東北大学 工学部 機械知能・航空工学科
2020年度 クラス C D

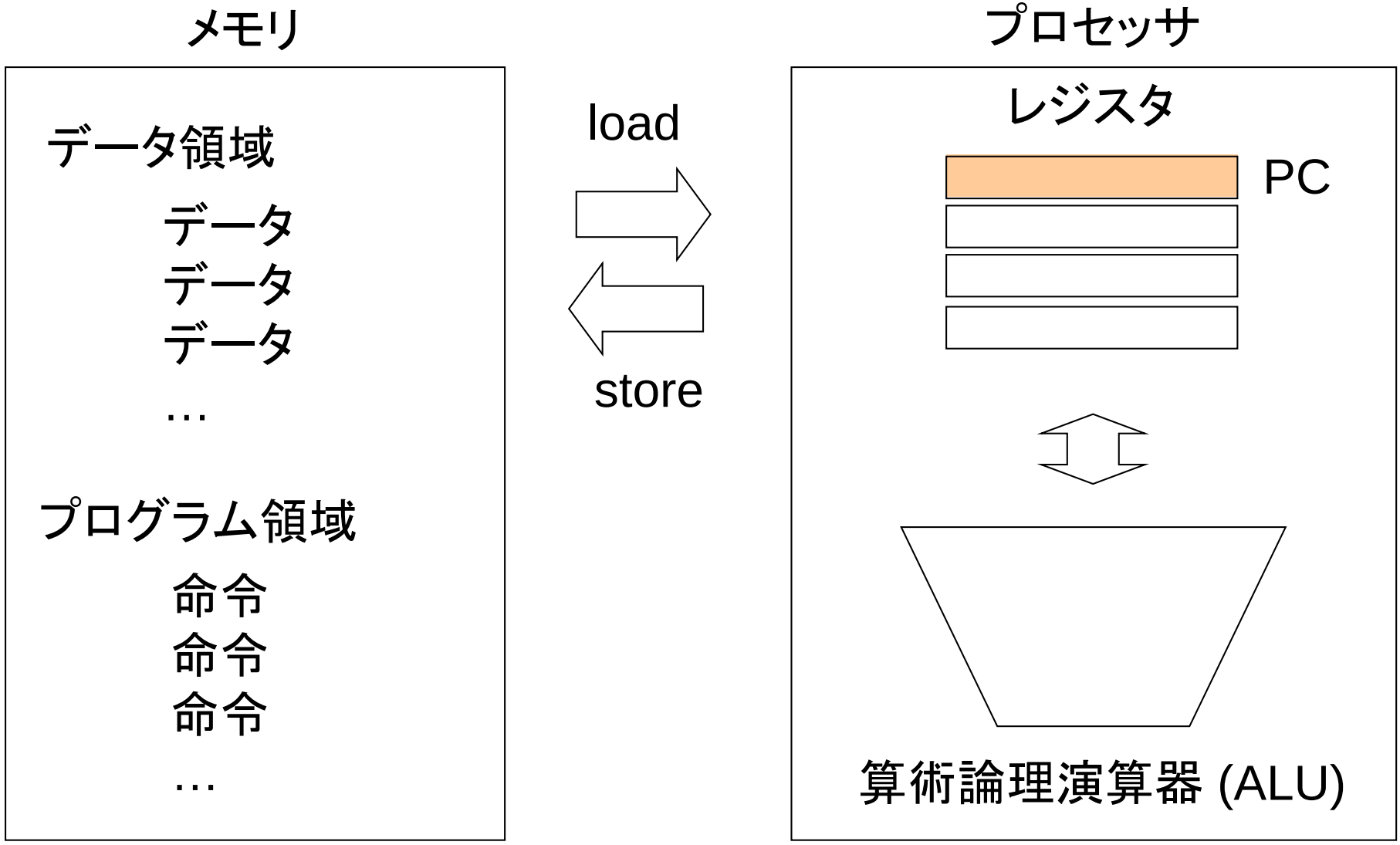
情報科学基礎 I

5. 命令セットアーキテクチャ

(教科書6.1節, 6.2節)

大学院情報科学研究科
鏡 慎吾

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
算術論理演算器 (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
  - そのためプロセッサは，次に実行する命令のアドレスを覚えておかなくてはならない
- プロセッサ内に設けられた小さな記憶領域を一般にレジスタと呼ぶ
  - プログラムカウンタ (PC) : 次の命令アドレスを保持するレジスタ
  - 汎用レジスタ: 演算などのために多用途で用いるレジスタ
- 必要に応じてメモリとレジスタ間でデータを移動する
  - load: メモリ → レジスタ
  - store: メモリ ← レジスタ

  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)
- 演算回路を ALU (Arithmetic Logic Unit) と呼ぶ
  - 単に演算器あるいは演算ユニットと呼ぶことも多い

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 用プロセッサ．ゲーム機では PlayStation 4, PlayStation 5, Xbox, Xbox One など．そのほか採用製品多数

- ARM
  携帯機器，スマートフォン，タブレットの大多数．ゲーム機では Switch, PlayStation Vita, ゲームボーイアドバンス, ニンテンドーDS など．ほか組込み製品多数

- MIPS
  Silicon Graphics 社ほかのワークステーション，ゲーム機では PlayStation, PlayStation 2, Nintendo 64, PlayStation Portable など．ほか組込み製品多数

注: これらは正確には「命令セットのファミリ」とでも呼ぶべきもので，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, …）

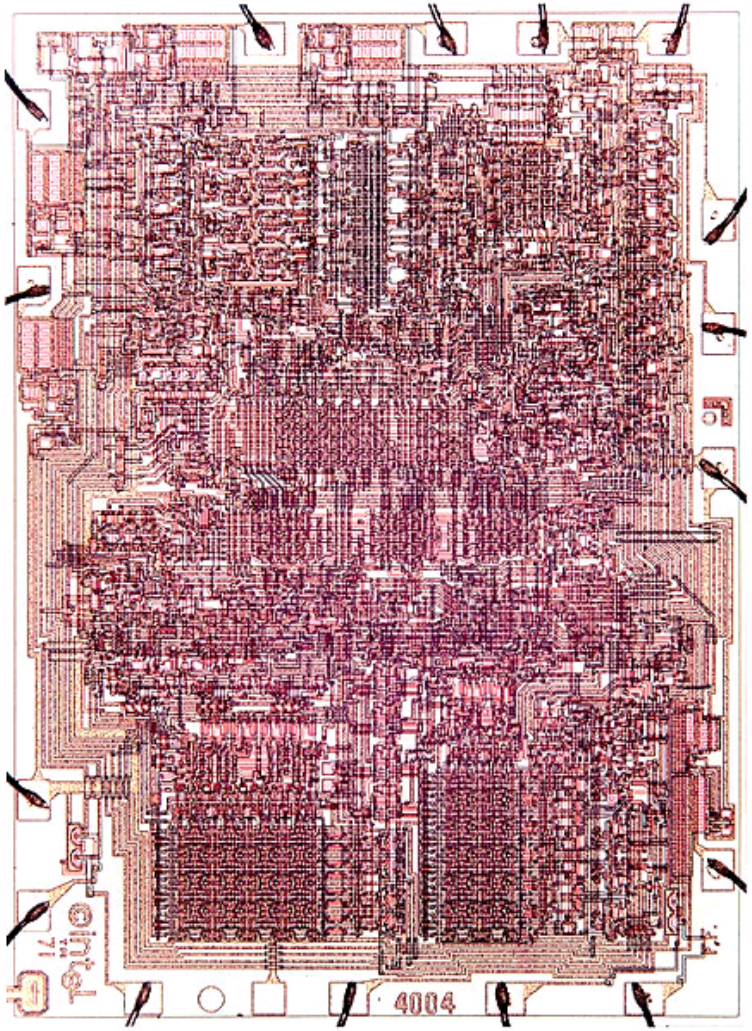
続・計算機の歴史

マイクロプロセッサの登場と変革

- 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 (1964) 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）

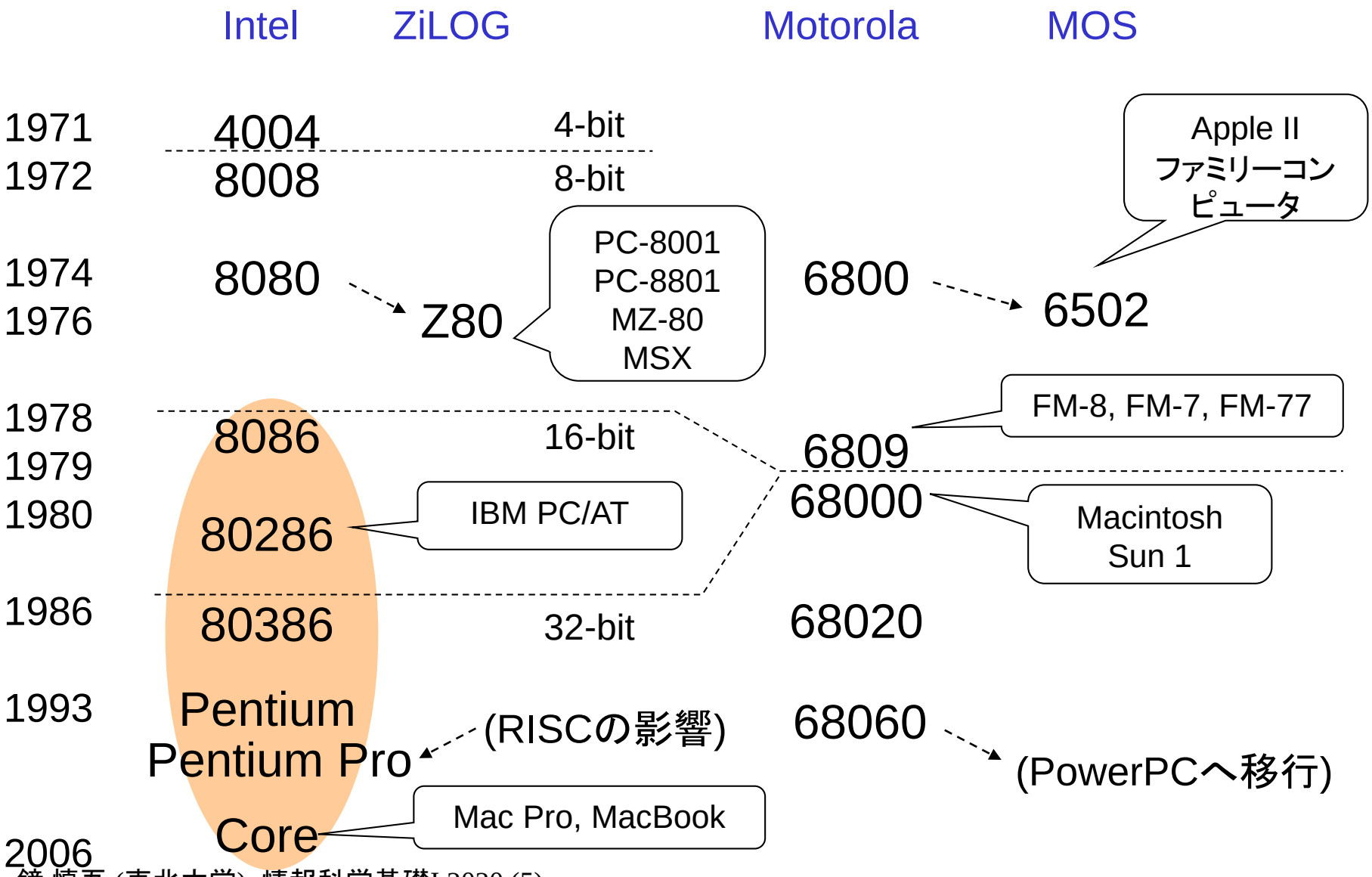
- 1980年代頃，RISCへの転回
  - **RISC** (Reduced Instruction Set Computer):
    命令セットを簡素化し，回路を単純化することで高速化
  - **CISC** (Complex Instruction Set Computer):
    RISC に対して従来のアーキテクチャをこう呼んだ

Intel 4004

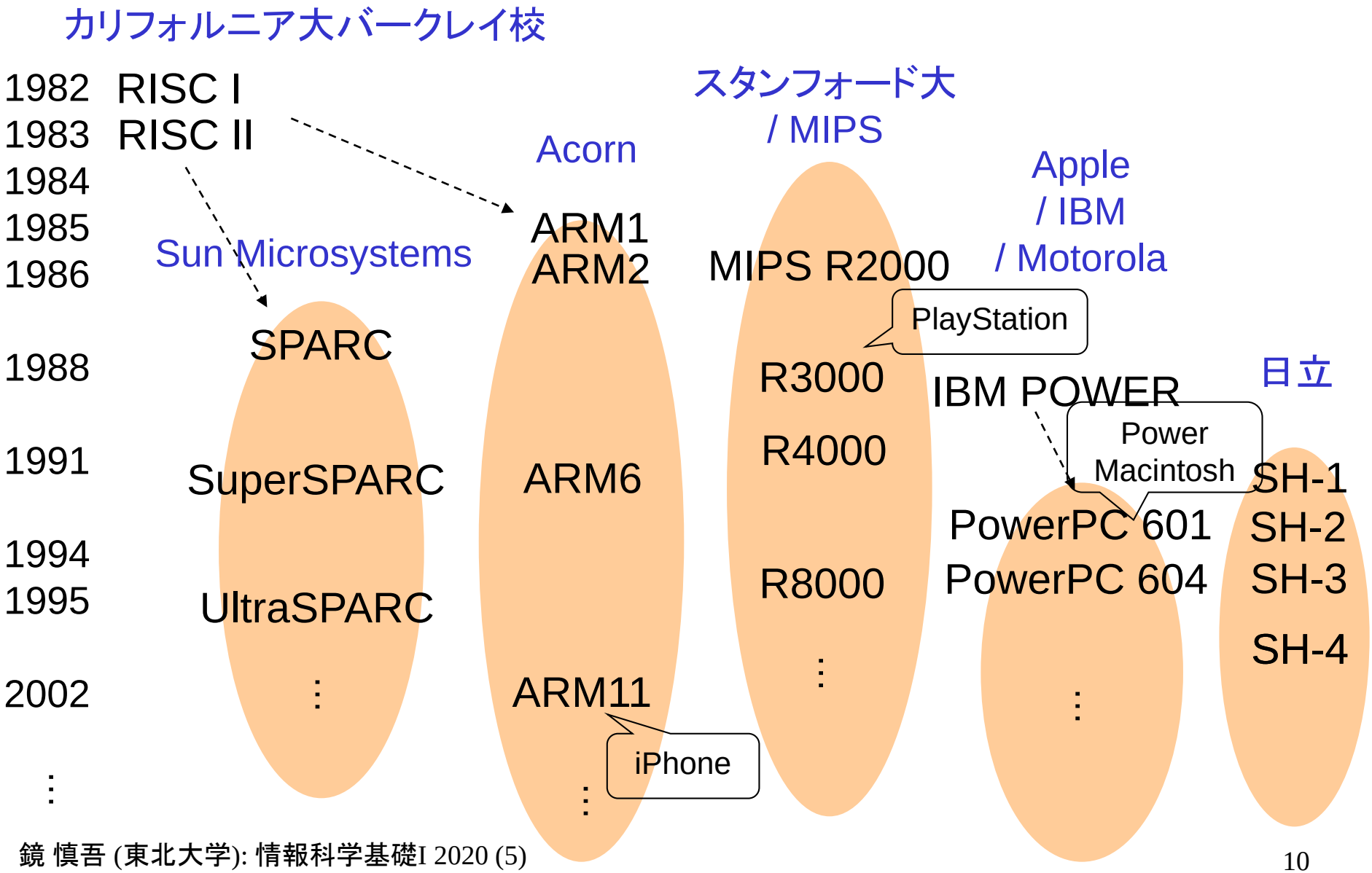


http://news.com.com/1971+Intel+4004+processor/2009-1006_3-6038974-3.html

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)
任天堂 ゲームキューブ (2001): PowerPC 750
マイクロソフト Xbox (2001): Mobile Celeron (x86)

マイクロソフト Xbox 360 (2005): Xenon (PowerPC)
ソニー PlayStation3 (2006): Cell (PowerPC)
任天堂 Wii (2006): Broadway (PowerPC)
任天堂 Wii U (2010): Espresso (Power)
ソニー PlayStation4 (2013): AMD Jaguar (x86)
マイクロソフト Xbox One (2013): AMD Jaguar (x86)
任天堂 Switch (2017): NVIDIA Tegra X1 (ARM Cortex-A57/A53)
ソニー PlayStation 5 (2020): AMD Ryzen (x86)
マイクロソフト Xbox One X (2020): AMD Ryzen (x86)

携帯電話・タブレット端末用プロセッサ

- Qualcomm SnapDragon (ARM)
- Apple A (ARM)
- HiSilicon Kirin (ARM)
- Samsung Exynos (ARM)
- MediaTek Helio (ARM)
- NVIDIA Tegra (ARM)
- Intel Atom (x86)

MIPSアーキテクチャ

この講義では，MIPS 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
- 世界中の大学の講義で取り上げられている
- 組込みプロセッサとして実際に世界中で使われている（たぶん）

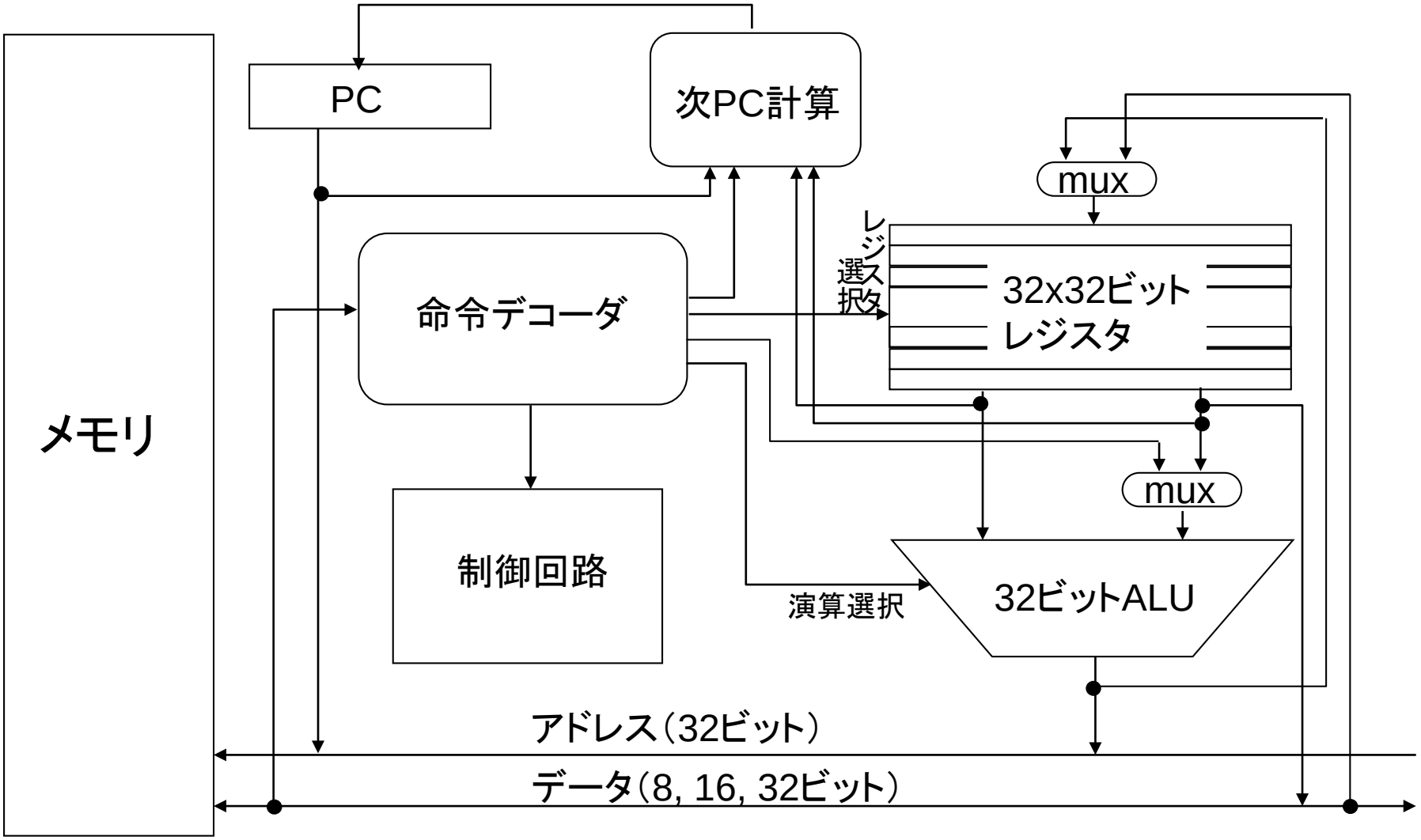
https://www.buffalo.jp/product/detail/wsr-1166dhp4-bk.html

https://wavecomp.ai/blog/ingenic-t10-processor-mips-based-360-camera/

はやぶさ２
http://jda.jaxa.jp/
素材番号 P100006873

MIPSプロセッサの基本的動作

MIPS（32ビット）の構造



mux は選択回路

MIPS の命令の例

(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 a, b, c という名前のレジスタに置かれているとする (「$a」でレジスタ a の値を表す)
- 特に断らない限り変数は整数 (int 型) 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

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
```

- 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，変数名をレジスタ名であるかのように用いて来たが，本来のレジスタ名ではない．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で書き表すことが多い

```
addu $t2, $t0, $t1      # $t2 ← $t0 + $t1
```

ただし 0番レジスタ（別名 $zero）は**常に値 0 が読み出される**特殊なレジスタである（したがって厳密には記憶素子ではない）

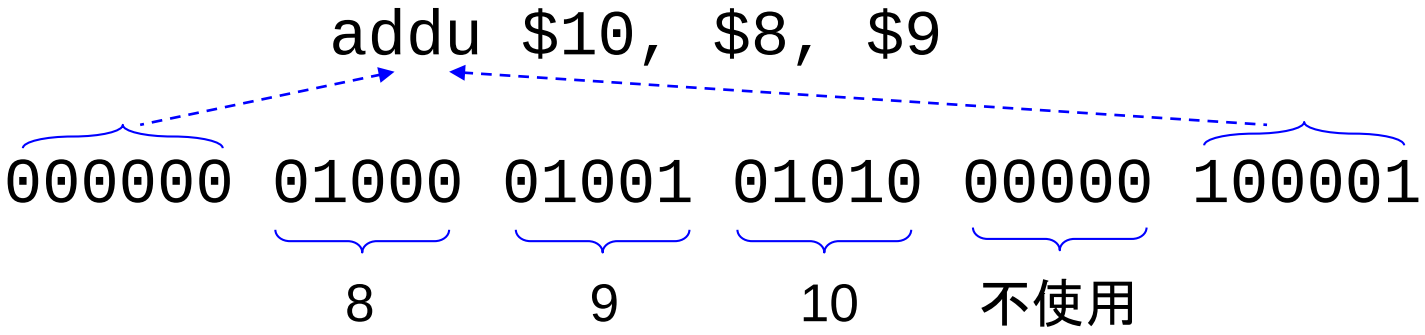
資料: レジスター覧

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

用語

- **オペコード** (opcode, operation code)
  - addu や subu などのように演算の種類を表すもの
  - 狭義にはそれに割り当てられた2進符号を指し，addu のような名前はオペコード・ニーモニック (opcode mnemonic) と呼ぶこともある

- **オペランド** (operand)
  - $10, $t0, $zero などのように演算の対象となるもの
  - 演算の入力になるものを入力オペランドと呼ぶ
  - 演算結果が書き込まれるものを出力オペランドと呼ぶ



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/

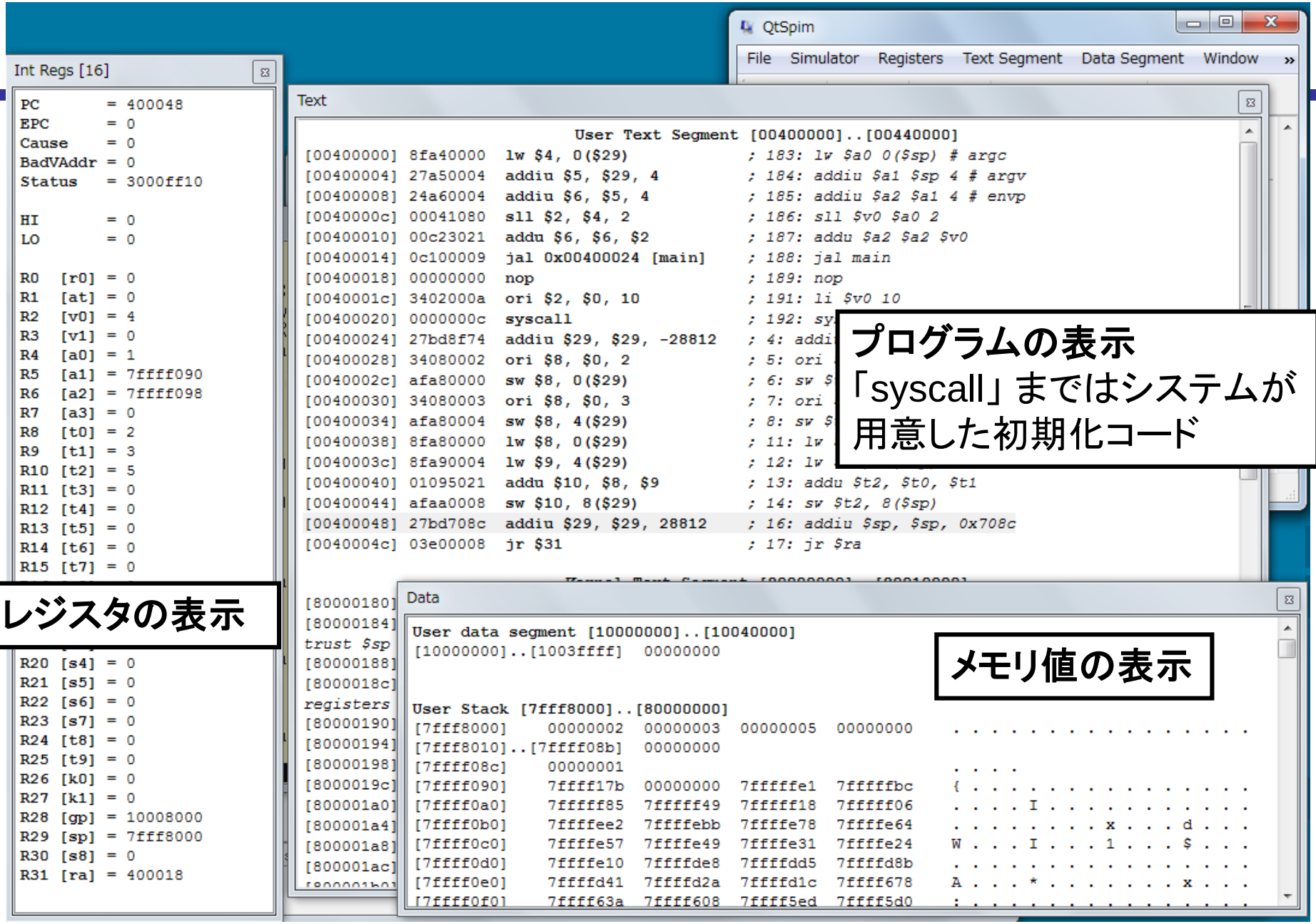
- Windows, macOS, Linux で動作

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:

- https://github.com/shingo-kagami/fis1/tree/master/asm


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
レジスタの表示
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
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
Data
User data segment [10000000]..[10040000]
[10000000]..[1003ffff] 00000000
メモリ値の表示
User Stack [7fff8000]..[80000000]
[7fff8000] 00000002 00000003 00000005 00000000
[7fff8010]..[7ffff08b] 00000000
[7ffff08c] 00000001
[7ffff090] 7ffff17b 00000000 7fffffe1 7fffffbc
[7ffff0a0] 7fffff85 7fffff49 7fffff18 7fffff06
[7ffff0b0] 7ffffee2 7ffffebb 7ffffe78 7ffffe64
[7ffff0c0] 7ffffe57 7ffffe49 7ffffe31 7ffffe24
[7ffff0d0] 7ffffe10 7ffffde8 7ffffdd5 7ffffd8b
[7ffff0e0] 7ffffd41 7ffffd2a 7ffffd1c 7ffff678
[7ffff0f0] 7ffff63a 7ffff608 7ffff5ed 7ffff5d0

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, -4096
    li    $t0, 1
    sw    $t0, 0($sp)

###
    addu  $t0, $sp, 4      # start
    lw    $t1, 0($sp)
    sll   $t1, $t1, 2
    addu  $t0, $t0, $t1
    li    $t2, 300
    sw    $t2, 0($t0)
###
    addu  $sp, $sp, 4096   # end
    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）

講義のその時点で理解してもらいたい部分

main の終了処理

シミュレータをインストールしたくない人向け

https://cpulator.01xz.net/

Architecture: MIPS32r5 (no delay slots)
System: MIPS (no delay slots) SPIM

- アセンブリ言語のエディタウィンドウの「main:」の次の行以降に，サンプルコードの「main:」の次の行以降の内容を貼り付ける
- Compile and Load (F5キー)
- Step Into (F2キー) で 1 行ずつ実行

例題

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以下のように書けるようになっている（アセンブラが自動変換してくれる）．このような命令を**マクロ命令**と呼ぶ

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

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．

解答例

1. 例えば以下の各命令（ほか多数）

```
or   $s1, $s0, $zero
or   $s1, $zero, $s0
addu $s1, $s0, $zero
```

2. 例えば以下の各命令

```
nor  $s1, $s0, $zero
nor  $s1, $zero, $s0
```

用語の確認

- レジスタ
- ロード と ストア
- ALU
- プログラムカウンタ (PC)
- RISC と CISC
- x86, ARM, MIPS
- オペコード と オペランド