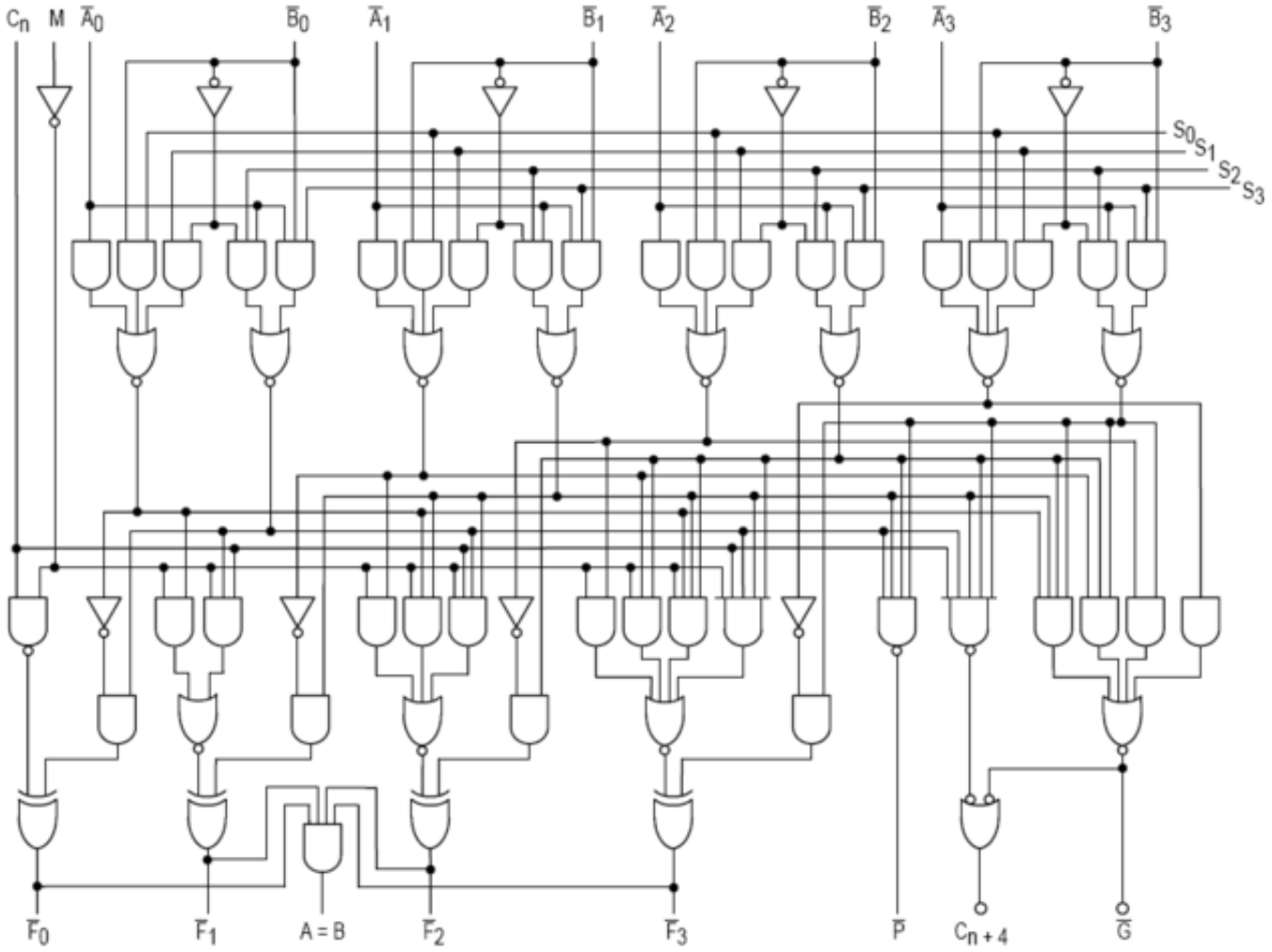
ALUと制御信号

- 現在のALUはCPUに内蔵されていますが・・・
  - 昔は外付けもありました
  - 教科書P.10参照

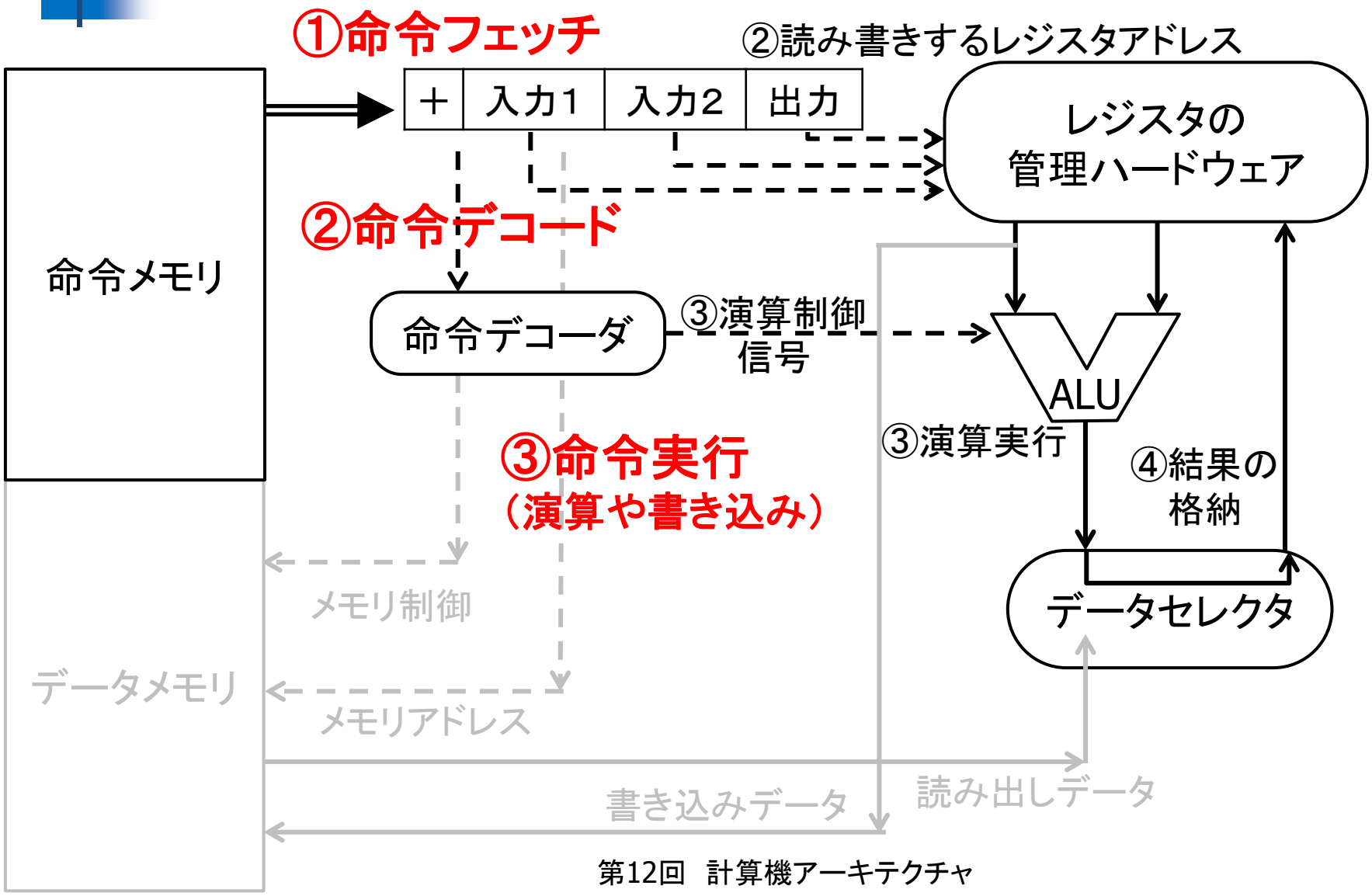




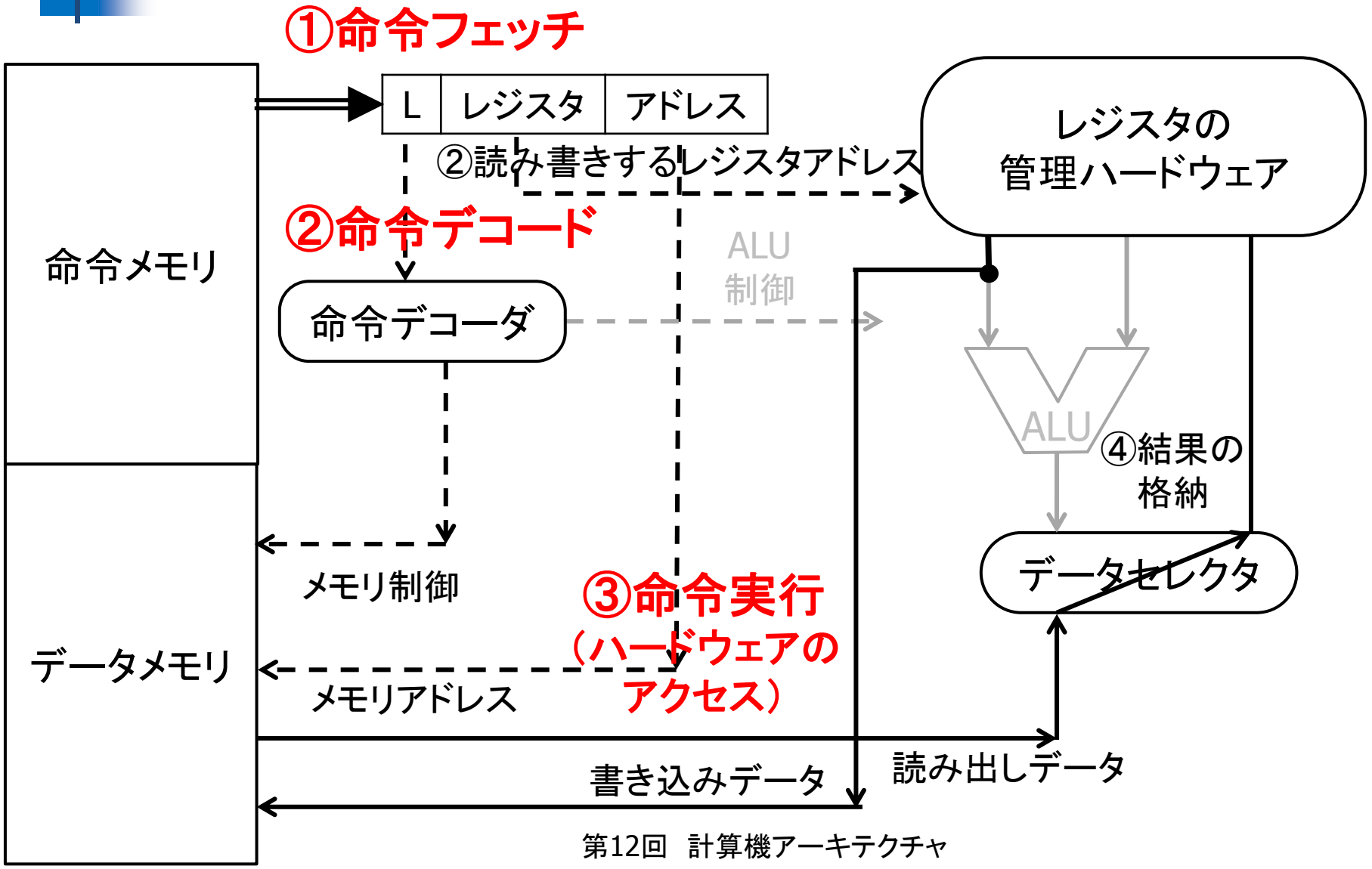
ALU(74181)の内部構造



算術論理演算命令の実行サイクル

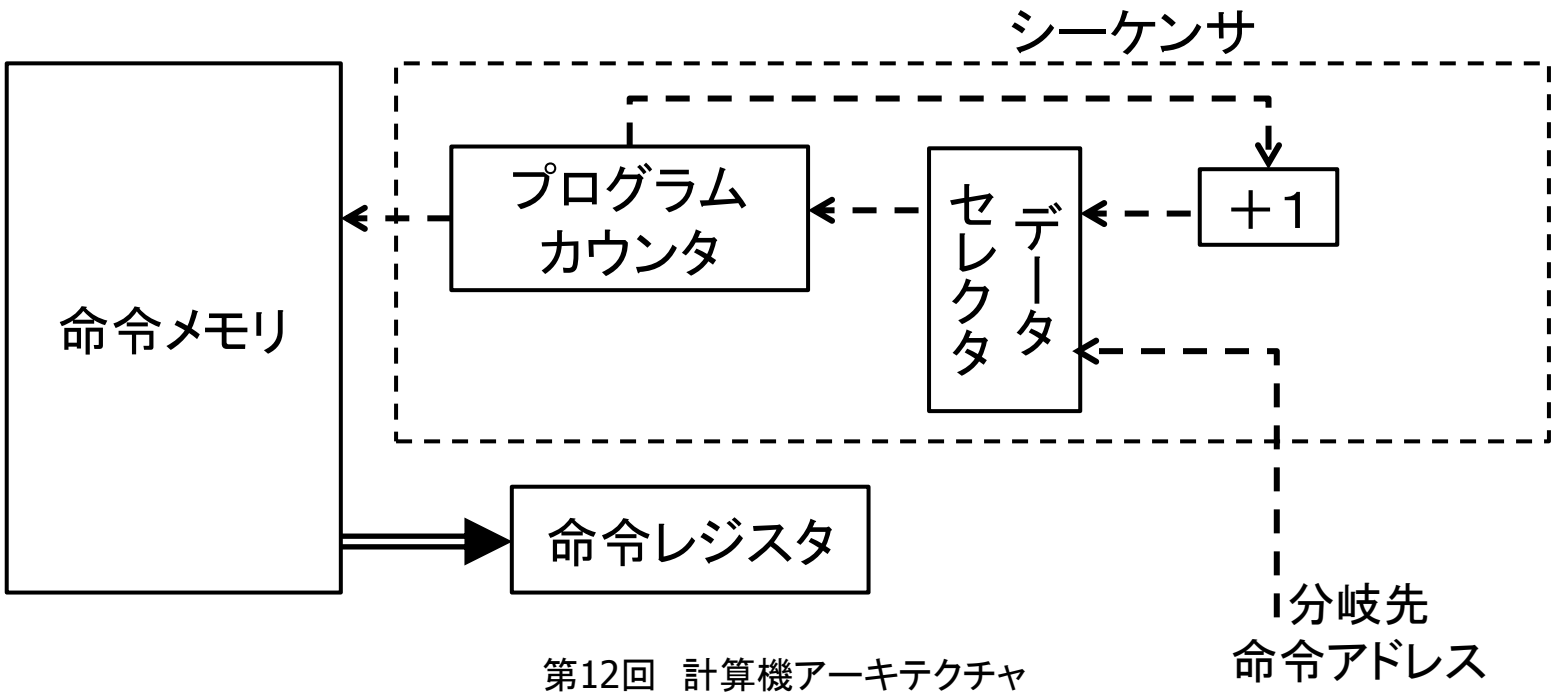


メモリ操作命令の実行サイクル

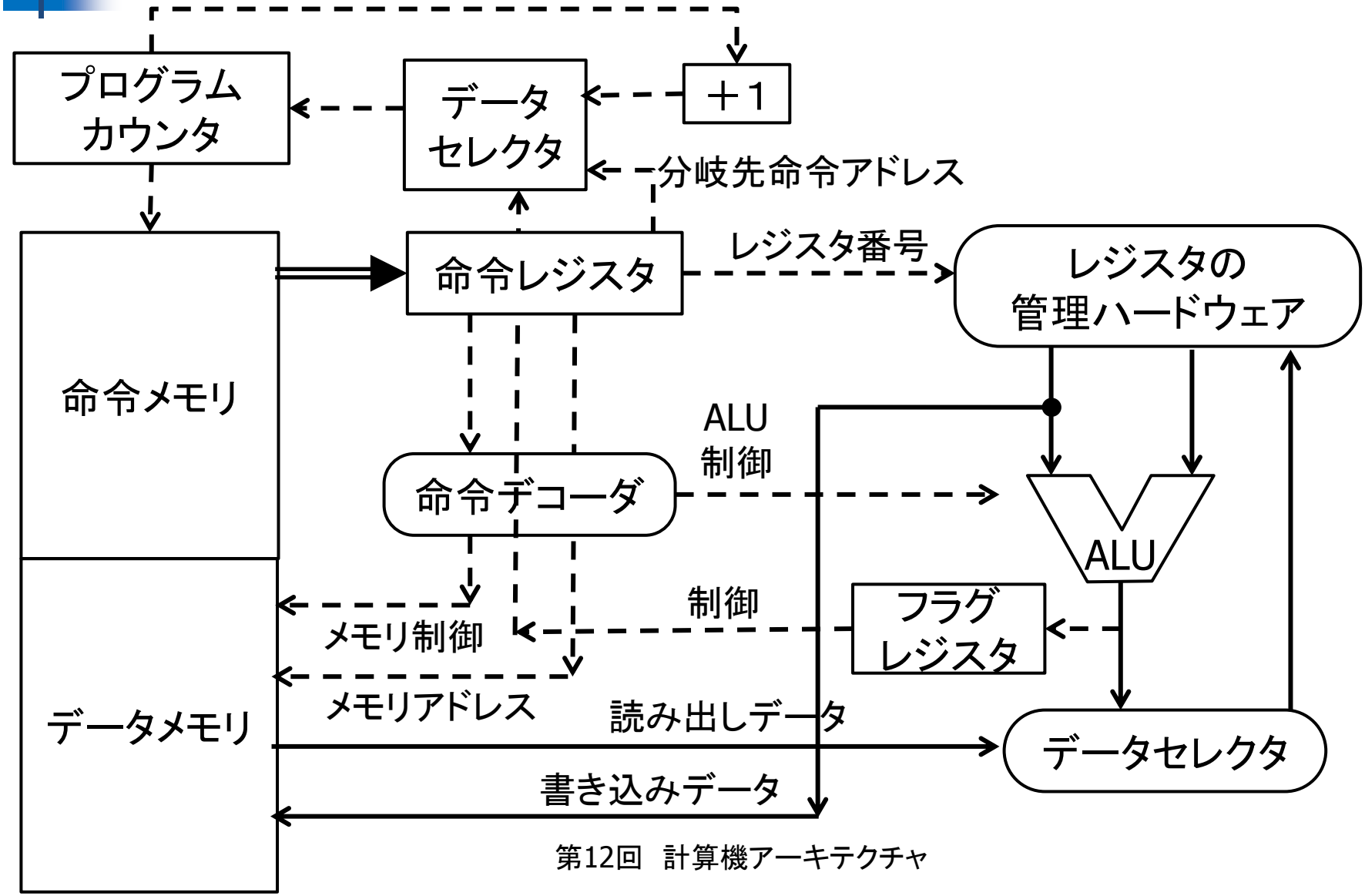


シーケンサ（sequencer）

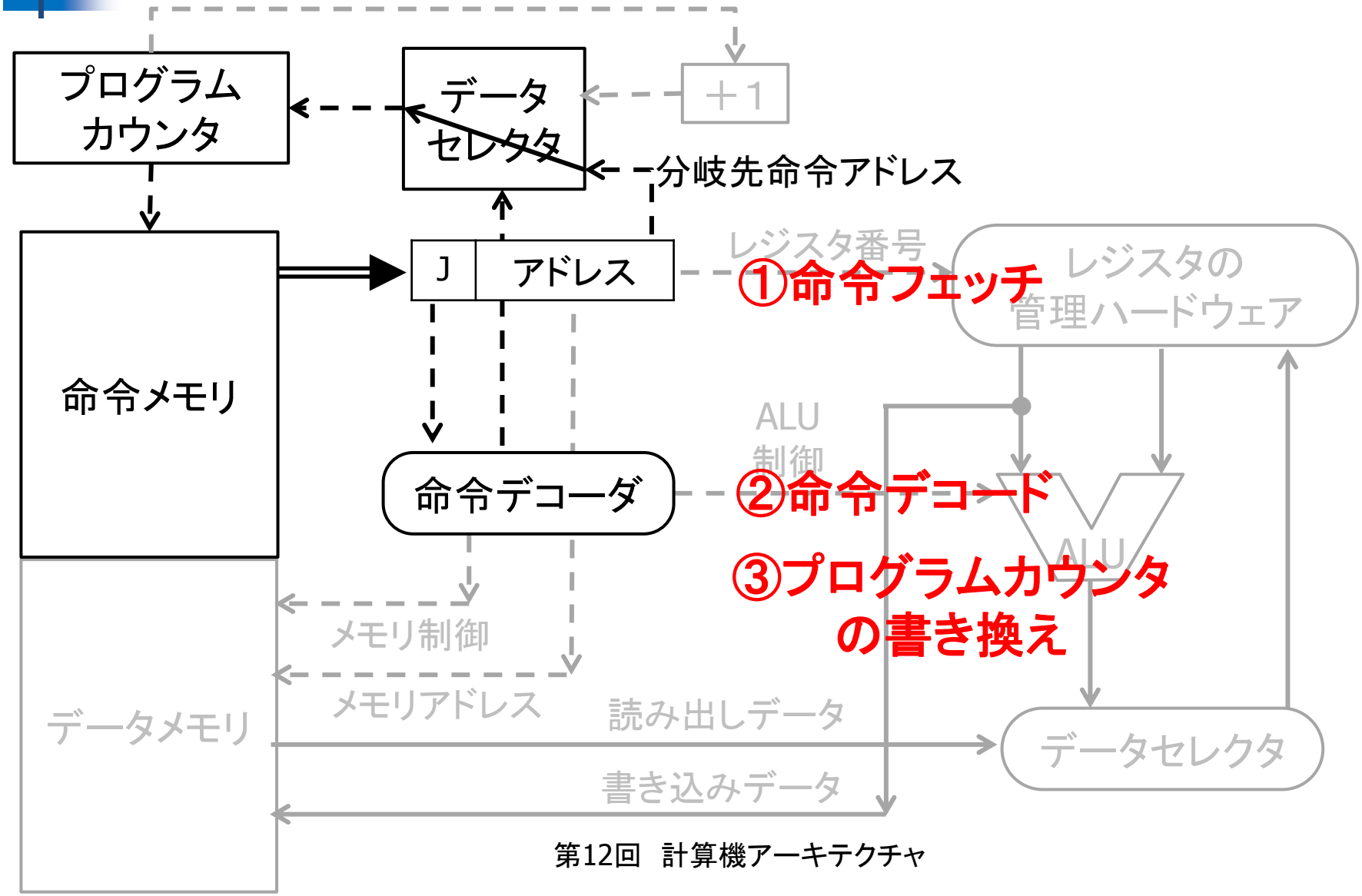
- 次に実行する命令を読み出すための回路
  - プログラムカウンタを制御するハードウェア
  - 命令メモリからプログラムを順に実行するための仕組み
  - 条件分岐や繰り返しの処理を行う際にも動作する



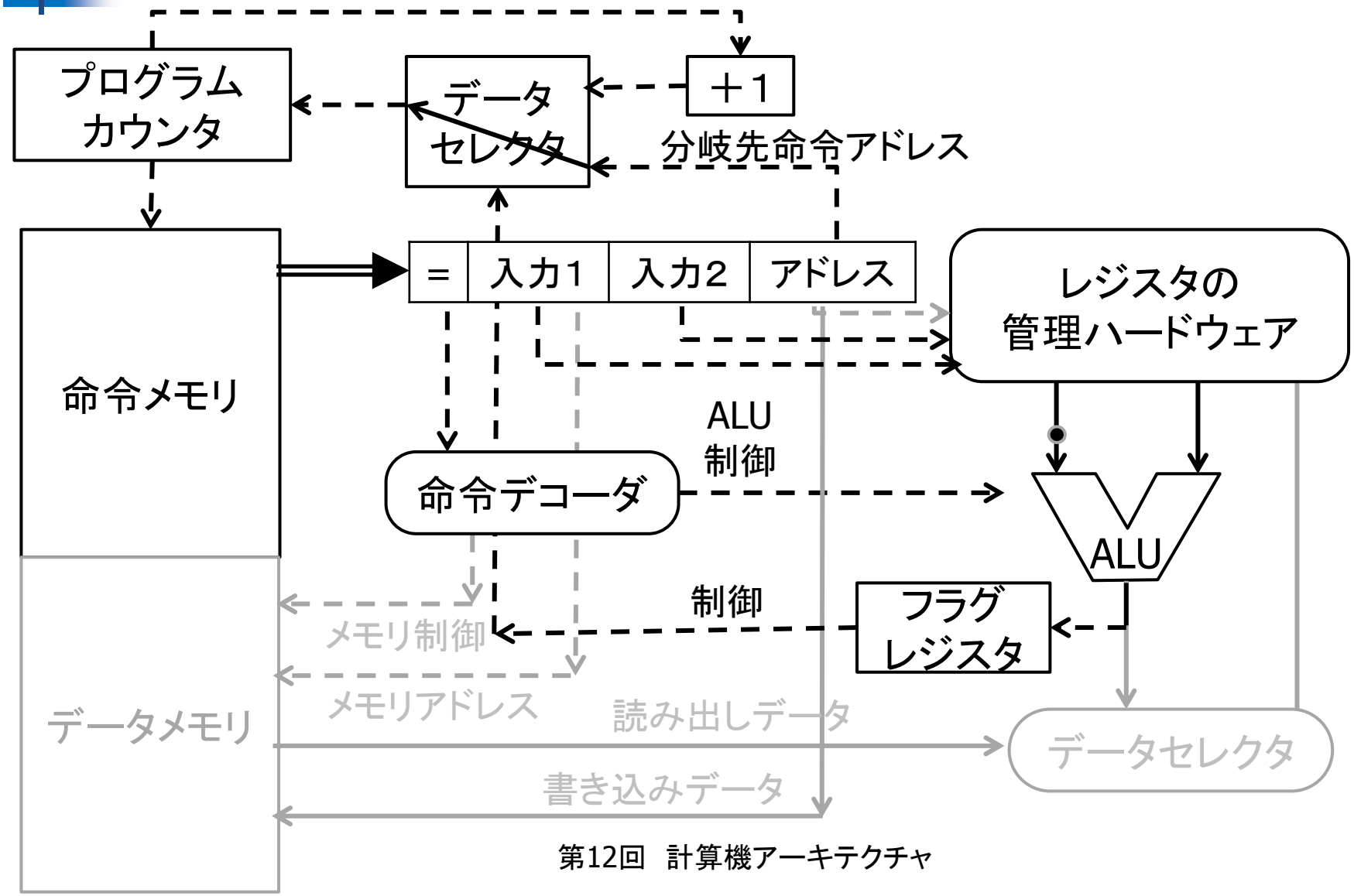
コンピュータの中枢部分の構成

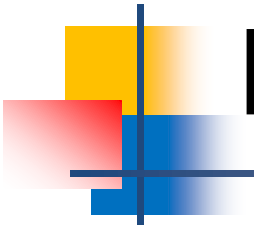


条件分岐命令の実行サイクル



フラグレジスタを使った条件分岐





MEMO

演習問題1

- 次のプログラムの各命令を分類せよ
  - 命令の種類にC,M,Jのいずれかを記入すること

ただし、C:算術論理演算、M:メモリ操作、J:条件分岐

番地	プログラムの内容	命令の種類
100	111番地の内容をレジスタ1へ読み込み	
101	112番地の内容をレジスタ2へ読み込み	
102	レジスタ1にレジスタ2の値を加算 (レジスタ1＝レジスタ1＋レジスタ2)	
103	レジスタ2から即値「1」を減算	
104	レジスタ2が1以上なら102番地へ処理を移す	
105	レジスタ1の値を113番地に書き込み	
106	114番地の値をレジスタ2へ読み込み	
107	レジスタ2からレジスタ1を減算	
108	レジスタ2が0以下なら110番地へ処理を移す	
109	113番地にレジスタ1の内容を書き込み	
110	プログラム終了	

演習問題2

- 各命令の実行時間を次の表に定めるとき、演習問題1のプログラムの実行時間を答えよ

処理内容	処理時間
命令のフェッチ	10秒
命令のデコード	3秒
算術論理演算命令の実行	5秒
メモリ操作命令の実行	10秒
条件分岐命令の実行	8秒

- ただし、データは次の表のとおりとする

メモリ番地	データ
111	5
112	2
113	−25684
114	7

命令機能の評価指数

早い計算機は何によって決まるか？

- 命令による実行時間の違い
  - 同じ種類の命令でも、加算と乗算で実行時間は違う
  - もちろん命令の種類によっても違う
- 計算機の動作周波数の違い
  - 計算機の動作周波数が高ければ高速にハードウェアが動作できる
- 評価指標：TPI（平均命令実行時間）
  - TPI ＝ TPC × CPI
  - TPC:マシンサイクル時間（クロック周波数の逆数）
  - CPI:平均命令実行サイクル数
    （1命令を何クロックで実行できるか）

演習問題3

- 同じ命令セットを持つコンピュータAとBがある
  このクロック周波数とCPIが表のようになっている

種類	クロック周波数	CPI
コンピュータA	4GHz	40
コンピュータB	1GHz	5

  - 問1：500命令のプログラムを実行した場合のCPU時間を計算せよ

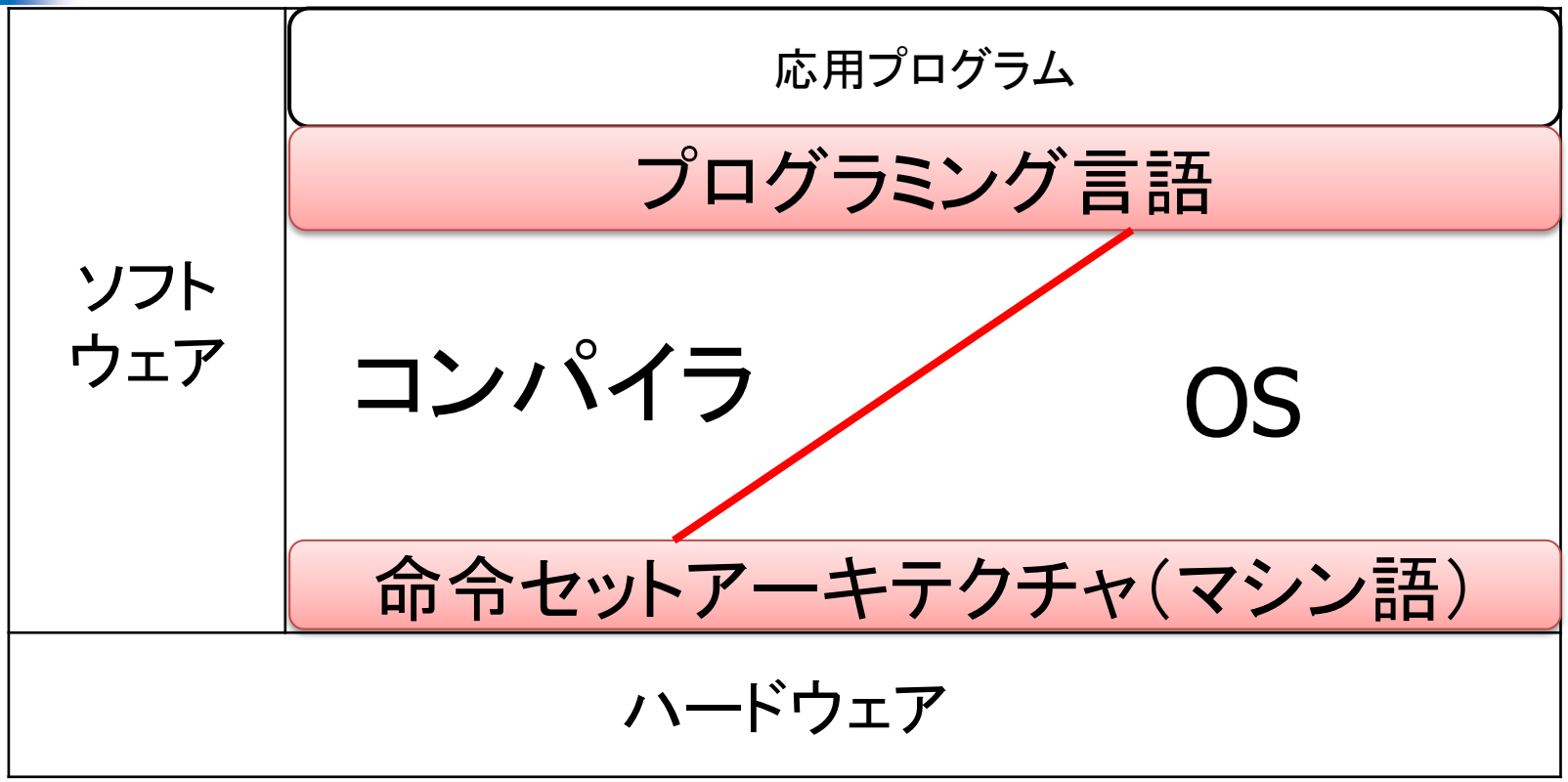
    ※ただし、CPU時間＝実行命令数×TPI
  - 問2：コンピュータA,Bではどちらが何倍速いといえるか

ハードウェアのトレードオフ論争

- 複雑で高度な機能は高速なハードウェアに任せたい
  - CISC（Complex Instruction Set Computer）
  - ソフトウェアの互換性が高くなる

- ハードウェアを単純化することで高速化し、複雑な命令はソフトウェアで対応したい
  - RISC（Reduced Instruction Set Computer）
  - CPIをできるかぎり小さくし、高速化が図れる
  - ハードウェアの小規模化、低価格化が図れる

ハードウェアとシステムプログラム



- 計算機アーキテクチャの対象範囲
  - 適切な命令セットアーキテクチャの設計
  - システムプログラム（コンパイラ、OS）とどう連携するか

プログラミング言語とコンパイラ

- 機械語は人間が記述するには最適とは言い難い
  - 機械語はハードウェアに合わせて作られた言語
  - ハードウェアが変わると、覚え直しが必要
- 人間にわかりやすい言語でプログラムを作りたい
  - プログラミング言語の開発
  - 機械語にまとめて変換しておいて実行

⇨ コンパイラ ⇨ コンパイル時に最適化処理が可能

  - プログラミング言語を解釈しながら実行

⇨ インタプリタ ⇨ ユーザがシステムと対話的にプログラムを作れる

システムプログラムとの連携

- 連携における検討事項
  - コンパイラとのインタフェース
    - データ型と内部表現形式
    - 命令形式
    - 命令機能（命令セット）
    - アドレス指定方法（アドレッシング）

    } 来週議論する予定
  - OSとのインタフェース
    - プロセス管理
    - メモリ管理
    - 入出力処理
    - 割り込み、例外処理