Lista Rozkazów:

Język komputera

Większość slajdów do tego wykładu to tłumaczenia i przeróbki oficjalnych sladjów do podręcznika Pattersona i Hennessy'ego

Lista rozkazów

- Zestaw rozkazów wykonywanych przez maszynę
- Komputery mają różne listy rozkazów
  - Ale z wieloma wspólnymi cechami
- Dawne komputery miały proste listy rozkazów
  - Łatwa implementacja sprzętowa
- Współczesne maszyny mają zazwyczaj również proste listy rozkazów

Lista rozkazów MIPS

- Nasza modelowa architektura
- Głównie komputery wbudowane
  - Urządzenie elektroniczne, aparaty, drukarki, urządzenia sieciowe i magazynujące
- Typowy przykład współczesnych architektur list rozkazów (Instruction Set Architecture, ISA)

Operacje arytmetyczne

- Dodawanie i odejmowanie
  - Dwa argumenty źródłowe, wynik

```
add a, b, c  # a := b + c
```

- Większość operacji arytmetycznych i logicznych jest tej postaci
- Jednolitość i regularność rozkazów
  - Ułatwia implementację
  - Większa wydajność w niewielkim koszcie

Przykład

- Kod w języku C:

```
f = (g + h) - (i + j);
```

- Kod MIPS:

```
add t0, g, h   # temp t0 := g + h
add t1, i, j   # temp t1 := i + j
sub f, t0, t1  # f := t0 - t1
```

Rejestry

- Rozkazy arytmetyczne działają na rejestrach
- MIPS ma zestaw 32 rejestrów 32-bitowych ogólnego przeznaczenia
  - Przechowują najczęściej używane dane
  - Numerowane od 0 do 31
  - 32-bitowe dane nazywamy *słowami*
- Nazwy rejestrów i konwencja ich użycia
  - $t0, $t1, …, $t9 – przechowują tymczasowe dane
  - $s0, $s1, …, $s7 – przechowują zmienne
  - Inne: o nich później
- Operacje na rejestrach są szybsze niż na pamięci

Przykład

- Kod w C:

```
f = (g + h) - (i + j);
```

  - f, …, j w $s0, …, $s4

- Kod MIPS:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Operacje na pamięci

- Pamięć główna przechowuje zazwyczaj złożone dane:
  - tablice, strukury, dane dynamiczne
- Operacje arytmetyczne wymagają:
  - Załadowanie danych z pamięci do rejestru
  - Zapisania wyniku do pamięci
- Pamięć adresowana jest *bajtowo*
  - Każdy adres wskazuje na 8-bitowy bajt
- Słowa w pamięci ułożone z wyrównaniem 4-bajt.
  - Adres słowa jest wielokrotnością 4
- MIPS działa jako *Big Endian*
  - Najbardziej znaczący bajt słowa ma najniższy adres
  - *Little Endian*: najniższy adres: najmniej znaczący bajt

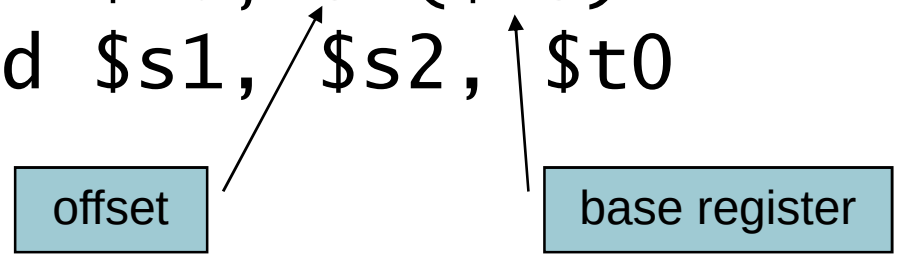
Operacje na pamięci

- Kod w C:

```
g = h + A[8];
```

  - g w $s1, h w $s2, adres bazowy A w $s3
- Kod MIPS:
  - Indeks 8 – przesunięcie adresu o 32=4*8
    - 4 bajty na słowo!

```
lw  $t0, 32($s3)     # load word
add $s1, $s2, $t0
```



Operacje na pamięci

- Kod w C:

  `A[12] = h + A[8];`

  - h w $s2, adres bazowy A w $s3
- Kod MIPS:

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

- Co w przypadku `A[i]`?

Rejestry a pamięć główna

- Szybszy dostęp do rejestrów
- Operacje na pamięci wymagają load i store
  - Więcej rozkazów musi być wykonanych
- Komilatory starają się używać rejestrów jak najczęściej
  - W pamięci tylko najrzadziej używane zmienne
  - Optymalizacja rejestrowa

Argumenty natychmiastowe

- Niektóre rozkazy potrafią działać na stałych (arg. natychmiastowe, ang. *immediate*)

```
addi $s3, $s3, 4
```

- Nie ma rozkazu odejmowania stałej, ale można użyć:
  - `addi $s2, $s1, -1`
- Rozkazy ze stałymi są wygodne
- Stałe 16-bitowe – wymagana sprzętowa konwersja do 32 bitów

Stała zero

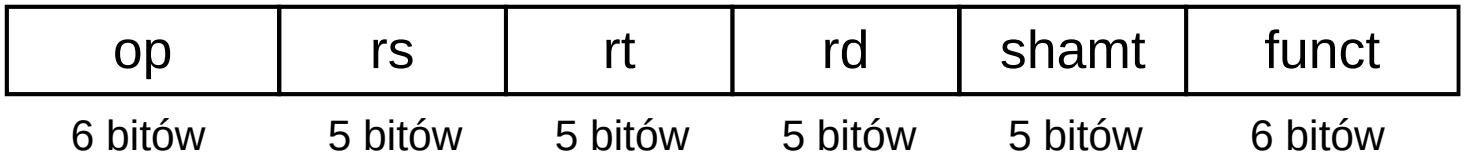
- Rejestr 0 ($zero) zawiera stałą 0 i nie może być modyfikowany
- Wygodne w wielu zastosowaniach
  - Np. kopiowanie zawartości rejestru (nie ma osobnego rozkazu move):

```
add $t2, $s1, $zero
```

Reprezentacja rozkazów

- Rozkazy kodowane są binarnie
  - *Kod maszynowy*
- Rozkazy MIPS:
  - Kodowane jako 32-bitowe słowa
  - Niewielki zestaw *formatów* rozkazów
  - Regularność
- Numery rejestrów
  - $t0 – $t7 to rejestry 8 – 15
  - $t8 – $t9 to rejestry 24 – 25
  - $s0 – $s7 to rejestry 16 – 23

Rozkazy typu R (*Register*)



- Pola rozkazu:
  - op: kod operacji (opcode)
  - rs: numer pierwszego rejestru źródłowego
  - rt: numer drugiego rejestru źródłowego
  - rd: numer rejestru wynikowego
  - shamt: shift amount (00000 na razie)
  - funct: kod funkcji (precyzuje opcode)

Przykład rozkazu typu R

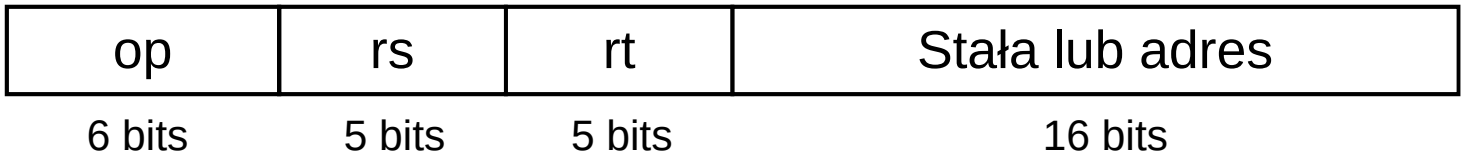
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

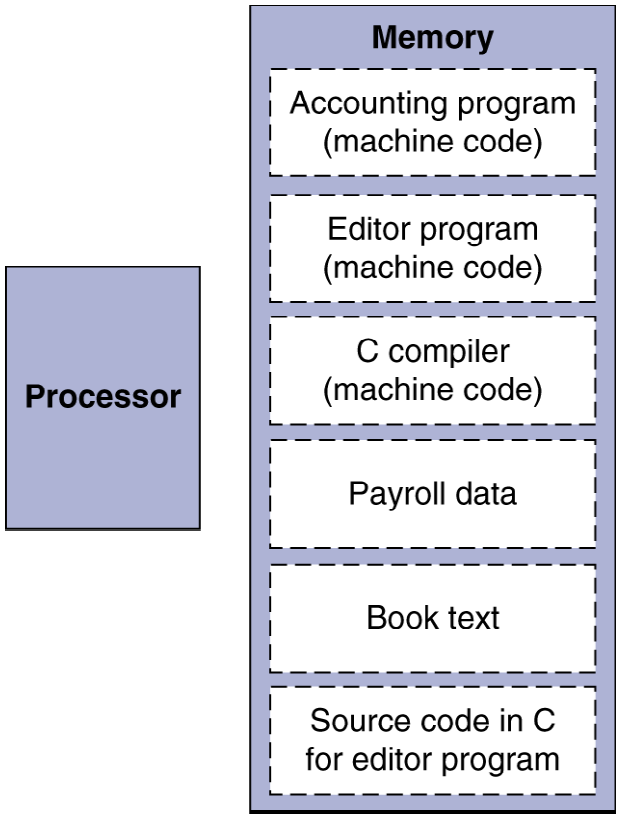
Rozkazy typu I



- Rozkazy ze stałymi oraz load i store
  - rt: rejestr wynikowy lub źródłowy
  - Stała w reprezentacji uzup. do 2: -2^{15} to $+2^{15} - 1$
  - Adres: przesunięcie dodawane do adr. bazowego rs
- Dobrze zaprojektowana lista rozkazów wymaga kompromisów
  - Różne formaty komplikują dekodowanie rozkazów, ale pozwalają utrzymać stałą 32-bitową długość rozkazu
  - Formaty maksymalnie proste

Program w pamięci

The BIG Picture



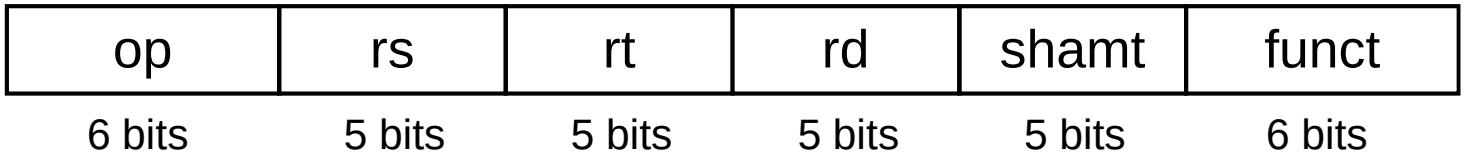
- Rozkazy reprezentowane binarnie tak jak dane
- Rozkazy i dane przechowywane w pamięci
- Programy mogą działać na programach
  - Np. kompilatory

Rozkazy logiczne

- Rozkazy manipulujące bitami w słowach

Operacja	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitowy AND	&	&	and, andi
Bitowy OR	|	|	or, ori
Bitowy NOT	~	~	nor

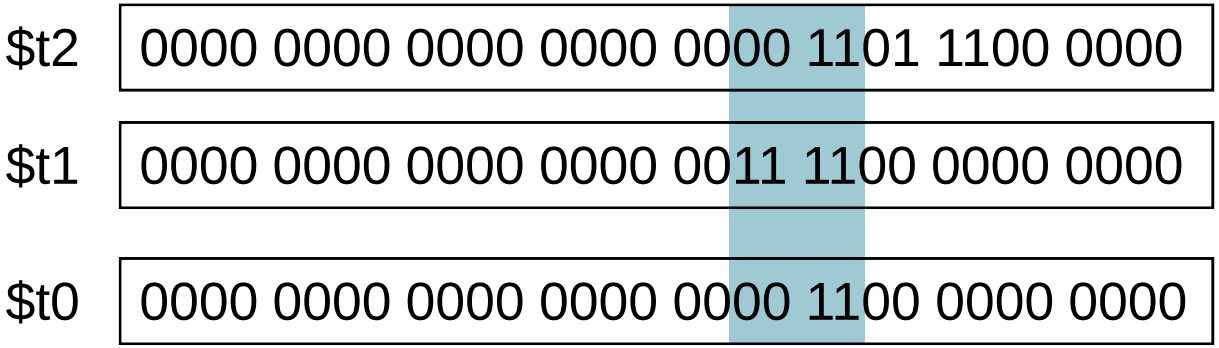
Rozkazy przesuwające (typ R)



- Shamt: o ile pozycji przesunąć
- Przesunięcie logiczne w lewo
  - Przesuwa bity w lewo, dopisując zera
  - `sll` o *i* bitów to mnożenie przez 2^i
  - `Np. sll $s1, $s2, 2`
  - Przesunięcie logiczne w prawo
  - Przesuwa bity w prawo, dopisując zera
  - `srl` o *i* bitów dzieli przez 2^i (tylko dla liczb bez znaku)

Rozkaz AND

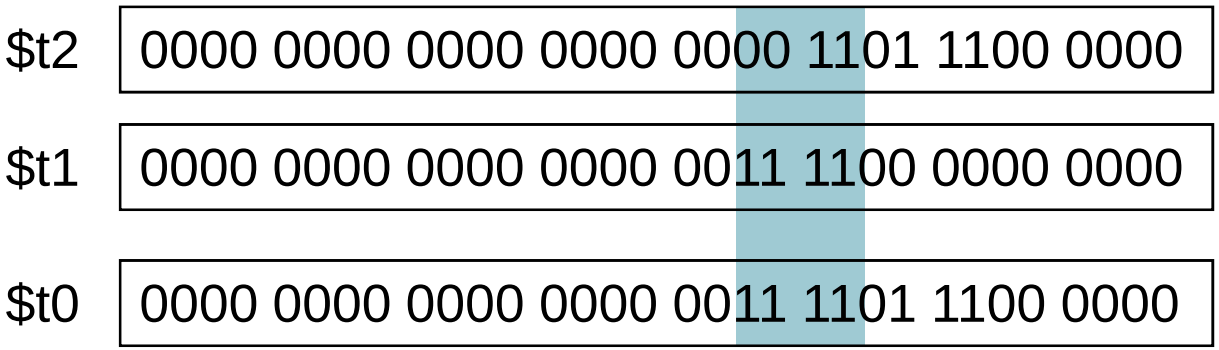
- Wygodne przy zerowaniu pewnych bitów bez zmiany pozostałych
- `and $t0, $t1, $t2`



Rozkaz OR

- Wygodne przy ustawianiu pewnych bitów na 1 bez zmiany pozostałych

```
or $t0, $t1, $t2
```



Rozkaz NOR

- Można go użyć do odwrócenia bitów słowa
  - zamina 0 na 1, a 1 na 0 (NOR $s1, $s1, $s1)
- MIPS ma 3-argumentowy rozkaz NOR (a nie ma NOT)
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Skoki warunkowe

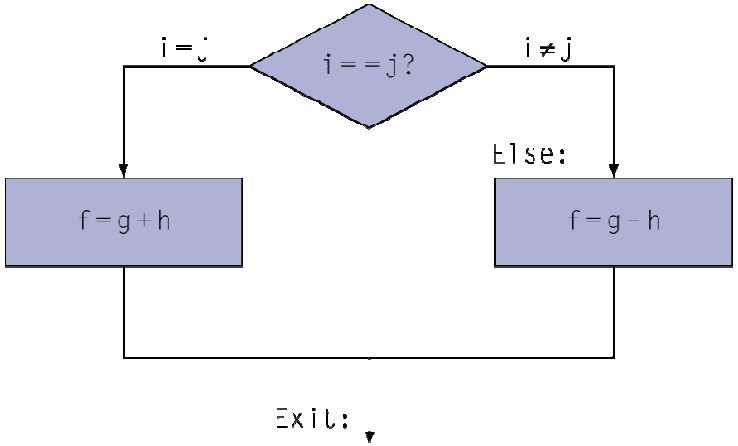
- Skok do rozkazu poprzedzonego etykietą jeśli warunek jest prawdziwy
  - Wpp. wykonuj kolejny rozkaz
- `beq rs, rt, L1`
  - jeśli (rs == rt) skocz do rozkazu etyk. L1;
- `bne rs, rt, L1`
  - jeśli (rs != rt) skocz do rozkazu o etyk. L1;
- `j L1`
  - bezwarunkowo skocz do L1

Przekład IF na MIPS

- Kod C:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … w $s0, $s1, …



- Kod MIPS:

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j   Exit
Else:   sub $s0, $s1, $s2
Exit:   …
```

Asembler wylicza adres

Pętle

- Kod C:

```
while (save[i] == k) i += 1;
```

  - i w $s3, k w $s5, adres save w $s6
- Kod MIPS:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Inne rozkazy warunkowe

- Ustaw 1 jeśli warunek prawdziwy
  - W przeciwnym wypadku ustaw 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Używane zazwyczaj razem z `beq`, `bne`

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Uwagi o skokach

- Dlaczego nie ma `blt`, `bge`, …?
- Sprawdzenie <, ≥, … wolniejsze niż =, ≠
  - W połączeniu ze skokiem wymaga więcej pracy na pojedynczą istrukcję, co może wymagać zwolnienia zegara
  - Wszystkie rozkazy wykonywałyby się wolniej
- `beq` i `bne` są potrzebne najczęściej
- Przykład dobrego kompromisu

Liczby ze znakiem i bez

- Uzupełnień do 2: `slt`, `slti`
- Bez znaku (nat. kod bin.): `sltu`, `sltui`
- Przykład
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1`
    - –1 < +1 ⇒ $t0 = 1
  - `sltu $t0, $s0, $s1`
    - +4,294,967,295 > +1 ⇒ $t0 = 0

Stałe 32-bitowe

- Większość stałych jest mała
  - 16-bitowe stałe wystarczają
- Czasem potrzeba stałej 32-bitowej

  `lui rt, constant`
  - Wstawia 16-bitową stałą do bardziej znaczącej części rejestru rt
  - Pozostałe bity rejestru ustawia na 0

```
lui $s0, 61
```

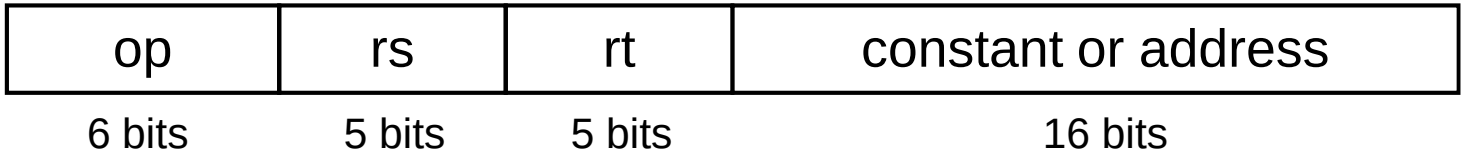
0000 0000 0111 1101	0000 0000 0000 0000

```
ori $s0, $s0, 2304
```

0000 0000 0111 1101	0000 1001 0000 0000

Adresy w skokach branch

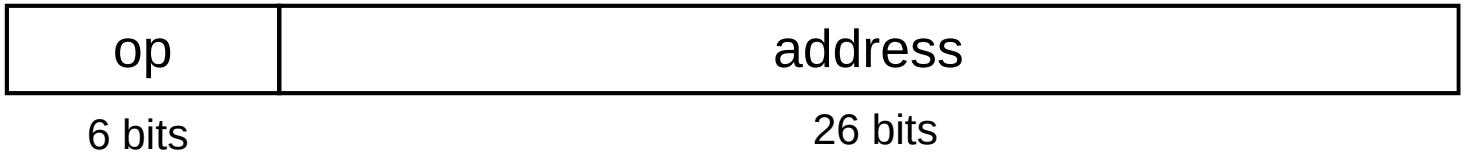
- Rozkaz skoku (typ I)
  - Opcode, dwa rejestry, adres skoku
- Zazwyczaj potrzebna ,,krótkie skoki'' do przodu lub do tyłu



  - Adresowanie względem PC
    - Adres skoku = PC + offset × 4
    - PC zwiększony o 4 przed obliczeniem adr.

Adresy w skokach Jump

- Skoki bezwarunkowe (`j`, `jal` – ten drugi poznamy wkrótce) skaczą „daleko”
  - Pełny adres zakodowany w rozkazie



- Adresowanie (pseudo)bezpośrednie:
  - Adres docelowy = $PC_{31...28}$: (address × 4)

Przykład adresowania

- Pętla z jednego z przykładów
  - Zakładamy, że Loop etykietuje 80000

```
Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: …                     80024
```

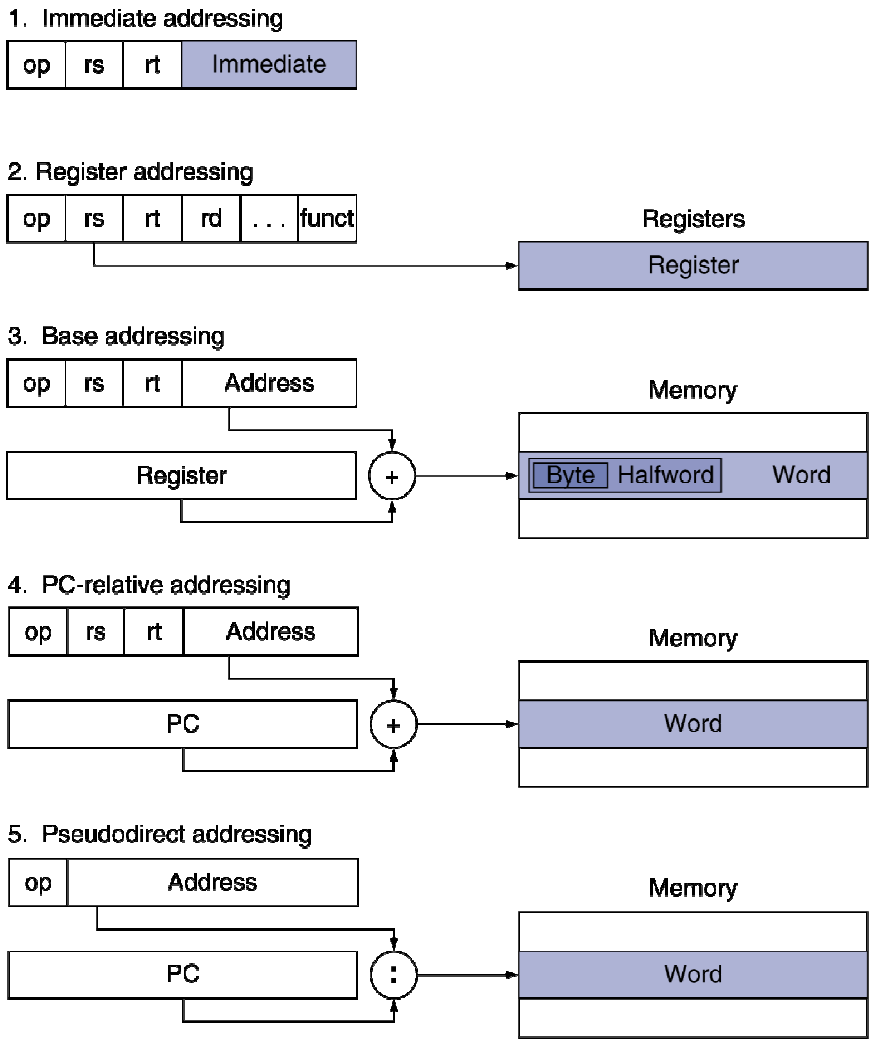
0	0	19	9	4	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

Dalekie „branche”

- Jeśli adres docelowy jest daleki asembler przeorganizuje kod.
- Przykład:

```
        beq $s0,$s1, L1
              ↓
        bne $s0,$s1, L2
        j L1
  L2: …
```

Tryby adresowania MIPS



Symulator SPIM

- Symulator architektury MIPS-32
- http://pages.cs.wisc.edu/~larus/spim.html

Przykład kodu dla SPIM-a

```
# Dodawanie elementow tablicy (przyklad zaczerpniety z
# ,,Krotkiego wprowadzenia do SPIM-a" autorstwa Salah A. Almajdouba
    .text
    .align 2
    .globl main
main:
        lw $s0, Size          # czyta rozmiar tablicy
        li $s1, 0             # indeks i
        li $s2, 0             # s2 bedzie przechowywac sume
        li $t2, 4             # t2 zawiera stala 4
loop:   mul $t1, $s1, $t2     # t1 dostaje i * 4
        lw $s3, Nstart($t1)   # s3 = N[i]
        add $s2, $s2, $s3     # sum = sum + N[i]
        addi $s1, $s1, 1      # i = i + 1
        beq $s1, $s0, STOR    # skaczemy do  STOR na koniec
        j loop
STOR:   sw $s2, Result        # wynik zachowujemy w Result
     .data
    .align 2
Nstart: .word 8, 25, -5, 55, 33, 12, -78
Size:   .word 7
Result: .word 0
```

Uwagi do kodu

- Dyrektywa **.text** oznacza, że kolejne dane należy umieszczać w segmencie programu
- Dyrektywa **.date** oznacza, że kolejne dane należy umieszczać w segmencie danych
- Dyrektwya **.align** 2 nakazuje umieszczać dane z wyrównaniem do 2^2
- **.word** każe umieszczać kolejne dane na 4 bajtach (jest też dyrektywa .byte)
- **.globl** mówi, że symbol jest globalny (widzialny z innych plików)
- **li** (*load immediate*} nie jest rozkazem, a tzw. pseudorozkazem -- podczas tłumaczenia na kod maszynowy przekładane jest na ciąg rozkazów (lub czasem pojedynczy rozkaz). Np. **li $s0, 4** przekładane jest na **ori $s0, $zero, 4**.
- Podobnie pseudorozkazem jest **mul**. Pseudorozkaz **mul $s1, $s2, $s3** zostanie przełożony na **mult $s2, $s3**, **mflo $s1**. Rozkaz **mul** umieszcza bardziej znaczącą część wyniku w specjalnym rejestrze **HI**, mniej znaczącą w rejestrze **LO**. Rozkaz **mflo $s1** kopiuje zawartość **LO** do rejestru **$s1**.
- Konstrukcja **lw $s3, Nstart($t1)** rówznież nie jest tutaj pojedynczym rozkazem maszynowym; powodem jest występienie etykiety Nstart jako przesunięcia
- Podobnie jest z pierwszym rozkazem **lw $s0, Size**, który tłumaczony jest na: **lui $1, 4097**, **lw $16, 28($1)**; rejestr **$1** (**$at)** to specjalny rejestr używany przez asembler do przekładu pseudorozkazów na ,,prawdziwe'' rozkazy.
- W tym przykładzie użyta jest nieco inna filozofia odwołań do pól tablicy niż w przykładzie wcześniejszym: stała w odwołaniu **lw $s3, Nstart($t1)** oznacza początek tablicy, a rejstr -- przesunięcie.