2010年 後学期

計算機アーキテクチャ 第二 (O)

3．RISC vs. CISC，RISCプロセッサ

大学院情報理工学研究科 計算工学専攻
吉瀬謙二 kise _at_ cs.titech.ac.jp
S321講義室 月曜日 5，6時限 13:20−14:50

Sample program

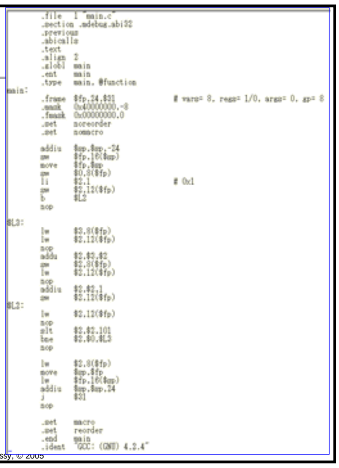
```
#include <stdio.h>

int main(){
  int i;
  int sum = 0;

  for(i=1; i<=100; i++)
    sum += i;

  return sum;
}
```

遅延分岐



```
        .file   1 "main.c"
        .section .mdebug.abi32
        .previous
        .abicalls
        .text
        .align  2
        .globl  main
        .ent    main
        .type   main, @function
main:
        .frame  $fp,24,$31              # vars= 8, regs= 1/0, args= 0, gp= 8
        .mask   0x40000000,-8
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro

        addiu   $sp,$sp,-24
        sw      $fp,16($sp)
        move    $fp,$sp
        sw      $0,8($fp)
        li      $2,1                    # 0x1
        sw      $2,12($fp)
        b       $L2
        nop

$L3:
        lw      $3,8($fp)
        lw      $2,12($fp)
        nop
        addu    $2,$3,$2
        sw      $2,8($fp)
        lw      $2,12($fp)
        nop
        addiu   $2,$2,1
        sw      $2,12($fp)
$L2:
        lw      $2,12($fp)
        nop
        slt     $2,$2,101
        bne     $2,$0,$L3
        nop

        lw      $2,8($fp)
        move    $sp,$fp
        lw      $fp,16($sp)
        addiu   $sp,$sp,24
        j       $31
        nop

        .set    macro
        .set    reorder
        .end    main
        .ident  "GCC: (GNU) 4.2.4"
```

Sample program

```
        .file   1 "main.c"
        .section .mdebug.abi32
        .previous
        .abicalls
        .text
        .align  2
        .globl  main
        .ent    main
        .type   main, @function
main:
        .frame  $sp,0,$31               # vars= 0, regs= 0/0, args= 0, gp= 0
        .mask   0x00000000,0
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro

        j       $31
        li      $2,5050                 # 0x13ba

        .set    macro
        .set    reorder
        .end    main
        .ident  "GCC: (GNU) 4.2.4"
```

```
# Makefile
all:
        mipsel-linux-gcc -O0 -S main.c -o main_opt0.s
        mipsel-linux-gcc -O1 -S main.c -o main_opt1.s
        mipsel-linux-gcc -O2 -S main.c -o main_opt2.s
        mipsel-linux-gcc -O3 -S main.c -o main_opt3.s
```

2010年 後学期

計算機アーキテクチャ 第二 (O)

2．RISC vs. CISC，RISCプロセッサ

大学院情報理工学研究科 計算工学専攻
吉瀬謙二 kise _at_ cs.titech.ac.jp
S321講義室 月曜日 5，6時限 13:20−14:50

CISC - Complex Instruction Set Computer

- **CISC philosophy**
  - ! fixed instruction lengths
  - ! load-store instruction sets
  - ! limited addressing modes
  - ! limited operations
- DEC VAX11 Intel 80x86, …

IA - 32

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: 57 new "MMX" instructions are added, Pentium II
- 1999: The Pentium III added another 70 instructions (SSE)
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends the architecture to increase address space to 64 bits, widens all registers to 64 bits and other changes (AMD64)
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and adds more media extensions
- "This history illustrates the impact of the "golden handcuffs" of compatibility

  "adding new features as someone might add clothing to a packed bag"

  "an architecture that is difficult to explain and impossible to love"

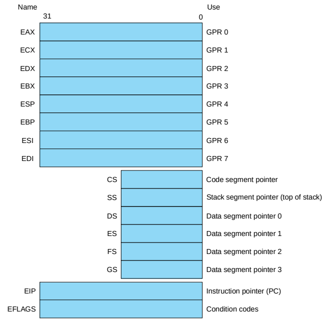
IA-32 Overview

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

7

IA-32 Registers and Data Addressing

- Registers in the 32-bit subset that originated with 80386



8

IA-32 Register Restrictions

- Registers are not “general purpose” – note the restrictions below

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	`lw $s0,100($s1) # ≤16-bit # displacement`
Base plus scaled index	The address is Base + (2^{Scale} x index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,0($t0)`
Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,100($t0) # ≤16-bit # displacement`

FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

9

IA-32 Typical Instructions

- Four major types of integer instructions:
  - Data movement including move, push, pop
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags)
  - String instructions, including string move and string compare

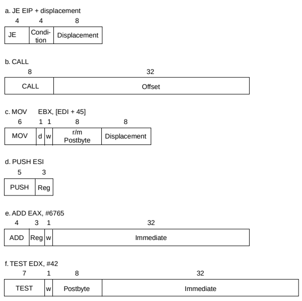
Instruction	Function
JE name	if equal(condition code) {EIP=name}; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags) with EDX and 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

FIGURE 2.43 Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

10

IA-32 instruction Formats

- Typical formats: (notice the different lengths)



11

VAX CALLS命令

1. 必要ならばスタックを整列化する.
2. 引数の個数をスタックにプッシュする.
3. スタック上の手続き呼出しマスクによって指示されたレジスタの退避をおこなう. マスクは呼び出される手続きのコード内に保持されている. これによって分割コンパイルの際にも, 被呼出側退避を呼出し側で実行できるようになる.
4. リターン・アドレスをスタックにプッシュし, 現在の活動記録に対するスタック・トップとスタック・ベースをプッシュする.
5. トラップ・イネーブルを既知の状態にセットする条件コードをクリアする.
6. ステータス情報のための語とゼロの値を持つ語をスタックにプッシュする.
7. 2つのスタック・ポインタを呼び出された手続きで利用できるように更新する.
8. 呼び出された手続きの最初の命令に分岐する.

12

RISC vs. CISC

- Section B.2
  - Use general-purpose registers with a load-store architecture.

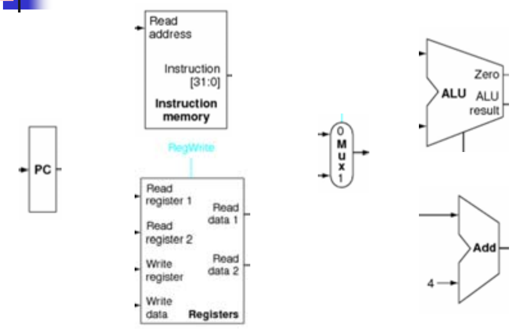
Computer Architecture A Quantitative Approach Fourth Edition

- 落とし穴
  - 高級言語構造を特別に支援することを目的に，高レベルの命令セットを設計すること．
- 誤信
  - 欠点のあるアーキテクチャは成功しない．

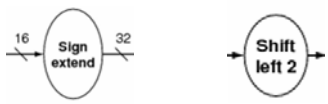
計算機アーキテクチャ 第二 (O)

RISCプロセッサとパイプライン処理

プロセッサの構成要素（1）



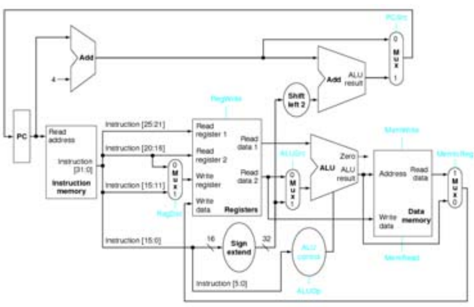
プロセッサの構成要素（2）



プロセッサのデータパス（シングル・サイクル）

op	rs	rt	rd	shamt	funct

add $t0, $s1, $s2 [add $8, $17, $18]

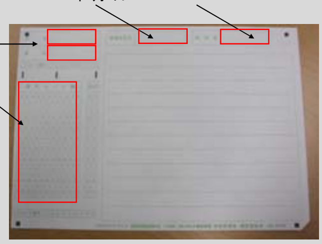


スキャネットシート

年 月 日　Arch II

氏名，学籍番号，

学籍番号マーク欄(右詰で)

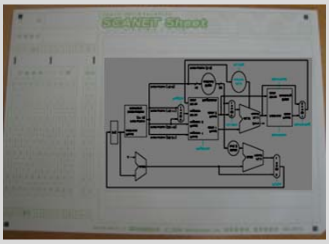


Exercise

op	rs	rt	16 bit immediate

I format

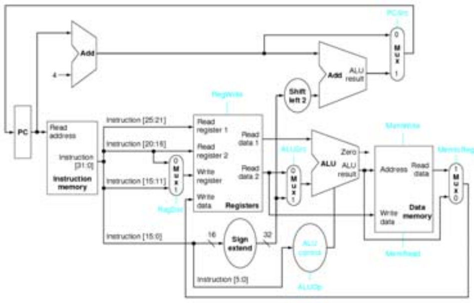
addi $t0, $t1, -1 **[addi $8, $9, -1]**



プロセッサのデータパス(シングル･サイクル)

op	rs	rt	rd	shamt	funct

add $t0, $s1, $s2 **[add $8, $17, $18]**



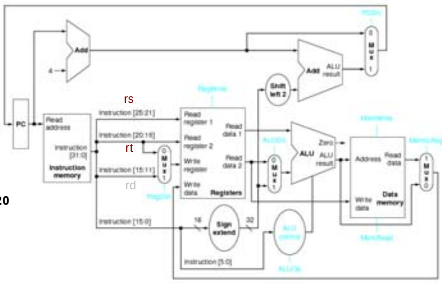
プロセッサのデータパス(シングル･サイクル)

op	rs	rt	16 bit immediate

I format

addi $sp, $sp, 4 **[addi $29, $29, 4]**

PC = 0x20
$29 = 7



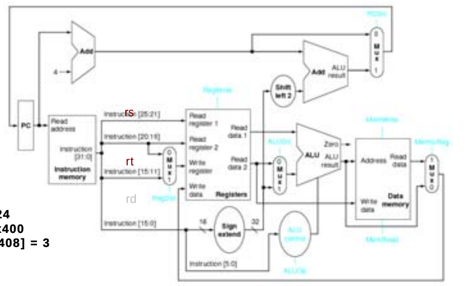
プロセッサのデータパス(シングル･サイクル)

op	rs	rt	16 bit immediate

I format

lw $t0, 8($s2) **[lw $8, 8($18)]**

PC = 0x24
$18 = 0x400
mem[0x408] = 3

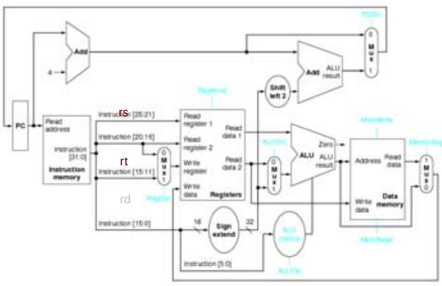


プロセッサのデータパス(シングル･サイクル)

op	rs	rt	16 bit immediate

I format

sw $t0, 24($s2) **[sw $8, 24($18)]**



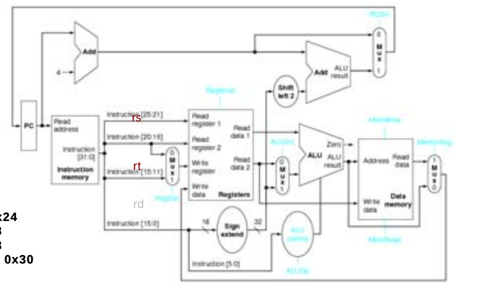
プロセッサのデータパス(シングル･サイクル) Exercise

op	rs	rt	16 bit immediate

I format

beq $s0, $s1, Label **[beq $16, $17, Label]**

PC = 0x24
$16 = 8
$17 = 8
Label = 0x30



MIPS **Control Flow** Instructions

- MIPS conditional branch instructions:

```
bne $s0, $s1, Lbl #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl #go to Lbl if $s0=$s1
```

  - Ex:

```
          if (i==j) h = i + j;

          bne $s0, $s1, Lbl1
          add $s3, $s0, $s1
Lbl1:     ...
```

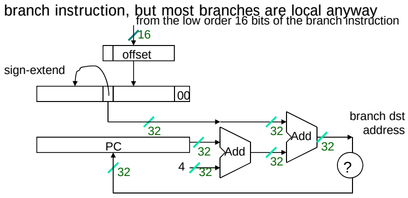
- Instruction Format (I format):

op	rs	rt	16 bit offset

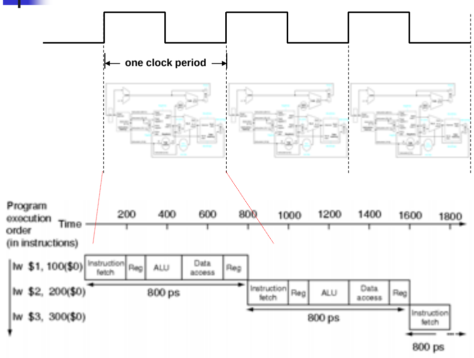
- How is the branch destination address specified?

Specifying Branch Destinations

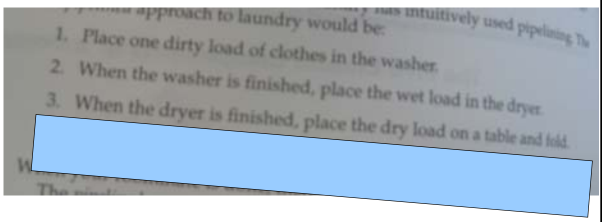
- Use a register (like in lw and sw) added to the 16-bit offset
  - which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
  - limits the branch distance to -2^{15} to $+2^{15}-1$ instructions from the (instruction after the) branch instruction, but most branches are local anyway



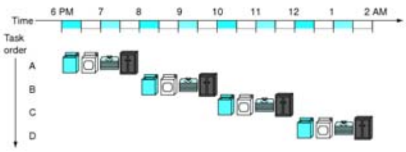
プロセッサのデータパス（シングル･サイクル）



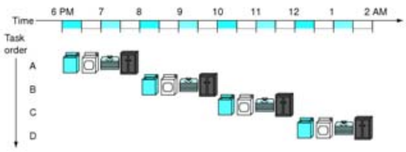
パイプライン処理 (pipelining)

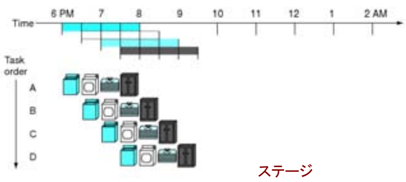


パイプライン処理 (pipelining)



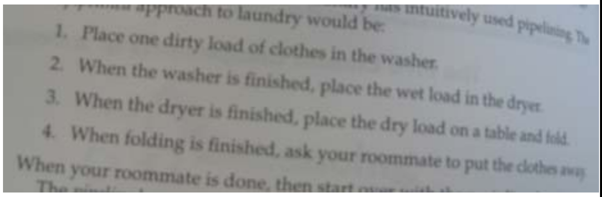
パイプライン処理 (pipelining)



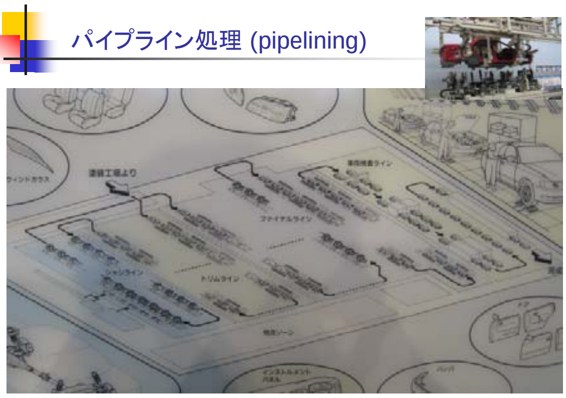


ステージ

パイプライン処理 (pipelining)



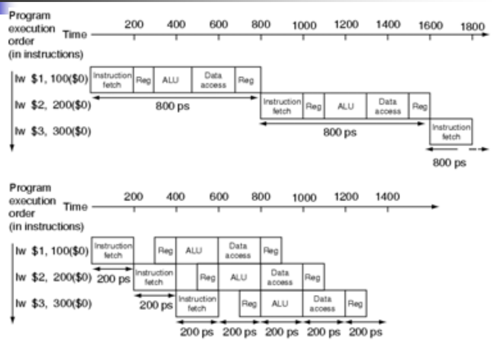
パイプライン処理 (pipelining)



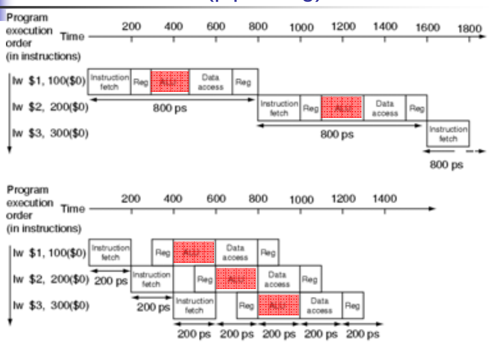
MIPSの基本的な5つのステップ（ステージ）

- **IF (Instruction fetch)ステージ**
  メモリから命令をフェッチする.
- **ID (Instruction decode and register file read) ステージ**
  命令をデコードしながら, レジスタを読み出す.
- **EX (Execution or address calculation) ステージ**
  命令操作の実行またはアドレスの生成を行う.
- **MEM (Data memory access) ステージ**
  データ・メモリ中のオペランドにアクセスする.
- **WB (Write back) ステージ**
  結果をレジスタに書き込む.

パイプライン処理 (pipelining)



パイプライン処理 (pipelining)



パイプラインによる速度向上

- パイプラインステージの数(段数): **n**
- 実行する命令の数: **s**
- パイプライン化されたプロセッサのクロックを単位時間とする.
- 全命令が終了するまでの理想的なサイクル数
  - **n + s – 1**
- パイプラインを利用しないシングルサイクルのプロセッサ
  - **n * s**

アナウンス

- 講義スライド，講義スケジュール
  - www.arch.cs.titech.ac.jp