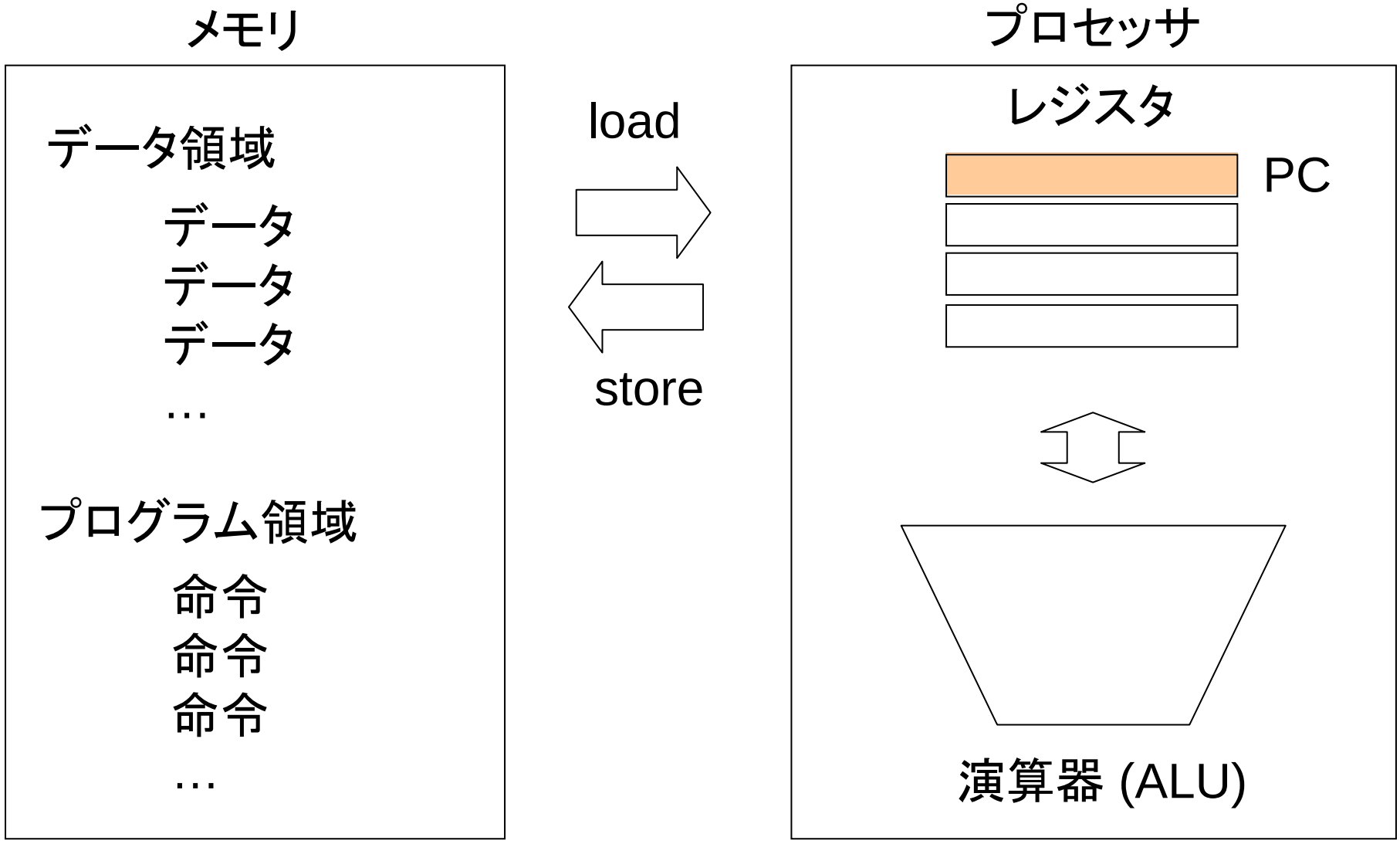
計算機の構造とプログラムの実行

計算機の基本構成


メモリ
データ領域
データ
データ
データ
…
プログラム領域
命令
命令
命令
…
load
store
プロセッサ
レジスタ
PC
演算器 (ALU)

計算機の基本動作

- プロセッサは，メモリのプログラム領域から**命令**をアドレス順に読み出して**実行**する
- 演算は **ALU** (Arithmetic Logic Unit) が行う
- 必要に応じて，メモリとプロセッサ内の**レジスタ**との間でデータを移動する
  - 演算を行うには，少なくとも瞬時にはプロセッサ内でデータを記憶しておく必要がある
  - 最近のほとんどのプロセッサは，メモリ内のデータではなくレジスタ内のデータを演算の対象とする (∵ メモリはプロセッサに対して遅いため)
  - **load**: メモリ → レジスタ
  - **store**: メモリ ← レジスタ
- **PC** (Program Counter) と呼ばれる特殊レジスタに，次に実行する命令のアドレスが保存されている
  - PCの内容は命令実行ごとに更新される

命令セットアーキテクチャ

- プロセッサが実行できる命令の集合を**命令セット** (instruction set) と呼ぶ．実際には，プログラムから使用できるレジスタの種類，メモリアドレスの指定方法なども含めて命令セットと呼ぶのが通常である
- ソフトウェアから見たときに，そのプロセッサがどんなものであるかは，命令セットによって決まる．この観点から見たアーキテクチャを**命令セットアーキテクチャ** (Instruction Set Architecture: ISA) と呼ぶ
- それに対し，ある命令セットアーキテクチャをどのような回路でどのような動作タイミングで実現するかという観点から見たアーキテクチャをマイクロアーキテクチャと呼ぶ
  - 同じ ISA に対して多数のマイクロアーキテクチャがあり得る

命令セットアーキテクチャの例

- x86 (IA-32, i386)
  いわゆる PC 用のCPUで採用．PC以外にも広く利用される．
- PowerPC
  以前の Macintosh．PlayStation 3，Xbox 360，Nintendo Wii
- SPARC
  Sun Microsystems のワークステーション，各種組み込み機器
- MIPS
  Silicon Graphics，Sony, NEC のワークステーション，
  初代 PlayStation，Nintendo 64，PSP, 各種組込み機器，携帯機器など
- ARM
  携帯機器・携帯電話の多く，ゲームボーイアドバンス，Nintendo DS, DSi
- SuperH (SH)
  各種組み込み機器，携帯機器，セガサターン，ドリームキャスト

注: 厳密な命令セットアーキテクチャ名としては，さらに細かく分類される
（例えば MIPS I, MIPS II, MIPS32, MIPS64…）

歴史的な経緯

- 当初は，計算機の設計と具体的な製品は 1 対 1 対応
- IBM System/360 (1960) で，統一的なアーキテクチャによる「計算機ファミリ」の概念が現れる
- 初の商用マイクロプロセッサ Intel 4004 (1971) 以降，計算機本体とは独立の「部品」としてプロセッサを扱えるようになる（計算機メーカとプロセッサメーカの分離）

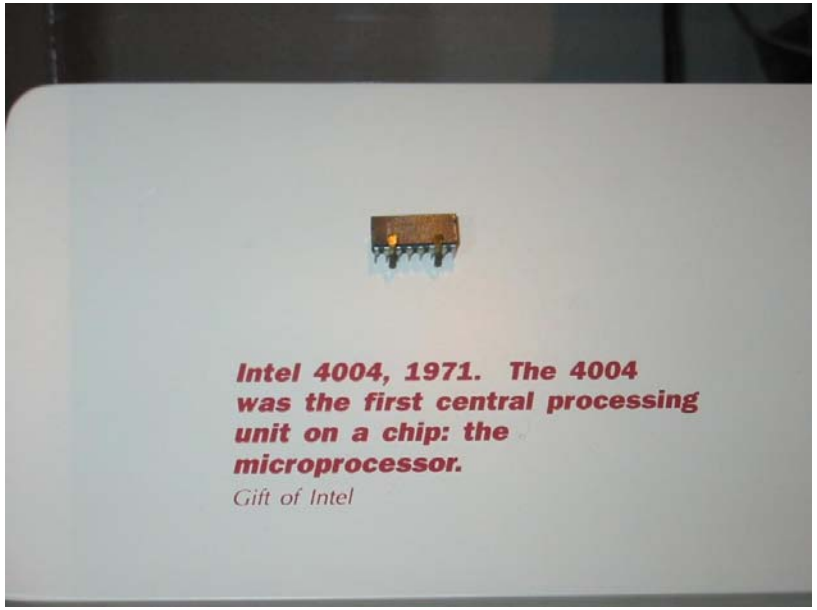
- 1980年代頃，RISCへの転回
  - **RISC** (Reduced Instruction Set Computer):
    命令セットを簡素化し，回路を単純化することで高速化
  - **CISC** (Complex Instruction Set Computer):
    RISC に対して従来のアーキテクチャをこう呼んだ

IBM System/360

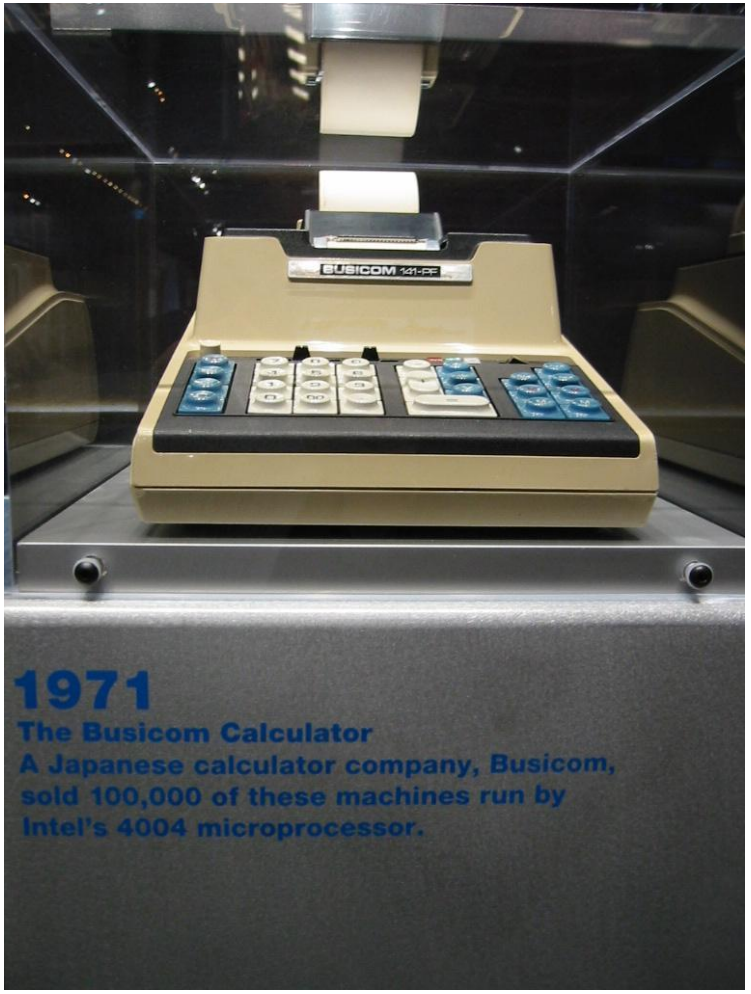
http://upload.wikimedia.org/wikipedia/commons/8/8d/Bundesarchiv_B_145_Bild-F038812-0014%2C_Wolfsburg%2C_VW_Autowerk.jpg

「コンピュータアーキテクチャ」という概念をおそらく最初に明確に導入した商用計算機．オペレーティングシステム(OS)を最初に導入した商用計算機でもある

intel 4004

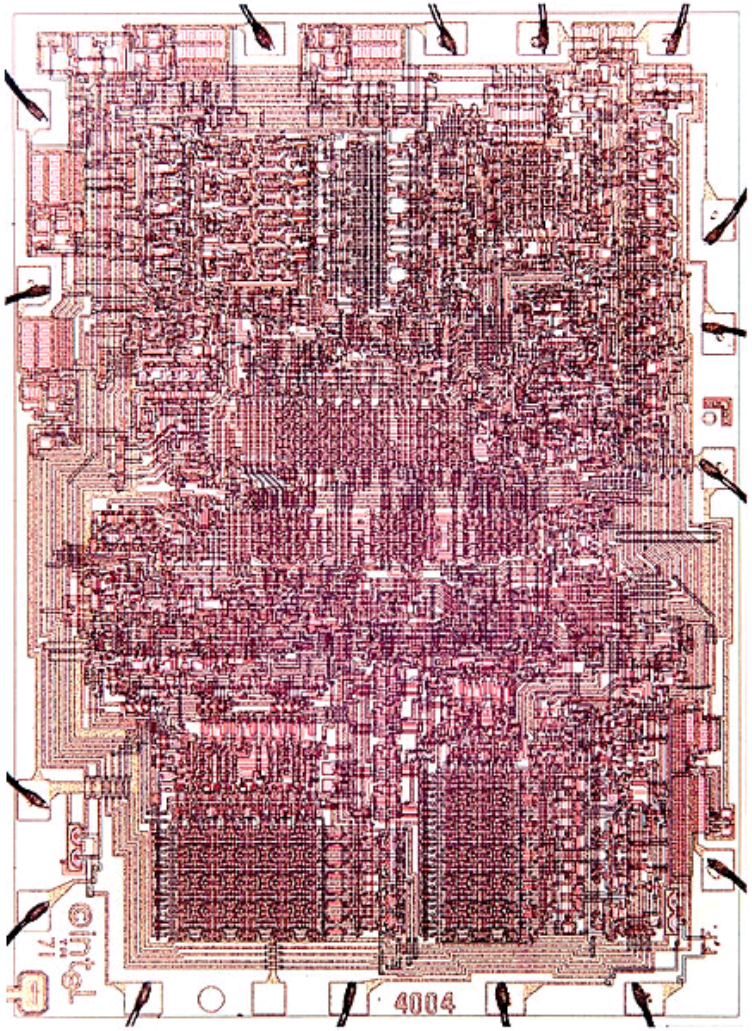


(American History Museum)



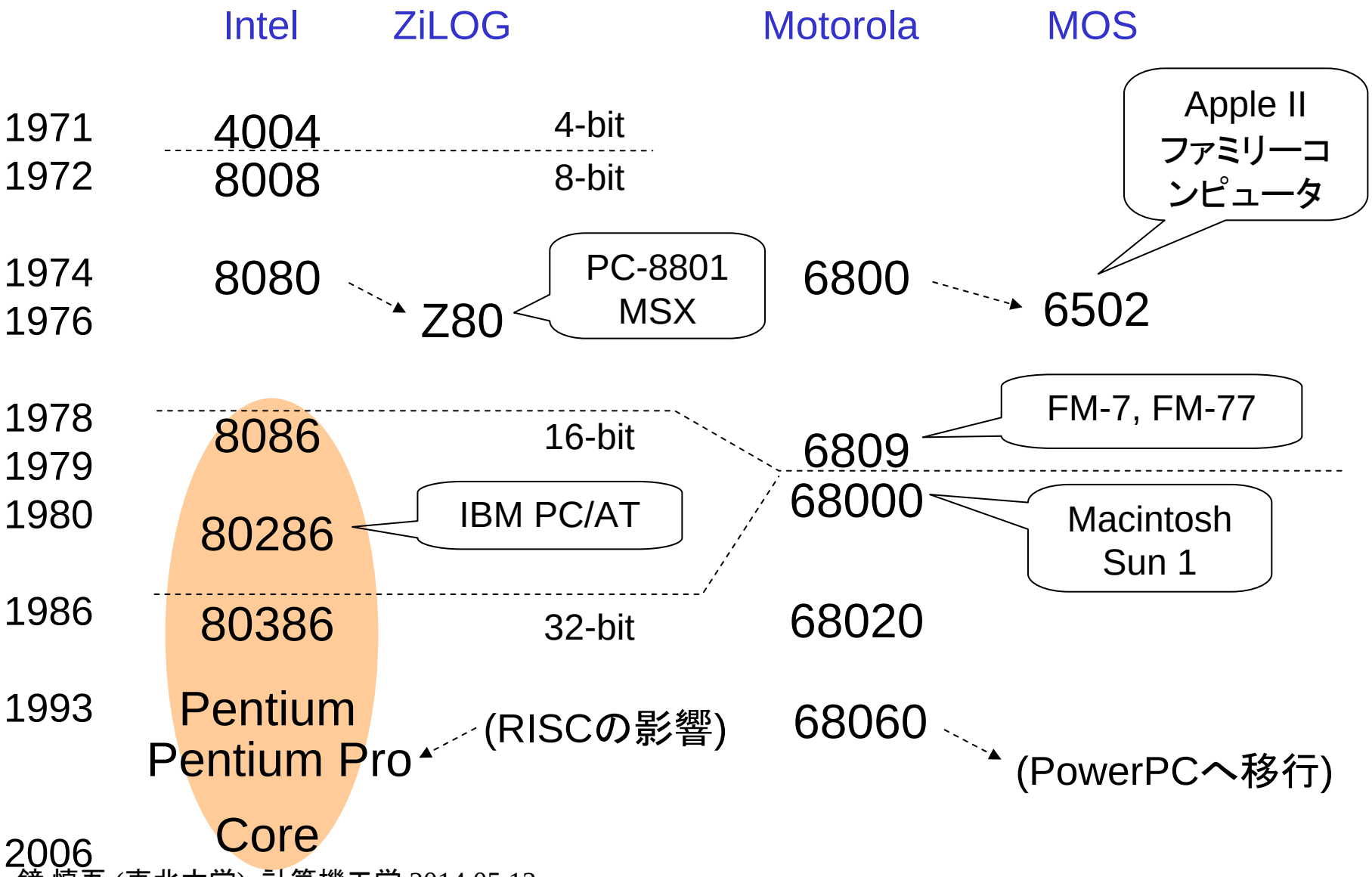
Busicom 141-PF
(Intel Museum, Santa Clara)

intel 4004

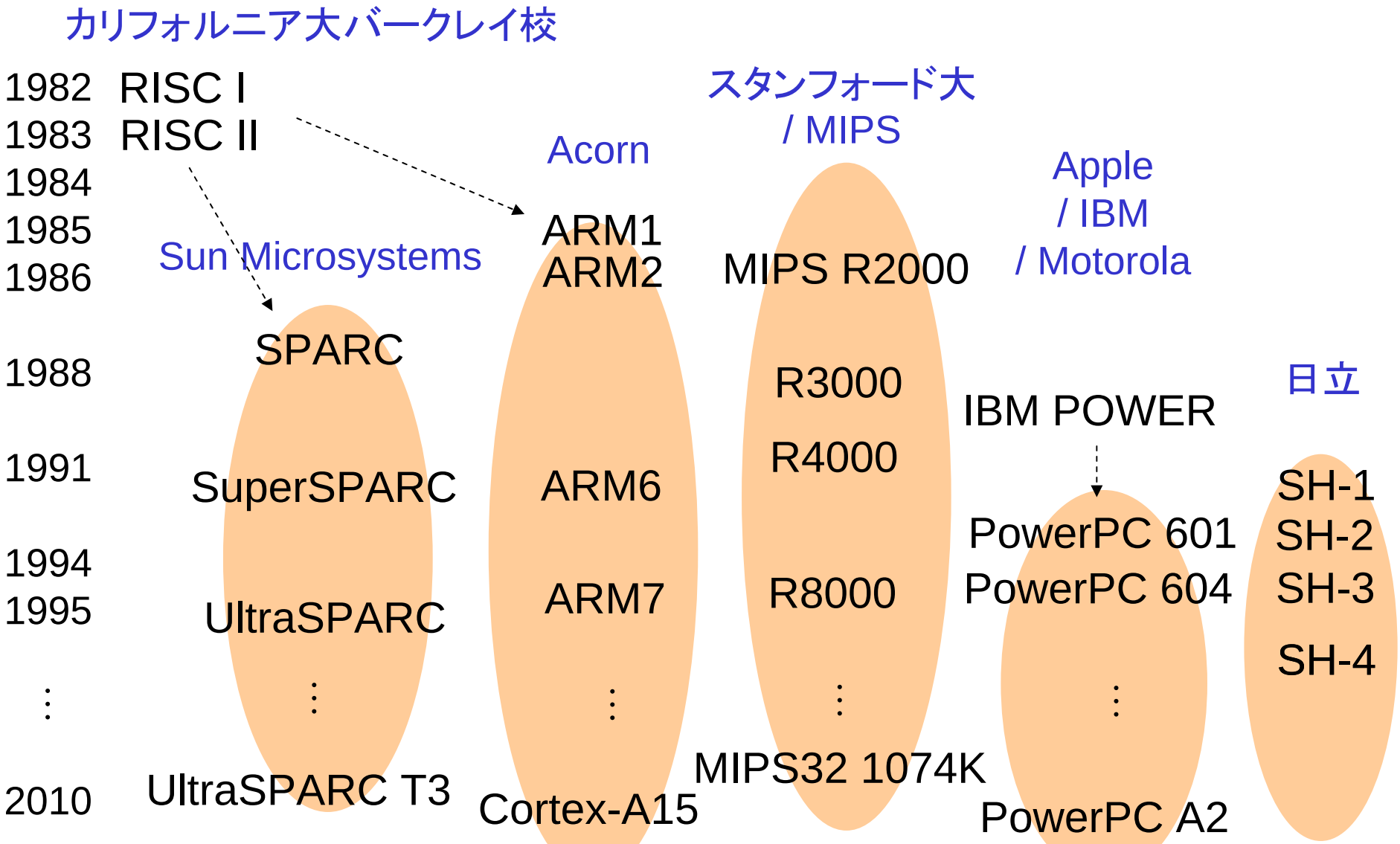


http://news.com.com/1971+Intel+4004+processor/2009-1006_3-6038974-3.html

マイクロプロセッサの系譜 (CISC)



マイクロプロセッサの系譜 (RISC)



ゲーム機用プロセッサ

任天堂 ファミリーコンピュータ (1983), NEC PCエンジン (1987): 6502
セガ マークIII (1985): Z80
セガ メガドライブ (1988): 68000 + Z80
任天堂 スーパーファミコン (1990): 65C816 (6502の後継)

セガサターン (1994): SH-2
ソニー PlayStation (1994): MIPS R3000
任天堂 NINTENDO64 (1996): MIPS R4300
セガ ドリームキャスト (1998): SH-4
ソニー PlayStation2 (2000): EmotionEngine (MIPS R5900ベース)
任天堂 ゲームキューブ (2001): PowerPC 750
マイクロソフト Xbox (2001): Mobile Celeron (Pentium IIIベース)

マイクロソフト Xbox 360 (2005): Xenon (PowerPCベース)
ソニー PlayStation3 (2006): Cell (PowerPCベース)
任天堂 Wii (2006): Broadway (PowerPCベース)
任天堂 Wii U (2010): Espresso (Powerベース)
ソニー PlayStation4 (2013): AMD Jaguar (x86ベース)

携帯電話・タブレット端末用プロセッサ

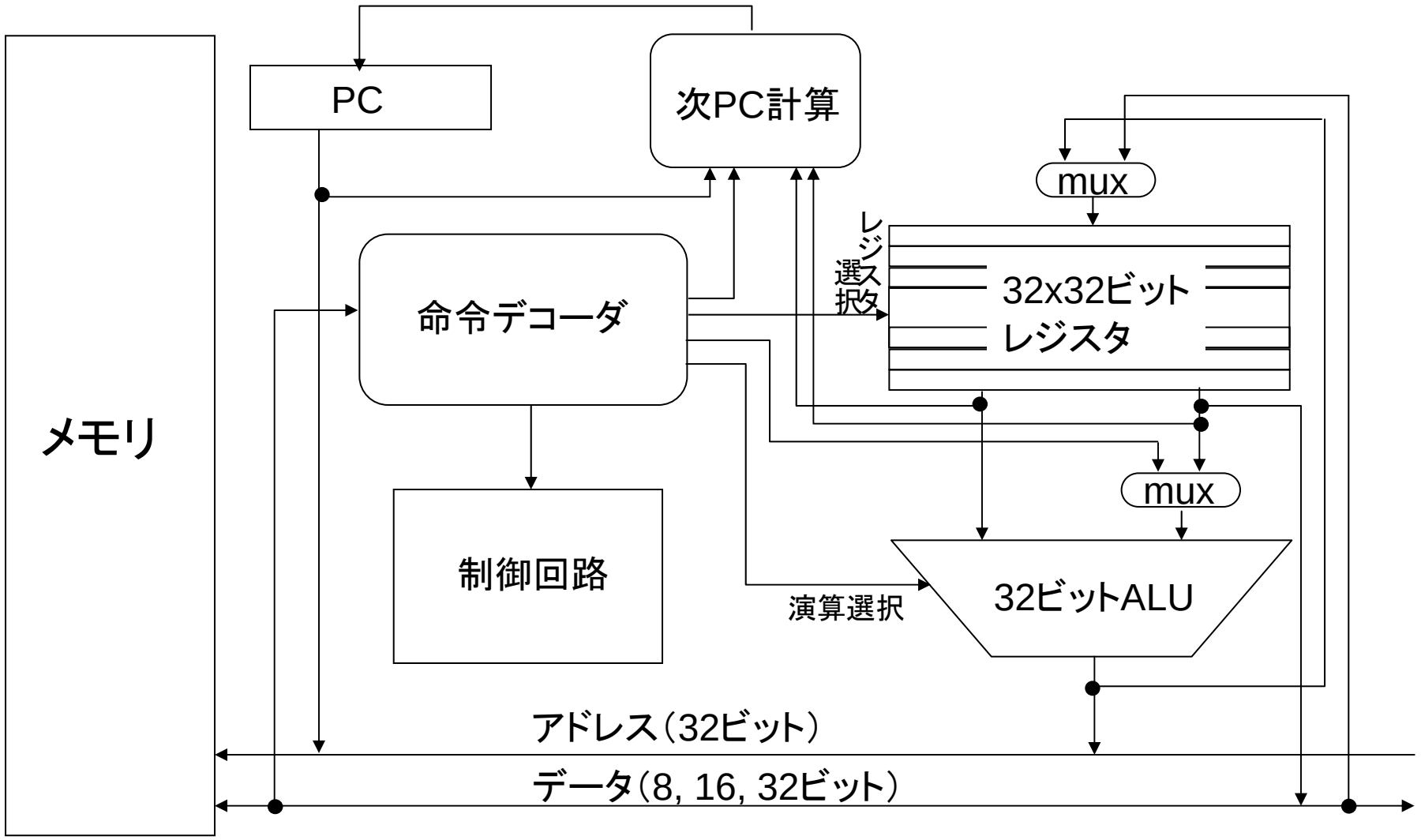
- Qualcomm SnapDragon (ARM)
- Texas Instruments OMAP (ARM)
- Samsung Exynos (ARM)
- NVIDIA Tegra (ARM)
- Apple A6 (ARM)
- Intel Atom (x86)
- ルネサス SH-Mobile (SuperH)

MIPSアーキテクチャ

- この講義では，MIPS I アーキテクチャを取り上げて計算機の動作を学ぶ
  - 現代的なアーキテクチャの基本形ともいえる構成
  - 組み込み機器を中心に，世界中で使われている
  - 世界中の大学の講義で取り上げられている

- 特徴
  - 32本 × 32ビット汎用レジスタ
  - 32ビットALU
  - 32ビットのメモリアドレス空間
  - PCは汎用レジスタとは別に存在（勝手にロード・ストアできない）

MIPSの構造



mux は選択回路

(参考) MIPSシミュレータ SPIM

参考書 (パターソン・ヘネシー) でも紹介されているシミュレータ SPIM を使うと，MIPSの動作を確認することができる.

http://spimsimulator.sourceforge.net/

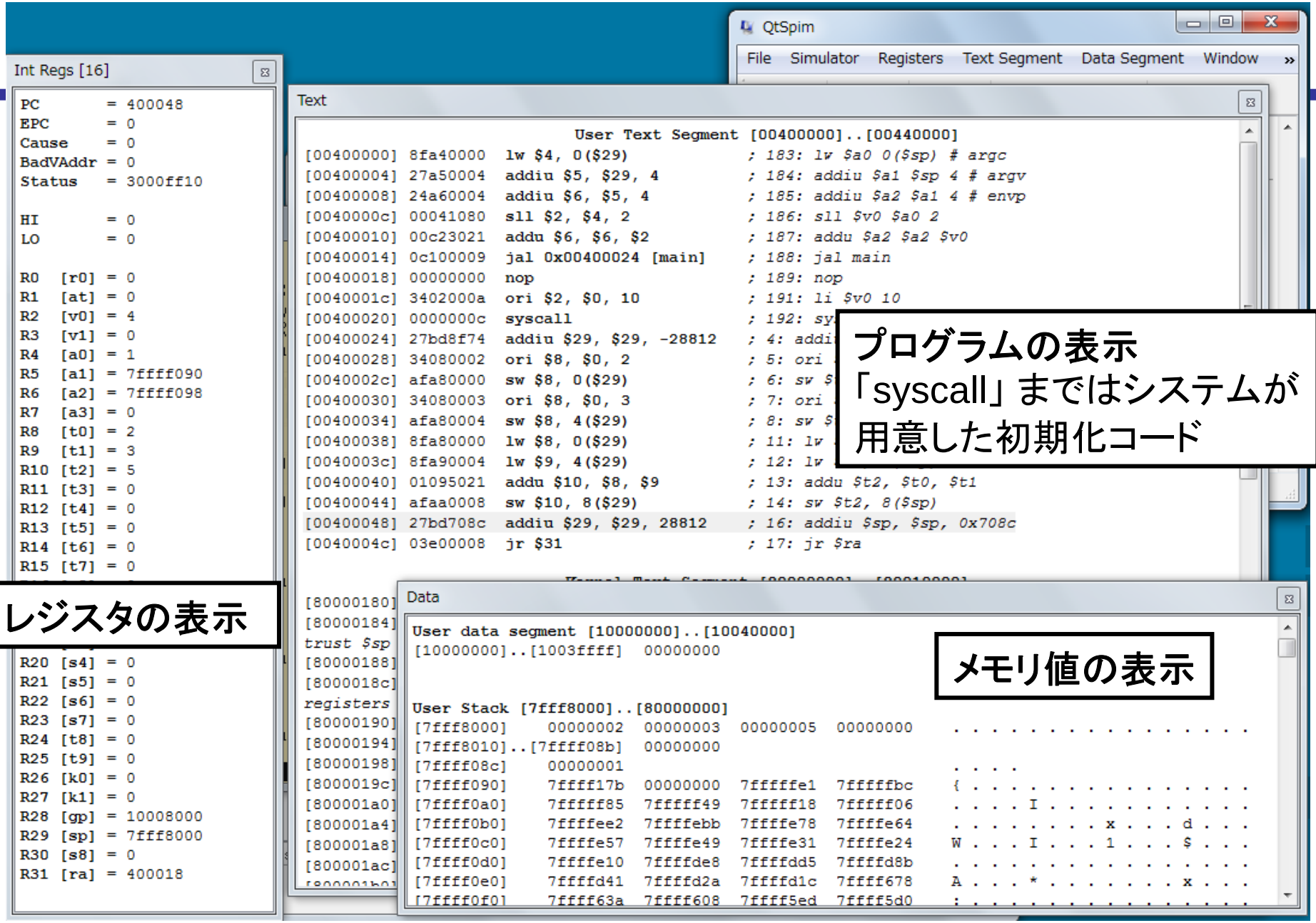
- UNIX, MacOS, Windows で動作

最低限の動かし方:

- File →Reinitialize and Load File でアセンブリ言語ファイルを開く
- Simulator → Run/Continue (F5) で実行
- あるいは Simulator → Single Step (F10) で1行ずつ実行

講義に対応したサンプルプログラム:

- http://www.ic.is.tohoku.ac.jp/~swk/lecture/


QtSpim
File Simulator Registers Text Segment Data Segment Window
Int Regs [16]
PC = 400048
EPC = 0
Cause = 0
BadVAddr = 0
Status = 3000ff10
HI = 0
LO = 0
R0 [r0] = 0
R1 [at] = 0
R2 [v0] = 4
R3 [v1] = 0
R4 [a0] = 1
R5 [a1] = 7ffff090
R6 [a2] = 7ffff098
R7 [a3] = 0
R8 [t0] = 2
R9 [t1] = 3
R10 [t2] = 5
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
R28 [gp] = 10008000
R29 [sp] = 7fff8000
R30 [s8] = 0
R31 [ra] = 400018
Text
User Text Segment [00400000]..[00440000]
[00400000] 8fa40000 lw $4, 0($29) ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004 addiu $5, $29, 4 ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004 addiu $6, $5, 4 ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080 sll $2, $4, 2 ; 186: sll $v0 $a0 2
[00400010] 00c23021 addu $6, $6, $2 ; 187: addu $a2 $a2 $v0
[00400014] 0c100009 jal 0x00400024 [main] ; 188: jal main
[00400018] 00000000 nop ; 189: nop
[0040001c] 3402000a ori $2, $0, 10 ; 191: li $v0 10
[00400020] 0000000c syscall ; 192: sy
[00400024] 27bd8f74 addiu $29, $29, -28812 ; 4: addi
[00400028] 34080002 ori $8, $0, 2 ; 5: ori
[0040002c] afa80000 sw $8, 0($29) ; 6: sw $
[00400030] 34080003 ori $8, $0, 3 ; 7: ori
[00400034] afa80004 sw $8, 4($29) ; 8: sw $
[00400038] 8fa80000 lw $8, 0($29) ; 11: lw
[0040003c] 8fa90004 lw $9, 4($29) ; 12: lw
[00400040] 01095021 addu $10, $8, $9 ; 13: addu $t2, $t0, $t1
[00400044] afaa0008 sw $10, 8($29) ; 14: sw $t2, 8($sp)
[00400048] 27bd708c addiu $29, $29, 28812 ; 16: addiu $sp, $sp, 0x708c
[0040004c] 03e00008 jr $31 ; 17: jr $ra
[80000180]
[80000184]
trust $sp
[80000188]
[8000018c]
registers
[80000190]
[80000194]
[80000198]
[8000019c]
[800001a0]
[800001a4]
[800001a8]
[800001ac]
Data
User data segment [10000000]..[10040000]
[10000000]..[1003ffff] 00000000
User Stack [7fff8000]..[80000000]
[7fff8000] 00000002 00000003 00000005 00000000 . . . . . .
[7fff8010]..[7ffff08b] 00000000
[7ffff08c] 00000001 . . . .
[7ffff090] 7ffff17b 00000000 7fffffe1 7fffffbc { . . . . . .
[7ffff0a0] 7fffff85 7fffff49 7fffff18 7fffff06 . . . . I . . . . . .
[7ffff0b0] 7ffffee2 7ffffebb 7ffffe78 7ffffe64 . . . . . x . . . d . . .
[7ffff0c0] 7ffffe57 7ffffe49 7ffffe31 7ffffe24 W . . . I . . . 1 . . . $ . . .
[7ffff0d0] 7ffffe10 7ffffde8 7ffffdd5 7ffffd8b . . . . . .
[7ffff0e0] 7ffffd41 7ffffd2a 7ffffd1c 7ffff678 A . . . * . . . . . x . . .
[7ffff0f0] 7ffff63a 7ffff608 7ffff5ed 7ffff5d0 : . . . . . .
プログラムの表示
「syscall」まではシステムが用意した初期化コード
レジスタの表示
メモリ値の表示

SPIMに読み込ませるアセンブリ言語ファイルの例

```
        .text
        .globl main
main:
    addu $sp, $sp, -0x300
    or  $t0, $zero, 1
    sw  $t0, 0($sp)

###
    addu $t0, $sp, 4
    lw $t1, 0($sp)
    sll $t1, $t1, 2
    addu $t0, $t0, $t1
    or $t2, $zero, 300
    sw $t2, 0($t0)
###
    addu $sp, $sp, 0x300
    jr $ra
```

おまじない．自分のプログラムは main ラベルから始める．

レジスタやメモリ等の初期化．わからなくても気にしない．

講義中の説明で理解して欲しい部分．

main の終了．

レジスタ間の演算命令

(C言語)

```
c = a + b;
```

(疑似的な MIPSアセンブリ言語)

```
addu $c, $a, $b          # $c ← $a + $b
```

- ただし，変数 a, b, c の内容がそれぞれレジスタ a, b, c に置かれているとする (「$a」でレジスタaの値を表す)
- 以下，特に断らない限り変数は整数 (int) とする
- #以下は説明用のコメント．アセンブリ言語の一部ではない
- addu をオペコード，$c, $a, $b をオペランドと呼ぶ
- 特に$cを出力オペランド，$a, $bを入力オペランドと呼ぶ

例

(C言語)

```
e = (a + b) - (c + d);
```

ただし，変数 a～e の内容がそれぞれレジスタ a～e に置かれており，それ以外にレジスタ t が自由に使えるとする

(疑似的な MIPSアセンブリ言語)

```
addu $e, $a, $b      # $e ← $a + $b
addu $t, $c, $d      # $t ← $c + $d
subu $e, $e, $t      # $e ← $e - $t
```

- レジスタ t のように計算の都合上一時的に使われるレジスタを一時レジスタ (temporary register) と呼ぶ
- addu, subu の u は unsigned の略である．add 命令，sub 命令も計算内容は同じだが，オーバフローが起きたときに例外処理が行われる．C言語では通常オーバフローは無視する

資料: 主なレジスタ間演算命令

命令	説明
addu $c, $a, $b	$c ← $a + $b (add unsigned の略)
subu $c, $a, $b	$c ← $a – $b (subtract unsigned の略)
and $c, $a, $b	$c ← $a & $b
or $c, $a, $b	$c ← $a | $b
nor $c, $a, $b	$c ← ~($a | $b)
xor $c, $a, $b	$c ← $a ^ $b
sll $c, $a, $b	$c ← $a << $b (shift left logical の略)
srl $c, $a, $b	$c ← $a >> $b (shift right logical の略)
slt $c, $a, $b	符号つきで $a < $b ならば $c ← 1; さもなくば $c ← 0 (set on less than の略)
sltu $c, $a, $b	符号無しで $a < $b ならば $c ← 1; さもなくば $c ← 0 (set on less than unsigned)

MIPSのレジスタ

ここまでは便宜上，レジスタ名として変数名をそのまま用いて来たが，実際には MIPS には a, b, c などのような名前のレジスタは存在しない

実際には，32本のレジスタに 0 ～ 31 の番号がついており，その番号により指定する

```
addu $10, $8, $9      # $10 ← $8 + $9
```

- このままではわかりにくいので，次ページのような別名がついている
- 0番レジスタは，常に値0が読み出される特殊なレジスタである

資料: レジスター覧

番号表示	別名	説明
$0	$zero	常にゼロ
$1	$at	アセンブラ用に予約
$2, $3	$v0, $v1	関数からの戻り値用
$4 ～ $7	$a0 ～ $a3	関数への引数用
$8 ～ $15	$t0 ～ $t7	(主に)一時レジスタ
$16 ～ $23	$s0 ～ $s7	(主に)変数割り当て用
$24, $25	$t8, $t9	(主に)一時レジスタ
$26, $27	$k0, $k1	OS用に予約
$28	$gp	グローバルポインタ
$29	$sp	スタックポインタ
$30	$s8	(主に)変数割り当て用
$31	$ra	リターンアドレス

練習問題

1. レジスタ s0 の内容を s1 にコピーする命令を示せ．（ヒント: レジスタ $zero を活用する．一般に，同じ動作をする命令は一通りとは限らない）
   - この操作はよく使うので，マクロ命令 を用いて

     ```
     move $s1, $s0                    #  $s1 ← $s0
     ```

     と書いてよいことになっている．
2. レジスタ s0 の内容の各ビットを反転した結果を s1 に保存する命令を示せ．（ヒント: nor と $zero を活用する）
3. レジスタ s0 の値が 123，レジスタ s1 の値が 200 だったときに，以下の 3 命令からなるプログラムを実行した．実行完了後のレジスタ s0, s1 の値を答えよ．

   ```
   xor $s1, $s0, $s1
   xor $s0, $s0, $s1
   xor $s1, $s0, $s1
   ```

練習問題 解答例

1. 例えば以下の各命令
   or $s1, $s0, $zero
   or $s1, $zero, $s0
   addu $s1, $s0, $zero
   ほか多数

2. 例えば以下の命令
   nor $s1, $s0, $zero

3. s0, s1 を交換する処理になる. 各ビットについて s0 対 s1 が:
   0対0 なら s1 ← 0, s0 ← 0, s1 ← 0
   1対1 なら s1 ← 0, s0 ← 1, s1 ← 1
   0対1 なら s1 ← 1, s0 ← 1, s1 ← 0
   1対0 なら s1 ← 1, s0 ← 0, s1 ← 1

レジスタ-即値間の演算命令

(C言語)

```
y = x + 100;
```

ただし，変数 x, y の内容がそれぞれレジスタ s0, s1 に置かれているとする

(MIPSアセンブリ言語)

```
addu $s1, $s0, 100    # $s1 ← $s0 + 100
```

- 命令内で直接指定される定数を**即値**（immediate）と呼ぶ.
- 演算命令は，2つめの入力オペランドとして即値を取れる.
  - 1つめは必ずレジスタ．出力オペランドももちろんレジスタ

- 即値を取る addu 命令は，実際には addiu という命令として実行される（アセンブラが自動的に変換する．SPIMの実行画面にも注目）
- 即値を取る subu 命令は，実際には addiu 命令に符号反転した即値を渡すことで実行される.

例

(C言語)

```
y = x - 8;
z = 15;
```

ただし，変数 x, y, z の内容がそれぞれレジスタ s0, s1, s2 に置かれているとする

(MIPSアセンブリ言語)

```
subu $s1, $s0, 8          # $s1 ← $s0 - 8
or   $s2, $zero, 15       # $s2 ← 15
```

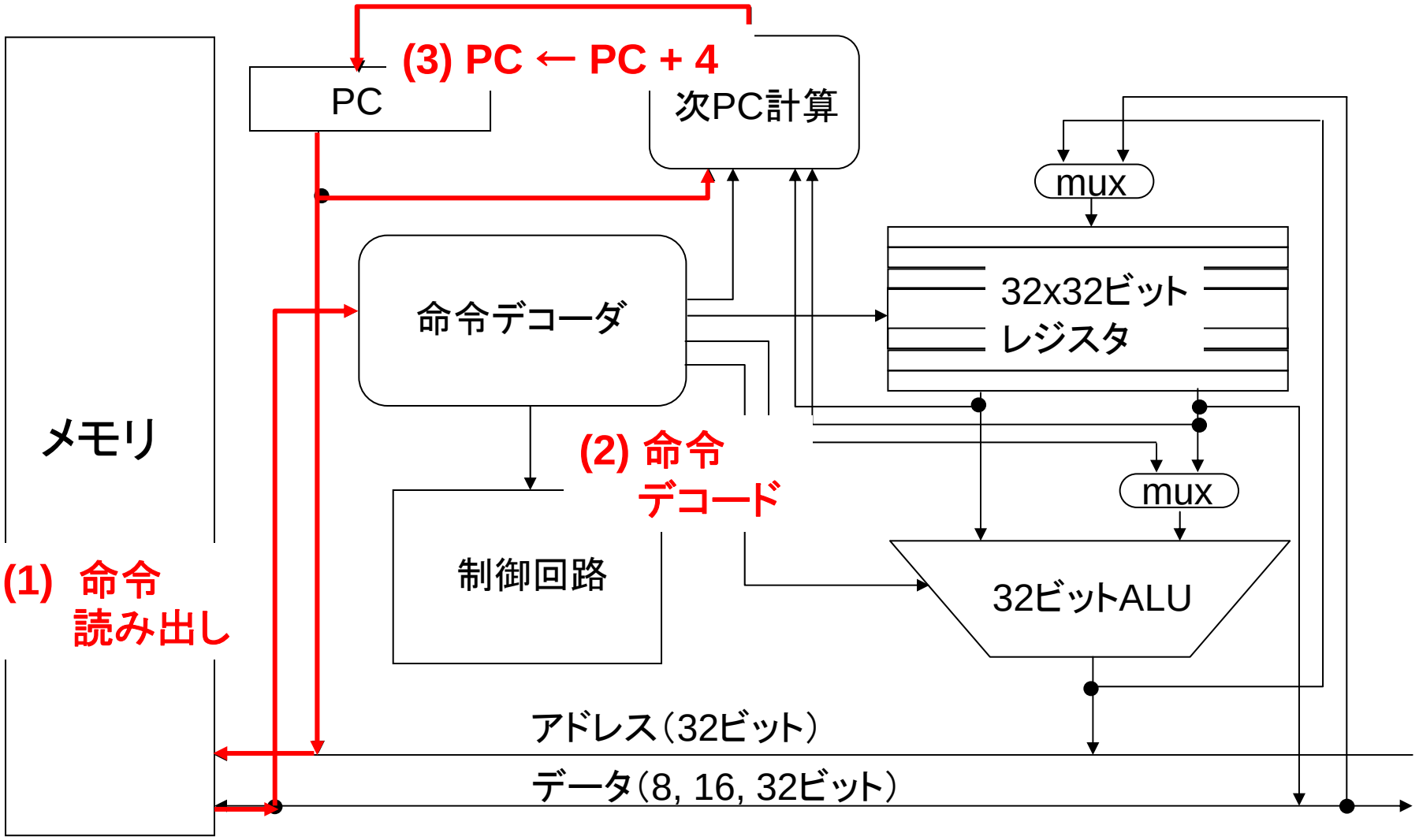
- 2行目のようにレジスタへの定数代入よく行われるので，li (load immediate) というマクロ命令でも書けるようになっている.

```
li   $s2, 15              # $s2 ← 15
```

資料: 主なマクロ命令

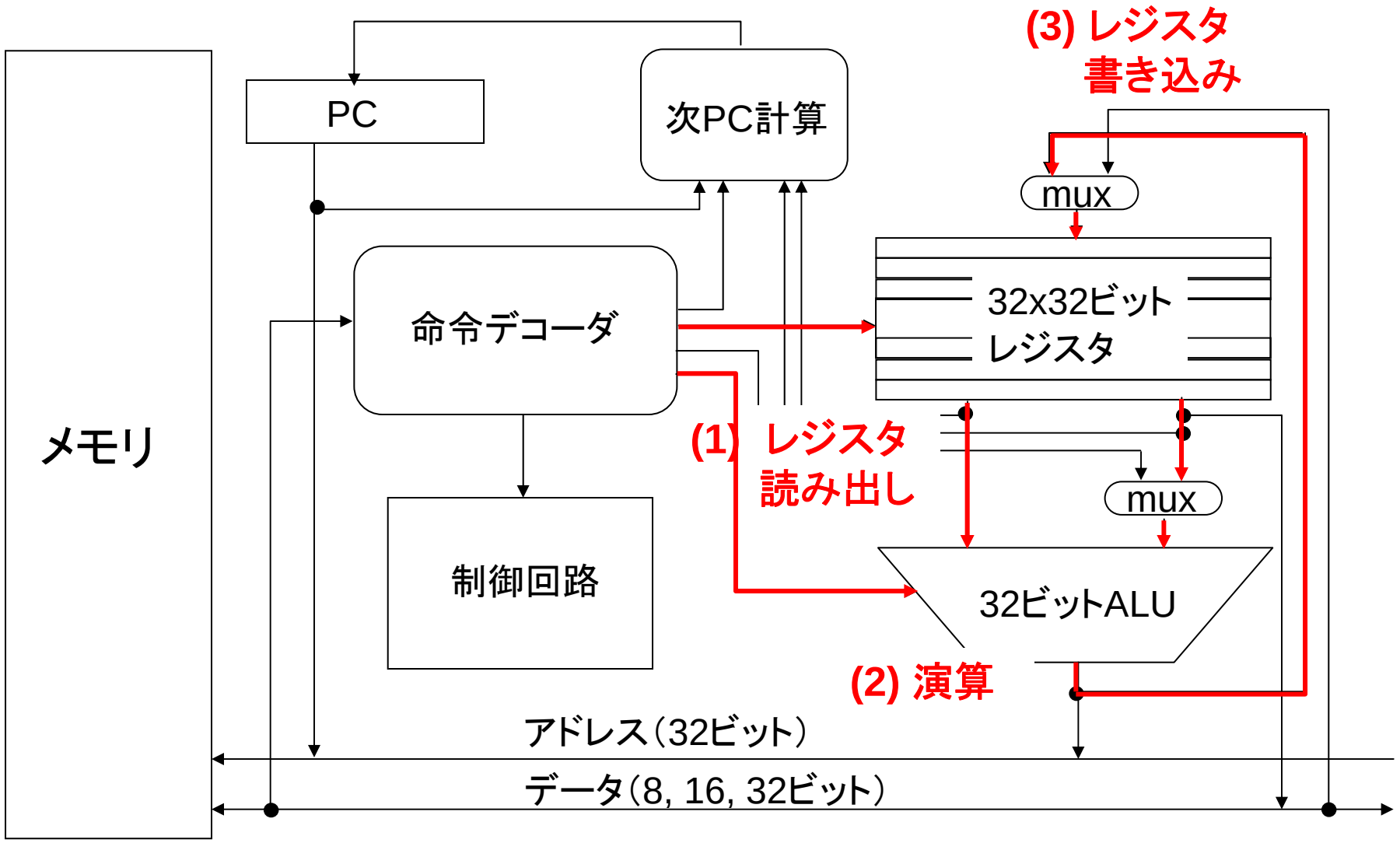
命令	説明
move $a, $b	$a ← $b　　# or $a, $zero, $b と等価
li $a, imm	$a ← imm　　# or $a, $zero, imm と等価 (load immediate)
nop	何もしない　　# sll, $zero, $zero, $zero と等価 (no operation)

命令フェッチと命令デコードの動作



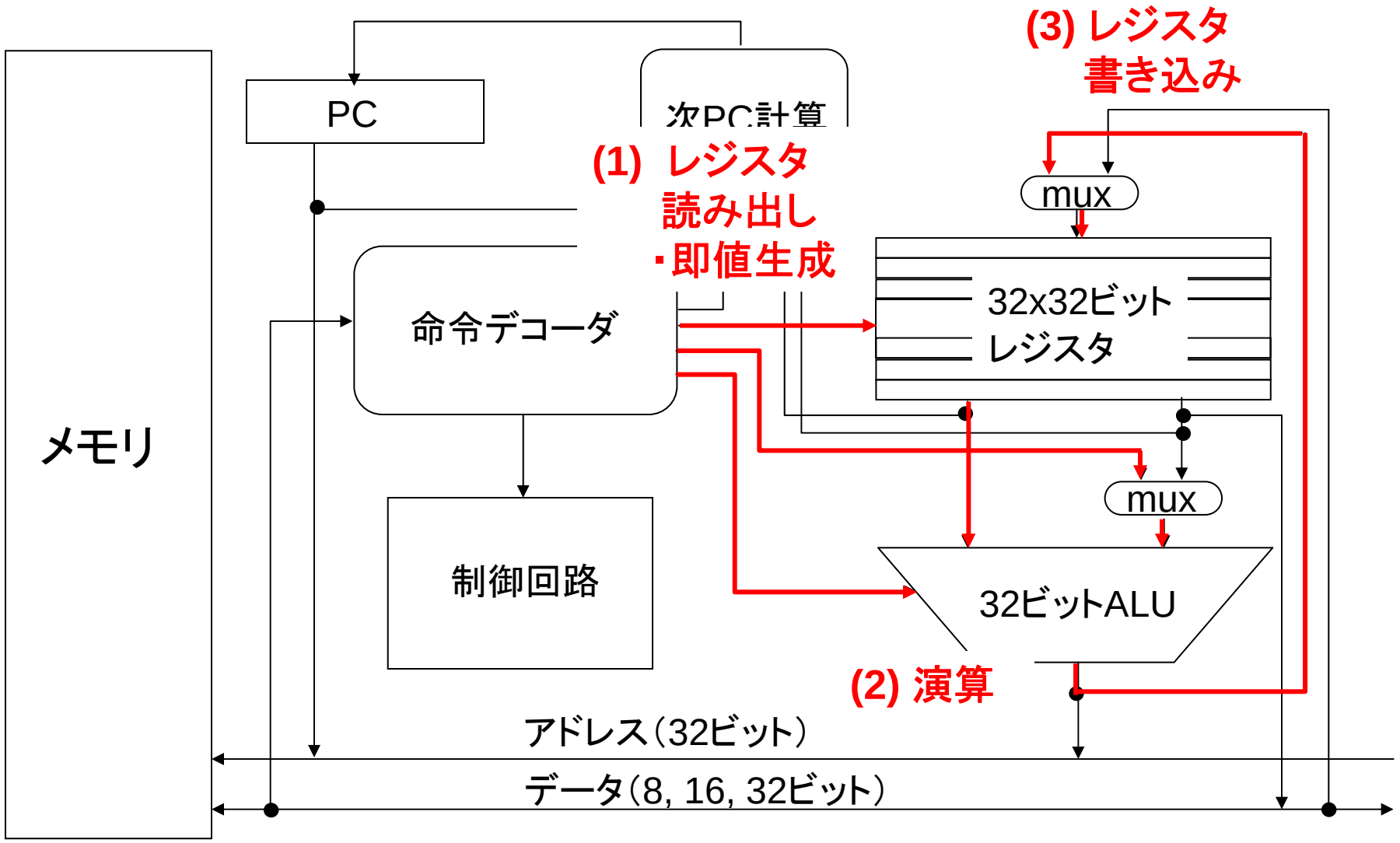
mux は選択回路

レジスタ間演算の動作



mux は選択回路

レジスタ-即値間演算の動作



mux は選択回路

ロード・ストア命令

- レジスタとメモリアドレスを指定して，相互間でデータ転送を行う
- メモリアドレスの指定方法をアドレッシングモードと呼ぶ
- MIPSの場合，アドレッシングモードは1つしかない

```
lw $s1, 12($s0)        # $s1 ← mem[12 + $s0]
```

- 操作対象のメモリアドレス（実効アドレス）はレジスタ s0 の値と定数 12 を加えたもの
- 例えばレジスタ s0 に 10000 が保存されていたとすると，アドレス 10012 から始まる 4 バイト (= 1 ワード) のデータを読み出し，レジスタ s1 に書き込む

資料: 主なロード・ストア命令

命令	説明
lw $t, offset($base)	$t ← mem[offset + $base] (load word)
sw $t, offset($base)	mem[offset + $base] ← $t (store word)

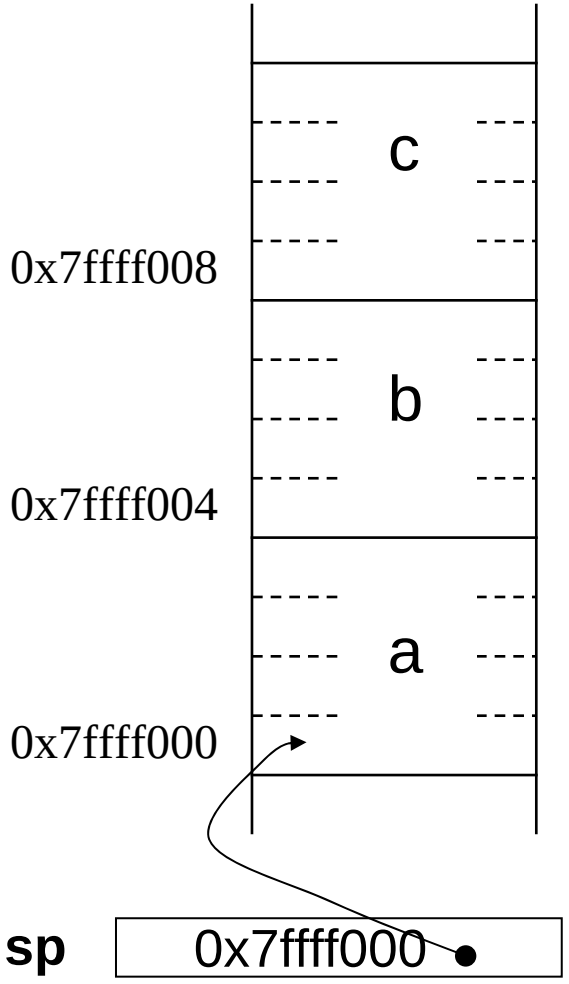
- 他に，half word 単位や byte 単位のロード命令，ストア命令がある

例

(C言語)

```
c = a + b;
```

ただし，各変数は右図のようにメモリに割り当てられているとし，レジスタ t0, t1, t2が自由に使えるとする．以下同様．



(MIPSアセンブリ言語)

```
lw $t0, 0($sp)        # $t0 ← mem[$sp + 0]
lw $t1, 4($sp)        # $t1 ← mem[$sp + 4]
addu $t2, $t0, $t1    # $t2 ← $t0 + $t1
sw $t2, 8($sp)        # mem[$sp + 8] ← $t2
```

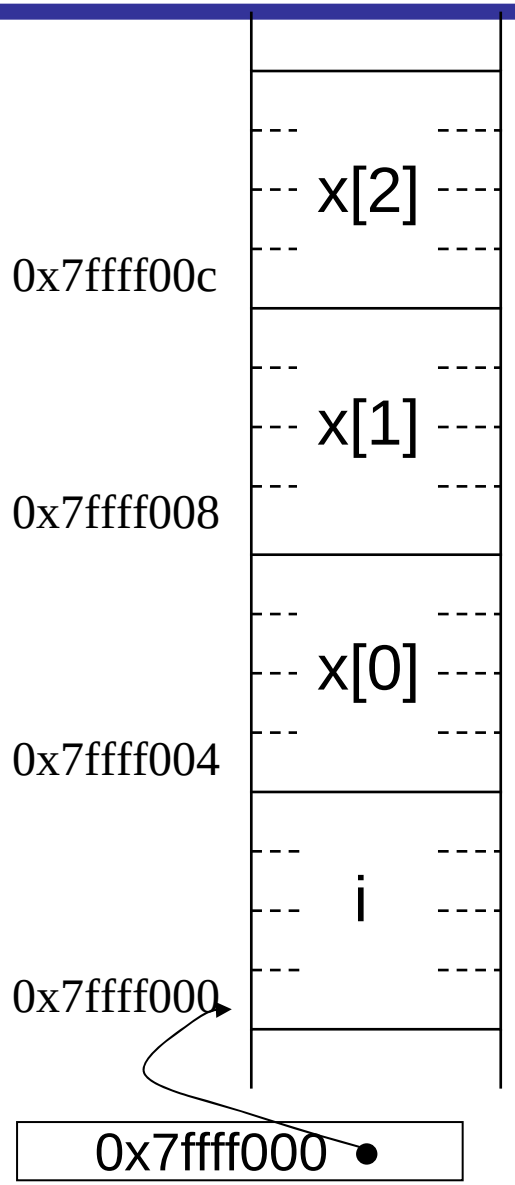
例

(C言語)

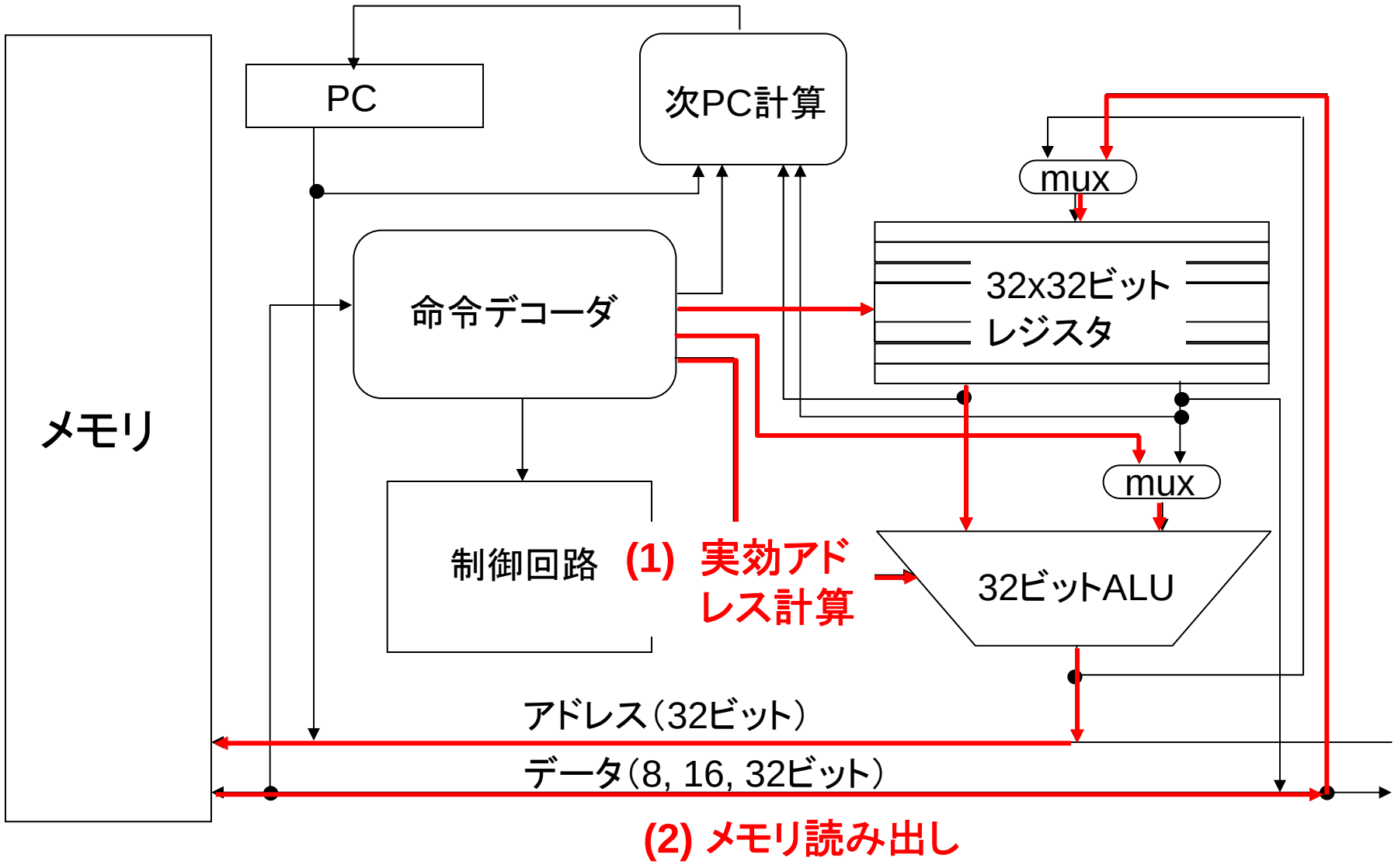
```
int i;
int x[3];

/* 中略 */
x[i] = 300;
```

(MIPSアセンブリ言語)

```
addu $t0, $sp, 4        # $t0 ← $sp + 4
lw $t1, 0($sp)          # $t1 ← mem[$sp + 0]
sll $t1, $t1, 2         # $t1 ← $t1 × 4
addu $t0, $t0, $t1      # $t0 ← $t0 + $t1
li $t2, 300             # $t2 ← 300
sw $t2, 0($t0)          # mem[$t0] ← $t2
```

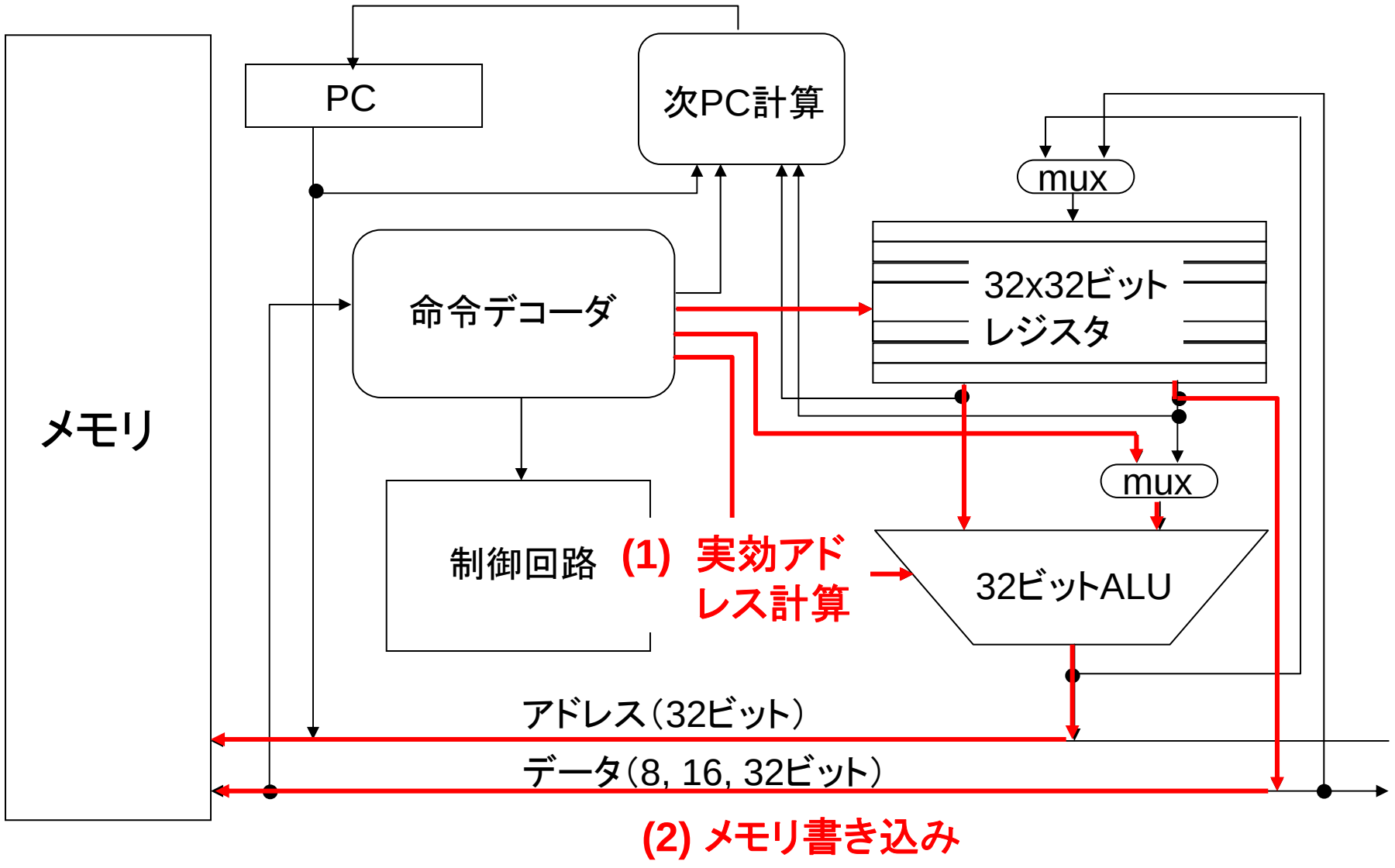


ロード命令の動作



mux は選択回路

ストア命令の動作

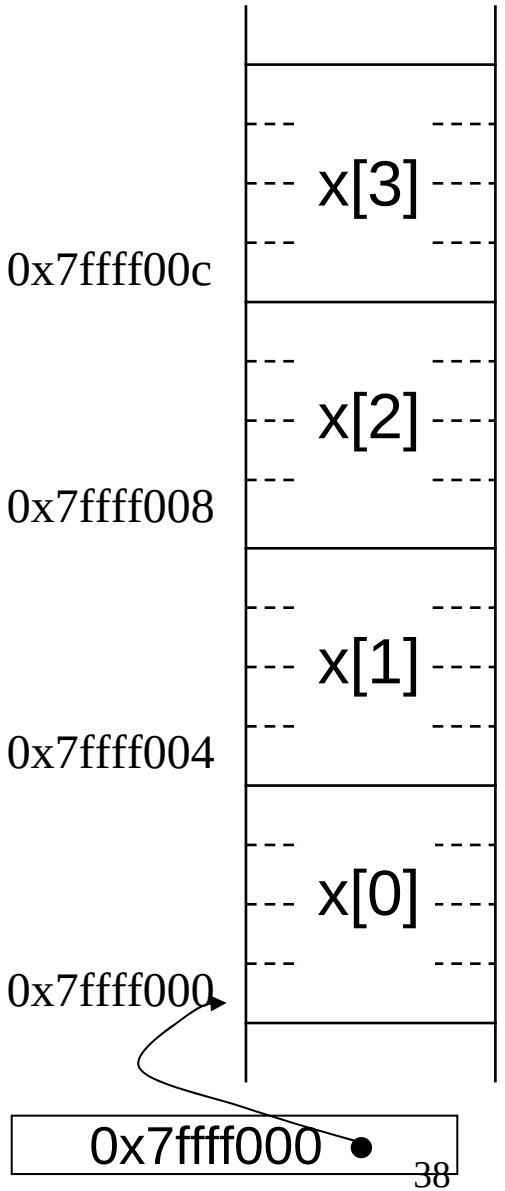


mux は選択回路

練習問題

4要素の4バイト整数型からなる配列が右図のようにメモリに配置されているとする. x[0] … x[3] に格納されている値がそれぞれ 1, 3, 5, 7 の状態から, 以下のコードを実行した. 実行後の x[0] … x[3] の値はどうなっているか.

```
lw $t0, 0($sp)
addu $t0, $t0, 3
sw $t0, 0($sp)
lw $t1, 4($sp)
lw $t0, 8($sp)
sll $t0, $t0, $t1
sw $t0, 8($sp)
lw $t0, 12($sp)
slt $t0, $t0, $t1
sw $t0, 12($sp)
```



練習問題 解答例

```
lw $t0, 0($sp)          # $t0 ← x[0] (= 1)
addu $t0, $t0, 3        # $t0 ← $t0 + 3 (= 1 + 3)
sw $t0, 0($sp)          # x[0] ← $t0 (= 4)
lw $t1, 4($sp)          # $t1 ← x[1] (= 3)
lw $t0, 8($sp)          # $t0 ← x[2] (= 5)
sll $t0, $t0, $t1       # $t0 ← $t0 << $t1 (= 5 << 3)
sw $t0, 8($sp)          # x[2] ← $t0 (= 40)
lw $t0, 12($sp)         # $t0 ← x[3] (= 7)
slt $t0, $t0, $t1       # $t0 ← ($t0 < $t1) ? 1 : 0
sw $t0, 12($sp)         # x[3] ← $t0 (= 0)
```

結局 x[0] … x[3] は 4, 3, 40, 0 になる.